Delay Differential Equations - EPSRC Summer Internship 2025

Eva Siney Jones

August 2025

1 Introduction

Research and interest in delay differential equations (DDEs) is increasing thanks to their use in more accurately modelling systems and predicting their behaviour [BMV14a].

While there is a widely-used package for DDE analysis in MATLAB - the package DDE-Biftool - code for Julia is not so readily used. In this internship, I looked into the theory behind DDEs systems with constant delays before writing and implementing code for analysing them. This included finding stability using two different methods, finding Hopf and fold bifurcations and the continuation of Hopf bifurcations over a 2-parameter space.

In this report, I start by introducing the key difference between ordinary differential equations (ODEs) and delay differential equations (DDEs) before following with common notation. I then explore more complicated ideas such as those behind finding equilibria, stability, bifurcations and continuation into a 2-parameter plane. To help demonstrate some of the theory discussed, I will introduce three examples of DDE systems. The examples are for a single-delayed inverted pendulum problem, the Mackey-Glass equation and an example representing the interaction between two neurons. I follow by giving an in-depth example of how my code can be implemented to analyse a DDE system. For this I use the neuron example.

1.1 ODEs vs DDEs

Ordinary differential equations (ODEs) and delay differential equations (DDEs) have very similar properties but the key difference is that DDEs depend on the history of the system while ODEs only require an initial condition at one point. This reliance on the past means that DDEs have infinite dimension while ODEs's dependence on a point makes their space finite dimensional. This makes analysis, such as stability analysis, of DDEs much more complex and computationally heavy in comparison to ODEs [Smi11a].

1.2 Terminology and notation

Since DDEs depend on the past, having the system's history is essential for solving and analysis. The notation x_t represents the history:

$$x_t(\theta) = x(t+\theta), -\tau \leq \theta \leq 0,$$

where x_t is the state of the system at time t [Smi11b] given the system's history. It is shorthand for the entire history segment of the function up to time t. The initial condition for a DDE is often notated as $x_\sigma = \phi$, $\phi(\theta) = x(\sigma+\theta), \theta \in [-\tau, 0]$ and gives all the values of x from time $\sigma - \tau$ up to σ. The notation $x_t(\sigma, \phi)$ is the state history at a time σ given that it started with history ϕ - essentially it is what the past of the solution looks like if you assume it started at time σ with the history ϕ.

We'll consider the general (autonomous) delay differential equations:

$$x'(t) = f(x_t, \eta), t \geq 0, t \in \mathbb{R},$$

with f representing the function(s) on the right-hand side of the DDE system, x the state(s) and where η are the parameters of the system.

A DDE system can have numerous delays. Note that the first delay τ_0 is always taken to be 0 in this project. These multiple delays can be constant (fixed) or they can be state-dependent, where the delay depends on the state of the system (its position) [Smi11c] and/or it can be parameter-dependent. Delays can be *discrete delays* or *distributed delays*. A delay is a discrete delay if $x(t)$ is a function of time [Smi11c] and the delay is fixed while a distributed delay is often a weighted average over several delays [Smi11c]. This project focuses on discrete (constant) delays.

1.3 Steady-states and equilibria

Consider the DDE system, with n fixed delays, in the given form:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}(t), \mathbf{x}(t-\tau_1), ..., \mathbf{x}(t-\tau_n)).$$

A delay differential equation is in steady state when it is constant for all time [Smi11a]. Since it is constant over all time then delays have no effect - the past, present and future are constant. A point x^* is an equilibrium (or steady-state) if it satisfies [BMV14b]:

$$\mathbf{f}(\mathbf{x}^*, \mathbf{x}^*, \mathbf{x}^*, ..., \mathbf{x}^*) = 0$$

1.4 Linearisation Principle

Consider the following DDE

$$x'(t) = f(x(t), x(t-\tau_1), ..., x(t-\tau_d)), x(t) \in \mathbb{R}^n,$$

where $x(t) \in \mathbb{R}^n$ is the state of the system at time t, d is the number of delays in the system and $\tau_i > 0$ are the delays for $i = 1, ...d$. Assume x^* is an equilibrium such that $f(x^*, x^*, ..., x^*) = 0$ is fulfilled. A non-linear DDE can be linearised around x^* such that the system is in a more manageable form, for instance to do stability analysis on. The general linearised form for a DDE system with d delays is:

$$\dot{u}(t) = A_0 u(t) + A_1 u(t-\tau_1) + ... + A_d u(t-\tau_d), \tag{1}$$

where A_i are the partial derivative matrices with respect to $x(t-\tau_i)$, $i = 0, .., d$ where $\tau_0 = 0$. For example if $\dot{x} = f(x) = a\sin(x(t)) - bx(t-\tau_1) + cx(t-\tau_2)$, it has an equilibrium point at $x^* = 0$. Then the partial derivative (1×1) matrices are:

$$A_0 = a\cos(x), \quad A_1 = -b, \quad A_2 = c$$

Hence the linearised form of this system around $x^* = 0$ is

$$\dot{u}(t) = a\cos(0)u(t) - bu(t-\tau_1) + cu(t-\tau_2),$$

meaning for this example $A_0 = a, A_1 = -b$ and $A_2 = c$.

The Linearisation Principle is key for finding the stability of equilibrium points [BMV14b].

1.5 Semigroup theory and the infinitesimal generator

The infinite nature of DDEs makes it hard to analyse the stability of solutions [Smi11a]. Linear(ised) DDE systems can be written in a way where semigroup theory can be used to help find the stability.

Let there be a linear DDE system with k discrete delays:

$$x'(t) = L_0 x(t) + \sum_{j=1}^{k} L_j x(t - \tau_j) \tag{2}$$

There exists an operator, called the *solution operator*, that describes what the DDE system looks like given it started with an initial function [BMV14b]. Given an initial function ϕ - the state the system started at - the solution operator $T(t), t > 0$ for the system (2) with the initial function ϕ is defined as $T(t)\phi = x_t$ [BMV09], where x_t is the current state of the system given that it started with state ϕ at $t = 0$. The family $\{T(t)\}_{t\geq 0}$ of solution operators is a C_0-semigroup [BMV09]. A C_0-semigroup satisfies the following [BMV14b]:

- *semigroup properties*: $T(0) = I$, where I is the identity operator, and $T(t+s) = T(s)T(t)$ for all $t, s \geq 0$ (evolving over s time units first then by t time units is the same as evolving by $t + s$)
- *strong continuity property*: for any $\phi \in X$, where X is a Banach space, we have $||T(t)\phi - \phi||_X \to 0$ as $t \to 0$ [BMV14b]

Assume $\{T(t)\}_{t\geq 0}$ is a C_0-semigroup. There exists an operator, called the *infinitesimal generator*, that describes the derivative of $T(t)$ at $t = 0$ [BMV14b]. The infinitesimal generator, $\mathcal{A} : \mathcal{D}(\mathcal{A}) \subseteq X \to X$, for system (2), is defined as [BMV09]:

$$\mathcal{A}\phi = \phi',$$

where $\mathcal{D}(\mathcal{A}) = \{\phi \in X : \phi' \in X, \phi'(0) = L_0\phi(0) + \sum_{j=1}^{k} L_j\phi(-\tau_j)\}$ [BMV14b]. This operator can be used to rewrite (2) into the following (on the state space X) [BMV09]:

$$\begin{cases} \frac{du}{dt}(t) = \mathcal{A}u(t), & t > 0, \\ u(0) = \phi \in X, \end{cases}$$

which has the solution $u(t) = x_t$ whenever $\phi \in \mathcal{D}(\mathcal{A})$ [BMV09].

The reason why the infinitesimal generator is particularly useful is that it writes the DDE in an infinite-dimensional system but in a way that by looking at the spectrum of $\mathcal{A}$ the stability of the DDE system can be found [BMV14b]. The spectrum of $\mathcal{A}$ are the set of points where $(\lambda I - \mathcal{A})$ fails to be bijective [BMV09]. So the spectrum of $\mathcal{A}$ are the points λ where there exists v such that $(\lambda I - \mathcal{A})v = 0$ - i.e. they are the roots of the characteristic equation of the DDE system. Hence the spectrum can be used to deduce the stability of the system.

2 Examples

During this project I focused on three examples to demonstrate how my Julia code works. Two of these examples (the Mackey-Glass equation and the neuron example) are also demos for MATLAB's DDE-Biftool package, while the inverted pendulum example is one I looked at and compared to known results found in the sources [KS23] and [KS04].

2.1 Inverted pendulum example

For simplicity, we'll look at the inverted pendulum example with a single constant delay (τ) in the simplified form of:

$$\begin{aligned}\dot{x} &= v(t)\\ \dot{v} &= \sin(x(t)) - ax(t-\tau) - bv(t-\tau),\end{aligned}$$

where x is the angle by which the pendulum deviates from its starting upright position and v is the angular velocity [KS23]. The delay, τ, models the reaction delay to right the falling pendulum.

2.2 Mackey-Glass equation

The Mackey-Glass equation considers a population of mature circulating (blood) cells, $x(t)$, and models the process of production of blood cells using the following DDE system:

$$\dot{x} = \frac{\beta x(t-\tau)}{1+[x(t-\tau)]^n} - \gamma x(t) \tag{3}$$

with parameters $\beta, \gamma, n > 0$. The parameter β expresses the dependence of blood cell production on the number of mature cells in the system, while γ represents the death rate of the blood cells [MR14]. The parameter n is used to capture non-linearity of blood cell production. A DDE is used because there is a finite time delay, τ, between when the blood cells are produced in the bone marrow to when they actually start to circulate in the blood system (their release). The important thing to note is that in the Mackey-Glass equation the condition $\beta \geq \gamma$ is needed for real solutions.

2.3 Neuron example

The interactions between two neurons are another system that can be modelled using delay differential equations. This neuron example is the main demo used to show how DDE-Biftool can be used. The DDE system is:

$$\begin{aligned}\dot{x_1}(t) &= -\kappa x_1(t) + \beta \tanh(x_1(t-\tau_s)) + a_{12}\tanh(x_2(t-\tau_2))\\ \dot{x_2}(t) &= -\kappa x_2(t) + \beta \tanh(x_2(t-\tau_s)) + a_{21}\tanh(x_1(t-\tau_1))\end{aligned}$$

where x_1, x_2 are the neurons, τ_1 and τ_2 are the delays between the two neurons and where τ_s is the delay between each neuron getting a stimulus and reacting to that stimulus.

3 Finding equilibria - examples

3.1 Inverted pendulum example

As discussed before, if $\mathbf{x}^*$ is an equilibrium point then it must satisfy $f(\mathbf{x}^*, ..., \mathbf{x}^*) = 0$ so for the inverted pendulum example it must satisfy:

$$\begin{aligned}v^* &= 0\\ \sin(x^*) - ax^* - bv^* &= 0,\end{aligned}$$

So for any equilibrium point v will always equal $v^* = 0$ and we have that x^* must fulfill:

$$\begin{aligned}&\implies sin(x^*) - ax^* = 0\\ &\implies a = \frac{sin(x^*)}{x^*}\end{aligned}$$

3.2 Mackey-Glass equation

The equilibria of the Mackey-Glass equation must satisfy:

$$\begin{aligned} & \beta \frac{x^*}{1+(x^*)^n} - \gamma x^* = 0 \\ \implies & \beta x^* - \gamma x^* - \gamma (x^*)^{n+1} = 0 \\ \implies & x^*(\beta - \gamma - \gamma (x^*)^n) = 0 \\ \implies & x^* = 0, \beta - \gamma = \gamma (x^*)^n \\ \implies & x^* = 0, x^* = \left(\frac{\beta - \gamma}{\gamma}\right)^{\frac{1}{n}} \end{aligned}$$

so we get the equilibrium points $x^* = 0$ and $x^* = \left(\frac{\beta-\gamma}{\gamma}\right)^{\frac{1}{n}}$.

3.3 Neuron example

Since for an equilibrium point, $\mathbf{x}^*$, we need $f(\mathbf{x}^*, ..., \mathbf{x}^*) = \mathbf{0}$, we have that $\mathbf{x}^* = (x_1^*, x_2^*) = (0, 0)$ is an equilibrium point for the neuron example.

4 Stability

Methods for determining the stability of delay differential equations are similar to determining the stability of ordinary differential equations [Smi11a]. However, the key difference is that the infinite-dimensional property of DDEs create complications so the methods used for ODEs have to be adapted or new methods found for determining DDE stability. In this section I discuss the background information around stability and the two methods I explored during this project - finding stability using large matrix and a method that uses barycentric interpolation as a foundation.

4.1 Characteristic equation

As with ODEs, we start by finding the characteristic equation by substituting the solution $u = ve^{\lambda t}$ into the linearised system (1). For an ODE the characteristic equation simplifies nicely into a manageable and finite-dimensional form while the DDE produces the following:

$$\implies \lambda v e^{\lambda t} = A_0 v e^{\lambda t} + A_1 v e^{\lambda t} e^{-\lambda \tau_1} + ... + A_d v e^{\lambda t} e^{-\lambda \tau_d} \tag{4}$$

$$\implies [\lambda I - A_0 - A_1 e^{-\lambda \tau_1} - ... - A_d e^{-\lambda \tau_d}] v = 0 \tag{5}$$

$$\implies det(\lambda I - A_0 - A_1 e^{-\lambda \tau_1} - ... - A_d e^{-\lambda \tau_d}) = 0 \tag{6}$$

where the last equation is the characteristic equation. The transcendental parts of the equation ($e^{-\lambda \tau_i}$), created by the delays means that this equation is significantly harder to solve than for the simpler ODE ones due to the infinite-dimensional nature of the transcendental equation. The stability of a point is found by considering the eigenvalues of the characteristic equation. It is found that there are a finite number of roots of the characteristic equation that have a real part bigger than some negative real number γ [Smi11a]. This idea means that we only need to focus on a finite number of roots to gain the stability.

To counteract the infinite-dimensional property of the characteristic equation, it can be simplified to a finite-dimensional system so that typical ODE methods can be used. Two methods of finding DDE stability are discussed below.

4.2 Approximating stability (using a large matrix)

One way to counteract the complexity of DDE stability analysis is to approximate it using a large (finite-dimensional) matrix. The created finite-dimensional system for a 1-delay DDE system is:

$$\lambda \begin{bmatrix} v_0 \\ v_1 \\ . \\ . \\ . \\ v_m \end{bmatrix} = \begin{bmatrix} A_0 & 0I_n & ... & 0I_n & A_1 \\ \frac{m}{\tau}I_n & -\frac{m}{\tau}I_n & 0I_n & ... & 0I_n \\ 0I_n & . & . & 0I_n & 0I_n \\ 0I_n & 0I_n & . & . & 0I_n \\ 0I_n & 0I_n & ... & \frac{m}{\tau}I_n & -\frac{m}{\tau}I_n \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ . \\ . \\ . \\ v_m \end{bmatrix}, \tag{7}$$

where n is the number of equations in the DDE system (number of states), m are the number of steps we want to discretise the system over and where I_n is the identity matrix with dimensions $n \times n$. This system gives the equations:

$$\begin{aligned} \lambda v_0 &= A_0 v_0 + A_1 v_m \\ \lambda v_1 &= \frac{m}{\tau} v_0 - \frac{m}{\tau} v_1 \\ \lambda v_2 &= \frac{m}{\tau} v_1 - \frac{m}{\tau} v_2 \\ &... \\ \lambda v_m &= \frac{m}{\tau} v_{m-1} - \frac{m}{\tau} v_m \end{aligned}$$

Rearranging gives:

$$\begin{aligned} \lambda v_0 &= A_0 v_0 + A_1 v_m \\ \frac{\tau\lambda}{m} v_1 &= v_0 - v_1 \\ \frac{\tau\lambda}{m} v_2 &= v_1 - v_2 \\ &... \\ \frac{\tau\lambda}{m} v_m &= v_{m-1} - v_m \end{aligned}$$

We can arrange for v_m:

$$\begin{aligned} & \frac{\tau\lambda}{m} v_m = v_{m-1} - v_m \\ \implies & \left[1 + \frac{\lambda\tau}{m}\right] v_m = v_{m-1} \\ \implies & v_m = \left[1 + \frac{\lambda\tau}{m}\right]^{-1} v_{m-1} \\ \implies & v_m = \left[1 + \frac{\lambda\tau}{m}\right]^{-m} v_0 \end{aligned}$$

The exponential, e^x, can be approximated by:

$$e^x \approx \lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n \tag{8}$$

so,

$$\begin{aligned}\lim_{m\to\infty}\left(1+\frac{\lambda\tau}{m}\right)^{-m} &= \lim_{m\to\infty}\left[\left(1+\frac{\lambda\tau}{m}\right)^{m}\right]^{-1}\\ &\approx [e^{\lambda\tau}]^{-1}\\ &= e^{-\lambda\tau}\end{aligned}$$

hence we can approximate v_m such that $v_m \approx e^{-\lambda\tau}v_0$. So,

$$\begin{aligned}&\lambda v_0 = A_0v_0 + A_1v_m\\ \implies &[\lambda I - A_0]v_0 - A_1v_m = 0\\ \implies &[\lambda I - A_0]v_0 - A_1\left[1+\frac{\lambda\tau}{m}\right]^{-m}v_0 = 0\\ \implies &\left[\lambda I - A_0 - A_1\left[1+\frac{\lambda\tau}{m}\right]^{-m}\right]v_0 = 0\\ \approx &\left[\lambda I - A_0 - A_1e^{-\lambda\tau}\right]v_0 = 0\end{aligned}$$

Hence, equation (5) and the DDE system's equilibrium's stability can be approximated by using this large matrix method, where v_0 is the first eigenvector of the $m+1$ eigenvectors of the large matrix. The stability is determined by looking at the real part of the eigenvalues on the large matrix in (7) - if any are greater than 0 then the point is unstable.

For the a 2-delay DDE system, the finite-dimensional matrix system is:

$$\lambda\begin{bmatrix} v_0\\ v_1^{(1)}\\ v_2^{(1)}\\ .\\ .\\ .\\ v_m^{(1)}\\ v_1^{(2)}\\ .\\ .\\ .\\ v_m^{(2)}\end{bmatrix} = \begin{bmatrix} A_0 & 0I_n & ... & ... & 0I_n & A_1 & 0I_n & ... & ... & A_2\\ \frac{m}{\tau_1}I_n & -\frac{m}{\tau_1}I_n & 0I_n & ... & 0I_n & 0I_n & ... & & ... & 0I_n\\ 0I_n & \frac{m}{\tau_1}I_n & -\frac{m}{\tau_1}I_n & 0I_n & ... & & ... & & ... & 0I_n\\ 0I_n & 0I_n & . & . & 0I_n & ... & & & ... & 0I_n\\ 0I_n & ... & 0I_n & . & . & 0I_n & ... & & ... & 0I_n\\ 0I_n & ... & ... & 0I_n & \frac{m}{\tau_1}I_n & -\frac{m}{\tau_1}I_n & 0I_n & ... & ... & 0I_n\\ \frac{m}{\tau_2}I_n & 0I_n & ... & & ... & 0I_n & -\frac{m}{\tau_2}I_n & 0I_n & ... & 0I_n\\ 0I_n & ... & & & ... & 0I_n & \frac{m}{\tau_2}I_n & -\frac{m}{\tau_2}I_n & 0I_n & 0I_n\\ 0I_n & ... & & & & ... & 0I_n & . & . & 0I_n\\ 0I_n & ... & & & & & ... & 0I_n & \frac{m}{\tau_2}I_n & -\frac{m}{\tau_2}I_n\end{bmatrix}\begin{bmatrix} v_0\\ v_1^{(1)}\\ .\\ .\\ .\\ v_m^{(1)}\\ v_1^{(2)}\\ .\\ .\\ .\\ v_m^{(2)}\end{bmatrix} \tag{9}$$

This gives the (rearranged) equations:

$$\begin{aligned}v_0 &= \left[1+\frac{\lambda\tau_1}{m}\right]v_1^{(1)}\\ v_1^{(1)} &= \left[1+\frac{\lambda\tau_1}{m}\right]v_2^{(1)}\\ &...\\ v_{m-1}^{(1)} &= \left[1+\frac{\lambda\tau_1}{m}\right]v_m^{(1)}\end{aligned}$$

and,

$$\begin{aligned} v_0 &= \left[1 + \frac{\lambda\tau_2}{m}\right] v_1^{(2)} \\ v_1^{(2)} &= \left[1 + \frac{\lambda\tau_2}{m}\right] v_2^{(2)} \\ &... \\ v_{m-1}^{(2)} &= \left[1 + \frac{\lambda\tau_2}{m}\right] v_m^{(2)} \end{aligned}$$

Some more rearranging gives,

$$\begin{aligned} v_0 &= \left[1 + \frac{\lambda\tau_1}{m}\right]\left[1 + \frac{\lambda\tau_1}{m}\right]...\left[1 + \frac{\lambda\tau_1}{m}\right] v_m^{(1)} \\ \implies v_m^{(1)} &= \left[1 + \frac{\lambda\tau_1}{m}\right]^{-m} v_0 \end{aligned}$$

and,

$$\begin{aligned} v_0 &= \left[1 + \frac{\lambda\tau_2}{m}\right]\left[1 + \frac{\lambda\tau_2}{m}\right]...\left[1 + \frac{\lambda\tau_2}{m}\right] v_m^{(2)} \\ \implies v_m^{(2)} &= \left[1 + \frac{\lambda\tau_2}{m}\right]^{-m} v_0 \end{aligned}$$

Again using Equation (8), we have that:

$$\begin{aligned} v_m^{(1)} &\approx e^{-\lambda\tau_1} v_0 \\ v_m^{(2)} &\approx e^{-\lambda\tau_2} v_0 \end{aligned}$$

Hence, we can use this large matrix method to approximate eigenvalues and get a good guess of the equilibrium stability as:

$$\begin{aligned} \lambda v_0 &= A_0 v_0 + A_1 v_m^{(1)} + A_2 v_m^{(2)} \\ &\approx A_0 v_0 + A_1 e^{-\lambda\tau_1} v_0 + A_2 e^{-\lambda\tau_2} v_0 \end{aligned}$$

So we can then look at the eigenvalues of

$$[\lambda I - A_0 - A_1 e^{-\lambda\tau_1} - A_2 e^{-\lambda\tau_2}] v_0 = 0$$

and estimate the stability of our equilibria. The stability is determined by looking at the real value of the eigenvalues on the large matrix in (9) - if any are greater than 0 then the point is unstable.

This method of finding stability is in my function `stab_func_matrix`, whose structure and function is explained further in subsection 8.1.6.

4.3 Stability using the method described in Breda et al 2009 paper [BMV09]

While using a large matrix to approximate the stability of equilibria works, it can be expensive to compute for large m, which is required for better accuracy. Another method to find the stability of equilibria was put forward in [BMV09]. Their method was built on the idea of (barycentric) interpolation and approximating the infinitesimal generator $\mathcal{A}$ by a matrix that incorporated this.

4.3.1 Barycentric interpolation

Given $N+1$ distinct nodes, x_j for $j = 1, ..., N+1$, with numbers f_j corresponding the value of x_j for a given (or unknown) function f, a polynomial p (of degree at most N) can be found such that p interpolates f at the points x_j (i.e. $p(x_j) = f_j$) [BT04]. This interpolation polynomial p can be used to evaluate desired points for the function f.

The common way the solution can be written is in the *Lagrange form* [BT04]:

$$p(x) = \sum_{j=1}^{N+1} f_j \ell_j(x), \qquad \ell_j(x) = \frac{\prod_{k=1,k\neq j}^{N+1}(x - x_k)}{\prod_{k=1,k\neq j}^{N+1}(x_j - x_k)},$$

where ℓ_j is the *Lagrange polynomial* corresponding to node x_j.

However, as discussed in [BT04], in this form each evaluation of $p(x)$ requires $O(N^2)$ additions and multiplications, and adding a new node and it's corresponding f value means starting from scratch. This is all while the computation is numerically unstable. To combat this, [BT04] rewrote $p(x)$ into a more useful form, known as the *barycentric formula* for p:

$$p(x) = \frac{\sum_{j=1}^{N+1} \frac{w_j}{x-x_j} f_j}{\sum_{j=1}^{N+1} \frac{w_j}{x-x_j}}, \tag{10}$$

where w_j are the *barycentric weights*, defined as:

$$w_j = \frac{1}{\prod_{k\neq j}(x_j - x_k)}, \quad j = 1, ..., N+1$$

This new form means the number of operations required to evaluate or add a new node decreases compared to the Lagrange form, making computation less expensive.

Equation (10) can be rearranged to give the barycentric representation of ℓ_j [BT04]:

$$\ell_j(x) = \frac{\frac{w_j}{(x-x_j)}}{\sum_{k=1}^{N+1} \frac{w_k}{x-x_k}} \tag{11}$$

This form is found by:

$$\begin{aligned} p(x) =& \frac{\sum_{j=1}^{N+1} \frac{w_j}{x-x_j} f_j}{\sum_{j=1}^{N+1} \frac{w_j}{x-x_j}} \\ =& \frac{\sum_{j=1}^{N+1} f_j \cdot \frac{w_j}{x-x_j}}{\sum_{k=1}^{N+1} \frac{w_k}{x-x_k}} \\ & \text{since } \sum_{k=1}^{N+1} \frac{w_k}{x - x_k} \text{ is a constant} \\ =& \sum_{j=1}^{N+1} f_j \cdot \frac{\frac{w_j}{x-x_j}}{\sum_{k=1}^{N+1} \frac{w_k}{x-x_k}} \\ =& \sum_{j=1}^{N+1} f_j \cdot \ell_j(x) \end{aligned}$$

Hence,

$$\ell_j(x) = \frac{\frac{w_j}{x-x_j}}{\sum_{k=1}^{N+1} \frac{w_k}{x-x_k}}$$

The (11) form of ℓ_j is key for finding the first-order *differentiation matrices* $D^{(1)}$ [BT04]:

$$D^{(1)}_{ij} = \ell'_j(x_i),$$

where

$$\ell'_j(x_i) = \begin{cases} \ell'_j(x_i) = \frac{w_j/w_i}{x_i - x_j}, & \text{for } i \neq j \\ \ell'_j(x_j) = -\sum_{i \neq j} \ell'(x_j), & \text{for } i = j \end{cases}$$

The derivation of these results can be found in detail in [BT04]. The interpretation of the matrix $D^{(1)}$ is that for a vector, f, made up of the f_j values, the product $D^{(1)}f$ returns the derivative of the interpolant at the interpolation nodes x_j.

The form (11) is also the form used in my `j_eval` function, which is introduced in subsection 4.4.1 and explained further in the subsection 8.1.8.

4.4 Breda et al 2009 method

Let there be a linear(ised) DDE system with n discrete delays and m functions with the form:

$$\dot{u} = A_0 u(t) + A_1 u(t - \tau_1) + ... + A_n u(t - \tau_n) \tag{12}$$

The stability idea put forward by D. Breda, S. Maset and R. Vermiglio is underpinned by the theory given in the above subsection. They approximated the infinitesimal generator, $\mathcal{A}$, with a matrix called the *spectral differentiation matrix* [BMV09], $\mathcal{A}_N$. The matrix $\mathcal{A}_N$ has the components:

$$\mathcal{A}_N = \begin{pmatrix} a_1 & a_2 & ... & a_{N+1} \\ d_{2,1} & d_{2,2} & ... & d_{2,N+1} \\ . & . & . & . \\ . & . & ... & . \\ . & . & . & . \\ d_{N+1,1} & d_{N+1,2} & ... & d_{N+1,N+1} \end{pmatrix},$$

where $a_j = A_0\ell_j(0) + A_1\ell_j(-\tau_1) + ... + A_n\ell_j(-\tau_n)$ (for Equation (12)) and where $d_{i,j}, i = 2, ..., N+1, j = 1, ..., N+1$ are the components of the first-order differentiation matrix $D^{(1)}$. Note here that the interpolation points would be t_j rather than x_j as our system depends on t and not x.

The stability is found by looking at the eigenvalues of $\mathcal{A}_N$ to see if any have real parts greater than 0 to indicate any instability.

For this project, I created a function `stab_func_DDE` that uses this method to determine stability (note that the function's current form has only been trialled on DDE systems with constant delays). The function `stab_func_DDE` is explained further in the following subsection.

4.4.1 `stab_func_DDE`

The `stab_func_DDE` function has the following input arguments:

- `A` which are matrices of the (linearised) DDE system - they should be linearised around the equilibrium point the user wishes to find the stability of. This argument should be written in the form of a vector of matrices.
- `taus` are the delays. This argument should be written in vector form.
- `N` is the number of nodes for interpolation the user wishes to have (so you get N+1 nodes) - note the function uses Chebyshev points of the 2nd kind for the interpolation points

- Optional argument: `eigvecs` tells the function to output the eigenvectors (in matrix form, where the i-th column are the eigenvectors of i-th eigenvalue); default is `0` (eigenvectors not outputted)

The `stab_func_DDE` function outputs the following:

- `stab` is the stability of the (equilibrium) point (gives `1` if the point is stable, `0` if the point is unstable)
- `sm_eigvals` are the eigenvalues of the point
- If asked: `sm_eigvecs` are the eigenvectors of the eigenvalues

The function `stab_func_DDE` uses two functions: `j_diff` and `j_eval`. The function `j_diff` is used to find the first-order differentiation matrix $D^{(1)}$ for the interpolation points. The function `j_eval` is used to find the values of the function f at some desired points x_e (and when asked, it can find the nth-order derivative of f at the points x_e). The function `j_eval` outputs a matrix such that when multiplied by the vector $\mathbf{f} = (f_1, ...f_{N+1})$ it will return the value of $f(x_e)$ (or the nth derivative of f at the points x_e).

An example of `stab_func_DDE` in use can be found in Section 8. For further information and more examples on how `stab_func_DDE` works please see '`Stability_finding_for_DDEs.ipynb`'.

4.5 Inverted Pendulum - 1 delay example

The system for the single (constant) delayed inverted pendulum is:

$$\begin{aligned}\dot{x} &= v(t)\\ \dot{v} &= sin(x(t)) - ax(t-\tau) - bv(t-\tau),\end{aligned}$$

The matrices of partial derivatives are:

$$A_0 = \begin{bmatrix} \frac{\partial f_1}{\partial x_t} & \frac{\partial f_1}{\partial v_t} \\ \frac{\partial f_2}{\partial x_t} & \frac{\partial f_2}{\partial v_t} \end{bmatrix}, A_1 = \begin{bmatrix} \frac{\partial f_1}{\partial x_\tau} & \frac{\partial f_1}{\partial v_\tau} \\ \frac{\partial f_2}{\partial x_\tau} & \frac{\partial f_2}{\partial v_\tau} \end{bmatrix}$$

so we have the partial derivative matrices:

$$A_0 = \begin{bmatrix} 0 & 1 \\ cos(x(t)) & 0 \end{bmatrix}, A_1 = \begin{bmatrix} 0 & 0 \\ -a & -b \end{bmatrix}$$

The system can be linearised so it takes the form:

$$\dot{u}(t) = A_0 u(t) + A_1 u(t-\tau)$$

4.5.1 Large matrix stability

The stability of the equilibria can be approximated by using the large matrix method for 1 delay, whose system is given in (7). The vectors v_i, for $i = 0, ...m$, are vectors with length equal to the number of states (for inverted pendulum it is two (x and v)), m is the number of (discretised) steps to be used in the matrix. The matrix equation is given below:

$$\lambda \begin{bmatrix} v_{0,1} \\ v_{0,2} \\ v_{1,1} \\ v_{1,2} \\ . \\ . \\ . \\ v_{m,1} \\ v_{m,2} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & .. & 0 & 0 & 0 \\ \cos(x^*) & 0 & 0 & ... & 0 & -a & -b \\ \frac{m}{\tau} & 0 & -\frac{m}{\tau} & 0 & ... & 0 & 0 \\ 0 & \frac{m}{\tau} & 0 & -\frac{m}{\tau} & 0 & ... & 0 \\ 0 & 0 & . & 0 & . & 0 & 0 \\ 0 & ... & 0 & . & 0 & . & 0 \\ 0 & ... & 0 & 0 & \frac{m}{\tau} & 0 & -\frac{m}{\tau} \end{bmatrix} \begin{bmatrix} v_{0,1} \\ v_{0,2} \\ v_{1,1} \\ v_{1,2} \\ . \\ . \\ . \\ v_{m,1} \\ v_{m,2} \end{bmatrix},$$

where $v_{i,j}$ is the jth component of the i-th vector.

4.5.2 Breda et al 2009 stability

The arguments to put into `stab_func_DDE` are:

- `A` is $\left[\begin{pmatrix} 0 & 1 \\ \cos(x^*) & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ -a & -b \end{pmatrix}\right]$
- `taus` is $[\tau]$
- `N` is the number of interpolation points the user wishes to have. In the Jupyter notebook demo ('`Single_delayed_inverted_pendulum_example.ipynb`') for the inverted pendulum example $N = 50$.

4.6 Mackey-Glass equation

As mentioned earlier for the Mackey-Glass equation we get the equilibrium points $x^* = x_1^* = 0$ and $x^* = x_2^* = \left(\frac{\beta-\gamma}{\gamma}\right)^{\frac{1}{n}}$. In [GB24], the following theorem is given:

Theorem 1 *Consider the Mackey-Glass equation (3):*

- *The equilibrium point $x_1^* = 0$ is unstable $\forall \tau \geq 0$.*
- *If $\frac{\gamma}{\beta} \geq 1 - \frac{2}{n}$ then, x_2^* is asymptotically stable $\forall \tau \geq 0$.*
- *If $\frac{\gamma}{\beta} < 1 - \frac{2}{n}$ then we get*

$$\tau_* = \frac{1}{\gamma\sqrt{\frac{(\gamma-\beta)^2 n^2}{\beta^2} + 2\frac{(\gamma-\beta)n}{\beta}}} \arccos\left(\frac{\beta}{(\gamma-\beta)n+\beta}\right)$$

such that for $0 \leq \tau \leq \tau_*, x_2^*$ is asymptotically stable and for $\tau > \tau_*$ x_2^* is unstable.

This is one way to find the stability but again the two methods discussed before can be used. To use either of the two methods, the partial derivatives of the Mackey-Glass equation for an equilibrium point, x^*, need to be found. These are:

$$A_0 = -\gamma, A_1 = -\beta\left[\frac{(n-1)[x^*]^n - 1}{(1+[x^*]^n)^2}\right]$$

so we have the linearised system in the form:

$$\dot{u}(t) = A_0 u(t) + A_1 u(t-\tau)$$

4.6.1 Large matrix stability

The Mackey-Glass has one delay and is 1-dimensional so the matrix system in Equation (7) for Mackey-Glass becomes:

$$\lambda \begin{bmatrix} v_0 \\ v_1 \\ \cdot \\ \cdot \\ \cdot \\ v_m \end{bmatrix} = \begin{bmatrix} -\gamma & 0 & ... & 0 & 0 & -\beta\left[\frac{(n-1)[x^*]^n-1}{(1+[x^*]^n)^2}\right] \\ \frac{m}{\tau} & -\frac{m}{\tau} & 0 & ... & 0 & 0 \\ 0 & . & . & 0 & ... & 0 \\ 0 & 0 & . & . & 0 & 0 \\ 0 & 0 & 0 & . & . & 0 \\ 0 & ... & 0 & 0 & \frac{m}{\tau} & -\frac{m}{\tau} \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ \cdot \\ \cdot \\ \cdot \\ v_m \end{bmatrix},$$

where vectors v_i are 1-dimensional. The eigenvalues of the large matrix dictate the stability of the equilibrium point x^*.

4.6.2 Breda et al 2009 stability

The arguments for the Mackey-Glass equation to put into `stab_func_DDE` are:

- `A` is $\left[(-\gamma), \left(-\beta\left[\frac{(n-1)[x^*]^n-1}{(1+[x^*]^n)^2}\right]\right)\right]$
- `taus` is $[\tau]$
- `N` is the number of interpolation points the user wishes to have. In the Jupyter notebook demo for Mackey-Glass, N=110 (see '`Mackey-Glass_demo.ipynb`' for the demo).

4.7 Neuron example

The system for modelling the interaction between two neurons being used is:

$$\begin{aligned} \dot{x_1}(t) &= -\kappa x_1(t) + \beta \tanh(x_1(t-\tau_s)) + a_{12}\tanh(x_2(t-\tau_2)) \\ \dot{x_2}(t) &= -\kappa x_2(t) + \beta \tanh(x_2(t-\tau_s)) + a_{21}\tanh(x_1(t-\tau_1)) \end{aligned}$$

The partial derivatives matrices A_0, A_1, A_2, A_3 for this system are given by:

$$A_0 = \begin{bmatrix} \frac{\partial f_1}{\partial (x_1)_t} & \frac{\partial f_1}{\partial (x_2)_t} \\ \frac{\partial f_2}{\partial (x_1)_t} & \frac{\partial f_2}{\partial (x_2)_t} \end{bmatrix}, A_1 = \begin{bmatrix} \frac{\partial f_1}{\partial (x_1)_{\tau_1}} & \frac{\partial f_1}{\partial (x_2)_{\tau_1}} \\ \frac{\partial f_2}{\partial (x_1)_{\tau_1}} & \frac{\partial f_2}{\partial (x_2)_{\tau_1}} \end{bmatrix}, A_2 = \begin{bmatrix} \frac{\partial f_1}{\partial (x_1)_{\tau_2}} & \frac{\partial f_1}{\partial (x_2)_{\tau_2}} \\ \frac{\partial f_2}{\partial (x_1)_{\tau_2}} & \frac{\partial f_2}{\partial (x_2)_{\tau_2}} \end{bmatrix}, A_3 = \begin{bmatrix} \frac{\partial f_1}{\partial (x_1)_{\tau_s}} & \frac{\partial f_1}{\partial (x_2)_{\tau_s}} \\ \frac{\partial f_2}{\partial (x_1)_{\tau_s}} & \frac{\partial f_2}{\partial (x_2)_{\tau_s}} \end{bmatrix}$$

So we have the partial derivative matrices:

$$A_0 = \begin{bmatrix} -\kappa & 0 \\ 0 & -\kappa \end{bmatrix}, A_1 = \begin{bmatrix} 0 & 0 \\ a_{21}sech^2(x_1(t-\tau_1)) & 0 \end{bmatrix}, A_2 = \begin{bmatrix} 0 & a_{12}sech^2(x_2(t-\tau_2)) \\ 0 & 0 \end{bmatrix},$$

$$A_3 = \begin{bmatrix} \beta sech^2(x_1(t-\tau_s)) & 0 \\ 0 & \beta sech^2(x_2(t-\tau_s)) \end{bmatrix}$$

Hence we have the linearised system in the form:

$$\dot{u}(t) = A_0u(t) + A_1u(t-\tau_1) + A_2u(t-\tau_2) + A_3u(t-\tau_s)$$

We'll focus on the equilibrium point $(x_1^*, x_2^*) = (0, 0)$ for this project.

4.7.1 Large matrix stability

The matrix equations (7) and (9) can be generalised for 3 delays, where v_i are 2-dimensional (for (x_1, x_2)). The stability of equilibria for the neuron example can be again found by looking at the real parts of the eigenvalues of the large matrix in the system.

4.7.2 Breda et al 2009 stability

For equilibrium point (0,0) and this neuron DDE system, the arguments for `stab_func_DDE` are:

- `A` is $\left[\begin{pmatrix} -\kappa & 0 \\ 0 & -\kappa \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ a_{21}sech^2(0) & 0 \end{pmatrix}, \begin{pmatrix} 0 & a_{12}sech^2(0) \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} \beta sech^2(0) & 0 \\ 0 & \beta sech^2(0) \end{pmatrix}\right]$
- `taus` is $[\tau_1, \tau_2, \tau_s]$
- `N` is the number of interpolation points the user wishes to have. In the Jupyter notebook demo for this neuron example N=30 (see '`neuron_example.ipynb`' for the demo).

5 Hopf bifurcation

The eigenvalues of the linearised system determine the equilibrium's stability. A Hopf bifurcation occurs when the stability behaviour changes and a pair of complex conjugate eigenvalues of the linearised system crosses the imaginary axis and become purely imaginary, i.e. $\lambda = \pm\omega i$. To help find the Hopf bifurcation, I created a function called `create_hopffunc` which finds an initial guess of the equilibrium and relevant Hopf values and also creates a function that helps to find the Hopf bifurcation. This section of the report focuses on the examples I explored during this project, more information on `create_hopffunc` can be found in Section 8.1.10.

5.1 Inverted Pendulum - 1 delay example

As seen in subsection 4.5, the system of the single-delayed inverted pendulum example can be linearised to take the form:

$$\dot{u}(t) = A_0 u(t) + A_1 u(t-\tau)$$

By using the solution u as $u(t) = ve^{\lambda t}$:

$$\begin{aligned} &\implies \lambda v e^{\lambda t} = A_0 v e^{\lambda t} + A_1 v e^{\lambda t} e^{-\lambda\tau} \\ &\implies [\lambda I - A_0 - A_1 e^{-\lambda\tau}]v = 0 \\ &\implies det(\lambda I - A_0 - A_1 e^{-\lambda\tau}) = 0 \end{aligned}$$

Notation note: we write

$$\Delta(\lambda) = \lambda I - A_0 - A_1 e^{-\lambda\tau} \tag{13}$$

There exists a steady-state solution at $x = 0, v = 0$:

$$\begin{aligned} &\implies det\left(\begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix} - \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} - \begin{bmatrix} 0 & 0 \\ -a & -b \end{bmatrix} e^{-\lambda\tau}\right) = 0 \\ &\implies det\left(\begin{bmatrix} \lambda & -1 \\ -1 + ae^{-\lambda\tau} & \lambda + be^{-\lambda\tau} \end{bmatrix}\right) = 0 \\ &\implies \lambda(\lambda + be^{-\lambda\tau}) - (-1)(-1 + ae^{-\lambda\tau}) = 0 \\ &\implies \lambda(\lambda + be^{-\lambda\tau}) + (ae^{-\lambda\tau} - 1) = 0 \\ &\implies \lambda^2 + b\lambda e^{-\lambda\tau} + ae^{-\lambda\tau} - 1 = 0 \\ &\implies ae^{-\lambda\tau} + b\lambda e^{-\lambda\tau} = 1 - \lambda^2 \end{aligned}$$

For Hopf bifurcation: $\lambda = \pm i\omega$. If $\lambda = \pm i\omega$ then $e^{-\lambda\tau} = e^{\pm\omega\tau i}$, for ease we'll discuss the example with $\lambda = +\omega i$. We have:

$$e^{-\lambda\tau} = e^{-\omega\tau i} = \cos(\omega\tau) - i\sin(\omega\tau)$$

Hence for $\lambda = +\omega i$,

$$\begin{aligned}
& ae^{-\lambda\tau} + b\lambda e^{-\lambda\tau} = 1 - \lambda^2 \\
\implies & a[\cos(\omega\tau) - i\sin(\omega\tau)] + b[\cos(\omega\tau) - i\sin(\omega\tau)]\omega i = 1 - (\omega i)^2 \\
\implies & \begin{cases} a\cos(\omega\tau) + b\omega\sin(\omega\tau) = 1 + \omega^2 \\ -a\sin(\omega\tau) + b\omega\cos(\omega\tau) = 0 \end{cases}
\end{aligned}$$

This can be used to find the true Hopf information and can be used to check `create_hopffunc`. By substituting the found values for the Hopf bifurcation into the final two equations it was seen in the demo '`Single_delayed_inverted_pendulum_demo.ipynb`' that `create_hopffunc` works well.

5.2 Mackey-Glass equation

The partial derivative matrices of the Mackey-Glass equation are:

$$A_0 = -\gamma, A_1 = -\beta\left[\frac{(n-1)[x(t-\tau)]^n - 1}{(1+[x(t-\tau)]^n)^2}\right]$$

so we have the linearised system in the form:

$$\dot{u}(t) = A_0 u(t) + A_1 u(t-\tau)$$

The solution we take for u is $u(t) = ve^{\lambda t}$

$$\begin{aligned}
& \implies \lambda v e^{\lambda t} = A_0 v e^{\lambda t} + A_1 v e^{\lambda t} e^{-\lambda\tau} \\
& \implies [\lambda I - A_0 - A_1 e^{-\lambda\tau}]v = 0 \\
& \implies det(\lambda I - A_0 - A_1 e^{-\lambda\tau}) = 0
\end{aligned}$$

There exists a steady-state solution at $x = 0$:

$$\begin{aligned}
& \implies \lambda + \gamma + \beta\left[\frac{-1}{(1)^2}\right]e^{\lambda\tau} = 0 \\
& \implies \lambda + \gamma - \beta e^{\lambda\tau} = 0 \\
& \implies \lambda = -\gamma - \beta e^{\lambda\tau}
\end{aligned}$$

There also exists a steady-solution at $x^* = \left(\frac{\beta-\gamma}{\gamma}\right)^{\frac{1}{n}}$:

$$\begin{aligned}
& \implies \lambda - (-\gamma) - (-\beta)\left[\frac{(n-1)(x^*)^n - 1}{(1+(x^*)^n)^2}\right]e^{\lambda\tau} = 0 \\
& \implies \lambda + \gamma + \beta\left[\frac{(n-1)\left(\frac{\beta-\gamma}{\gamma}\right) - 1}{\left(1+\left(\frac{\beta-\gamma}{\gamma}\right)\right)^2}\right]e^{\lambda\tau} = 0
\end{aligned}$$

Again the found Hopf values from `create_hopffunc` can be checked against this formula, with $\lambda = +\omega i$ like it was done for the single-delayed inverted pendulum example.

5.3 Neuron example

As seen in subsection 4.7 the linearised neuron DDE system takes the form:

$$\dot{u}(t) = A_0u(t) + A_1u(t-\tau_1) + A_2u(t-\tau_2) + A_3u(t-\tau_s)$$

Again the solution we take for u as $u(t) = ve^{\lambda t}$

$$\begin{aligned}
&\implies \lambda ve^{\lambda t} = A_0ve^{\lambda t} + A_1ve^{\lambda t}e^{-\lambda\tau_1} + A_2ve^{\lambda t}e^{-\lambda\tau_2} + A_3ve^{\lambda t}e^{-\lambda\tau_s} \\
&\implies [\lambda I - A_0 - A_1e^{-\lambda\tau_1} - A_2e^{-\lambda\tau_2} - A_3e^{-\lambda\tau_s}]v = 0 \\
&\implies det(\lambda I - A_0 - A_1e^{-\lambda\tau_1} - A_2e^{-\lambda\tau_2} - A_3e^{-\lambda\tau_s}) = 0
\end{aligned}$$

There exists a steady state at $(x_1^*, x_2^*) = (0, 0)$, and the partial derivative matrices become:

$$A_0 = \begin{bmatrix} -\kappa & 0 \\ 0 & -\kappa \end{bmatrix}, A_1 = \begin{bmatrix} 0 & 0 \\ a_{21} & 0 \end{bmatrix}, A_2 = \begin{bmatrix} 0 & a_{12} \\ 0 & 0 \end{bmatrix}, A_3 = \begin{bmatrix} \beta & 0 \\ 0 & \beta \end{bmatrix}$$

To find the values of the Hopf bifurcation, we start with the characteristic equation and work through the following:

$$\begin{aligned}
&det\left(\begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix} - \begin{bmatrix} -\kappa & 0 \\ 0 & -\kappa \end{bmatrix} - \begin{bmatrix} 0 & 0 \\ a_{21} & 0 \end{bmatrix} e^{-\lambda\tau_1} - \begin{bmatrix} 0 & a_{12} \\ 0 & 0 \end{bmatrix} e^{-\lambda\tau_2} - \begin{bmatrix} \beta & 0 \\ 0 & \beta \end{bmatrix} e^{-\lambda\tau_s}\right) = 0 \\
&\implies det\left(\begin{bmatrix} \lambda + \kappa - \beta e^{-\lambda\tau_s} & -a_{12}e^{-\lambda\tau_2} \\ -a_{21}e^{-\lambda\tau_1} & \lambda + \kappa - \beta e^{-\lambda\tau_s} \end{bmatrix}\right) = 0 \\
&\implies (\lambda + \kappa - \beta e^{-\lambda\tau_s})^2 - a_{21}a_{12}e^{-\lambda(\tau_1+\tau_2)} = 0 \\
&\implies \lambda^2 + 2\lambda\kappa - 2\lambda\beta e^{-\lambda\tau_s} - 2\kappa\beta e^{-\lambda\tau_s} + k^2 + \beta^2 e^{-2\lambda\tau_s} - a_{21}a_{12}e^{-\lambda(\tau_1+\tau_2)} = 0 \\
&\quad \text{Let } \lambda = +\omega i \text{ then } e^{-\lambda\tau} = e^{-\omega i\tau} = \cos(\omega\tau) - i\sin(\omega\tau) \\
&\implies (\omega i)^2 + 2\omega i\kappa - 2\omega i\beta[\cos(\omega\tau_s) - i\sin(\omega\tau_s)] - 2\kappa\beta[\cos(\omega\tau_s) - i\sin(\omega\tau_s)] + \kappa^2 + \\
&\quad \beta^2[\cos(2\omega\tau_s) - i\sin(2\omega\tau_s)] - a_{21}a_{12}[\cos(\omega(\tau_1+\tau_2)) - i\sin(\omega(\tau_1+\tau_2))] = 0 \\
&\implies \begin{cases} -\omega^2 - 2\omega\beta\sin(\omega\tau_s) - 2\kappa\beta\cos(\omega\tau_s) + \kappa^2 + \beta^2\cos(2\omega\tau_s) - a_{21}a_{12}\cos(\omega(\tau_1+\tau_2)) = 0 \\ 2\omega\kappa - 2\omega\beta\cos(\omega\tau_s) + 2\kappa\beta\sin(\omega\tau_s) - \beta^2\sin(2\omega\tau_s) + a_{21}a_{12}\sin(\omega(\tau_1+\tau_2)) = 0 \end{cases}
\end{aligned}$$

Using the above conditions, the values found for the Hopf bifurcation using `create_hopffunc` and `newton`, explained in Section 8.1.2, can be checked to see if they fulfil the above conditions. For a worked example of this please see Section 8.

6 Fold bifurcation

For a fold bifurcation to occur we need $\lambda = 0$ to fulfil Equation (6). Out of the 3 examples I studied in this project only the single-delayed inverted pendulum has an equilibrium that has $\lambda = 0$. For the single-delayed inverted pendulum example we take the solution to $u(t) = A_0u(t) + A_1u(t-\tau)$ to be in the form $u(t) = ve^{\lambda t}$. For $\lambda = 0$ we have:

$$\begin{aligned}
&[\lambda I - A_0 - A_1e^{-\lambda\tau}]v = 0 \\
&\implies [-A_0 - A_1]v = 0 \\
&\implies det(-A_0 - A_1) = 0 \\
&\implies det\left(-\begin{bmatrix} 0 & 1 \\ \cos(x^*) & 0 \end{bmatrix} - \begin{bmatrix} 0 & 0 \\ -a & -b \end{bmatrix}\right) = 0 \\
&\implies det\left(\begin{bmatrix} 0 & -1 \\ -\cos(x^*) + a & b \end{bmatrix}\right) = 0 \\
&\implies a - \cos(x^*) = 0
\end{aligned}$$

Looking at Figure 1, which show the stability of the equilibria of the demo in the Jupyter notebook '`Single_delayed_inverted_pendulum_example.ipynb`', it is seen that the stability changes (indicating a potential fold bifurcation) when $x^* = 0$. This stability change is indicated by the equilibrium becoming stable (blue) from unstable (red) at around the value $a = 1$. So for $\lambda = 0$ to exist, a must fulfil $a - \cos(0) = 1$. Hence, for any b value, $a = 1$ gives an eigenvalue $\lambda = 0$, which supports the plot in Figure 1. Like for Hopf bifurcation, I created a function `create_foldfunc` that finds an initial guess of the equilibrium and relevant fold values and also creates a function that can be use to find the fold bifurcation. When used in `newton` the outputs of the function `create_foldfunc` finds this $a = 1$ value.

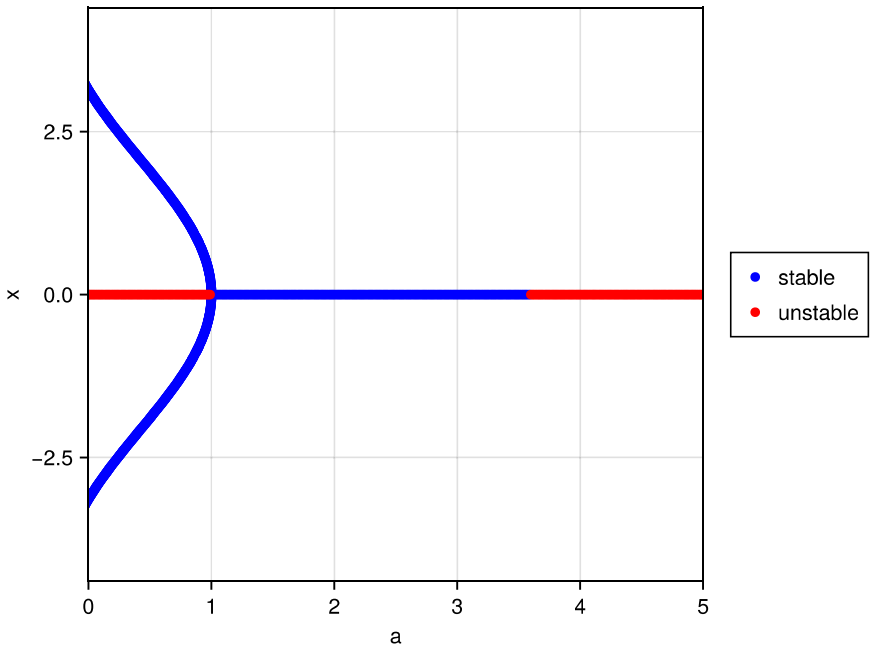


Figure 1: Plot to show the equilibria and stability for the single-delayed inverted pendulum example. The stable equilibrium are in blue while the unstable equilibria are in red. There are two stability changes at around $a = 1$ and $a \approx 3.5$, with a Hopf bifurcation at $a \approx 3.5$ and a pitchfork bifurcation at $a = 1$

While there exists $\lambda = 0$ the bifurcation for the single-delayed inverted pendulum example is actually a pitchfork bifurcation rather than a fold bifurcation due to the reflection symmetry of the system.

7 Continuation - tracking in a 2-parameter space

7.1 Hopf bifurcation continuation

To explain how to continue over the Hopf points in a 2-parameter space, I'll demonstrate with the single-delayed inverted pendulum example. For the single-delay inverted pendulum example, for parameters p and equilibrium x_{eq}, we have

$$f(x_{eq}, ..., x_{eq}, p) = 0.$$

To continue the Hopf bifurcation along the bifurcation curve we extend the above system to include Equation (13) and a normalisation condition to form the non-linear equation for Hopf

bifurcations:

$$G_H(y_H) = \begin{bmatrix} f(x_H, ..., x_H, p) \\ \Delta(x_H, p; i\omega_H)v_H \\ v_{ref}^T v_H - 1 \end{bmatrix}$$

for the unknowns $y_H = (x_H, v_H, \omega_H, p_i, p_j) \in \mathbb{R}^{3n+3}$ [KS23], where n is the dimension of the system, x_H is the state values for the Hopf, v_H is taken to be two real vectors (the real and imaginary parts of the eigenvector split up) and where p_i, p_j are the parameters of the space we wish to continue over. Note that $G_H(y_H) \in \mathbb{R}^{3n+2}$.

To find the Hopf bifurcation points in the (a, b) plane in Julia, I adjusted my Hopf function (`create_hopffunc`) so it would accept two varying parameters. My function outputs initial guesses for the values of y_H for the Hopf bifurcation and a function that accepts $3n+3$ variables and outputs $3n + 2$ values, my $G_H(y_H)$. Putting these into my `track_curve` function, I then tracked this function over the (a, b)-plane to get the green line in Figure 2 - this line shows the Hopf bifurcation values over the (a, b) space. Note that the purple line in Figure 2 are the pitchfork bifurcation points continued over the (a, b) plane.

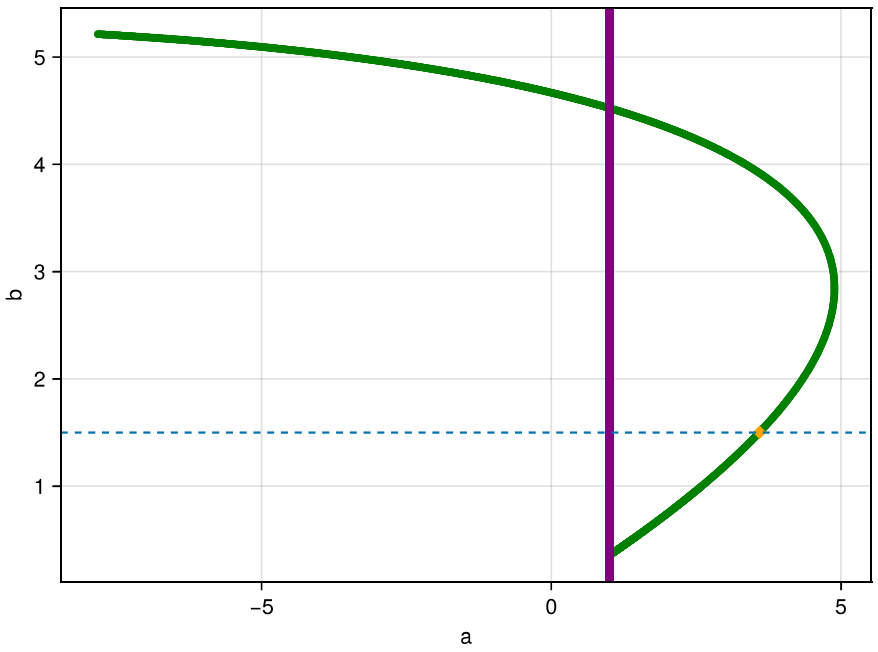


Figure 2: Plot to show the continuation of the bifurcation points in the (a, b) space. The Hopf bifurcation points are coloured green while the pitchfork bifurcation points are in purple. The found Hopf bifurcation in '`Single_delayed_inverted_pendulum_example.ipynb`' for fixed $b = 1.5$ is represented by the orange diamond

7.2 Fold bifurcation continuation

While none of the examples I explored have a fold bifurcation, a fold bifurcation can also be continued over a 2-parameter space. Assume there exists a DDE system

$$f(x_{eq}, ..., x_{eq}, p) = 0$$

for equilibrium point x_{eq} and parameters p. For the fold bifurcation the unknowns are $y_f = (x_f, v_f, p_i, p_j) \in \mathbb{R}^{2n+2}$, where n is the dimension of the system, x_f is the equilibrium that

the fold occurs at, v_f is the corresponding eigenvector and p_i, p_j are the two parameters the continuation is occurring over [KS23]. Like for the Hopf continuation, an extended system is needed. For the fold bifurcation, this is given as [KS23]:

$$G_f(y_f) = \begin{bmatrix} f(x_f, ..., x_f, p) \\ \Delta(x_f, p; 0)v_f \\ v_f^T v_f - 1 \end{bmatrix}$$

which is in $\mathbb{R}^{2n+1}$.

To continue the fold bifurcation points over the 2-parameter space in Julia, like for the Hopf bifurcation, I adjusted my fold function (`create_foldfunc`) to accept two varying parameters. The function outputs an initial guess of the fold values y_f and a function that accepts $2n+2$ variables and outputs $2n+1$ variables, my $G_f(y_f)$. To find all fold bifurcation points this is then input into my `track_curve` function with an appropriate initial tangent.

8 Worked example

To illustrate how my code works, I'll discuss the neuron example, which is the main demo of the DDE-Biftool package, alongside my code. For other examples (including a system with a single delay and a system with 1-dimension) please look at the following Jupyter notebook demos: '`Single_delayed_inverted_pendulum_example.ipynb`' and '`Mackey-Glass_equation_example.ipynb`'.

8.1 The functions

All the functions that are needed for DDE analysis are found in '`DDETools/shared`'. The functions that are shared in all examples are:

- `jacobian`: produces the Jacobian matrix for a system at a given point.
- `newton`: the Newton function that is used to find the true solution (especially the equilibrium).
- `track_curve`: tracks the curve of a function and in this example is used for finding the branches of the equilibria (or steady-states) and in the continuation of the bifurcation points.
- `f_deriv`: outputs the partial derivative matrix with respect to either state and/or parameters.
- `create_ststfunc`: outputs an initial guess for an equilibrium point and initialises a function to find the equilibria (when put into `track_curve`).
- `stab_func_matrix`: finds the stability (and eigenvalues and eigenvectors if asked) of an equilibrium point. It finds the stability (and eigenvalues, etc.) by approximating using the large matrix described in Equations (7) and (9). If the user asks, it will also output an initial guess for ω value for the Hopf bifurcation as well the real and imaginary parts of the first eigenvector of the eigenvalue $\pm\omega i$
- `j_diff`: finds the first-order differentiation matrix $D^{(1)}$ for some given interpolation points x_j
- `j_eval`: is used to find either $f(x_e), f'(x_e), f''(x_e), ...$ for some desired points x_e and a function f using (barycentric) interpolation. The derivative it outputs is chosen by the user.

- `stab_func_DDE`: finds the stability of an equilibrium point by using the Breda et al. 2009 method described in Section 4.3
- `create_hopffunc`: finds initial guess for Hopf parameter values and creates a function that can find the true Hopf values (when put into either `newton` or `track_curve`).
- `create_foldfunc` (used only in examples when fold bifurcations exist): finds initial guess for fold bifurcation parameter values and creates a function that can find the true fold bifurcation values (when put into either `newton` or `track_curve`).

8.1.1 `jacobian`

The function `jacobian` is similar to `f_deriv`, described in Section 8.1.4, however, it produces the matrix of the sum of $A_0, A_1, ..., A_{nd}$ rather than them individually. The arguments of this function are given below:

```
function jacobian(f,x; h=1e-5)
    ##inputs:
    #f is the RHS of the system
    #x is the point you want to find the Jacobian at
    #h is the stepsize for finite difference

    ##outputs: The Jacobian (matrix) for the system f at the point x0
```

8.1.2 `newton`

The function `newton` is used to find the root (the solution of $f(x) = 0$) of a function f. The function works in a way that it tries to find the point x at which the function it is fed equals 0. For `newton` to successfully converge to the point x is must fulfil the two conditions: that $|x - x_{pen}| <$ `tol`, where x_{pen} is the penultimate x, and that $|f(x)| <$ `tol`, where `tol` is defined below. The arguments of `newton` are given below:

```
function newton(f,x0; h=1e-5,df=x->jacobian_new(f,x;h=h),tol=1e-8,maxit=100)
    ##inputs:
    #f is the RHS of system
    #x0 is starting guess for x
    #h is the stepsize needed for jacobian function
    #tol is the tolerance for which the two convergence conditions (described above)
    must be fulfilled
    #maxit is the number of maximum iterations the function is allowed to do (if
    the function does not fulfil the tolerance conditions and exceeds maxit then
    the function did not converge succesfully onto a point x)

    ##outputs:
    #x is the value of x that fulfils newton (if converged)
    #converged tells user if their run was successful and x has converged to a point
    #J is the Jacobian at the previous point of the outputted x
```

The `newton` function is used for finding equilibria where $f(x^*, ..., x^*) = 0$ and in finding bifurcation such as Hopf and fold.

8.1.3 track_curve

The track_curve functions tracks a curve, for a varying parameter, given an initial starting point, an initial tangent and the (tracked) function's formula. It utilizes jacobian and newton to do so. The arguments of the track_curve are given below:

```
function track_curve(userf,y0,ytan; h=1e-5,userdf=x->jacobian_new(userf,x;h=h),
nmax=100,stepsize=0.01,tol=1e-8,maxit=6)
##inputs:
#userf is the function the user wishes to track (in a form such that it can be used
in track curve - y0 is one dimension more than original x0 so the tangent can be used)
#y0 is the initial point - it contains x0 (initial states) and a starting point for
the varying parameter
#ytan is the initial tangent
#h, tol and maxit are defined as in the newton function
#userdf is the jacobian function adapted for use in track_curve
#nmax is the number of points you want to track (tracking more points gives a more
complete visual of the tracked function)
#stepsize is the step size added to the tangent (ytan) when predicting the
new tracked point

##outputs:
#ylist are all the points that have been tracked
#ytan is the final tangent
```

The track_curve function is used for finding the branches of equilibria and is used in the continuation of bifurcation points in a 2-parameter space.

8.1.4 f_deriv

The function f_deriv is used to find the partial derivative matrices with respect to state - the matrices $A_0, A_1, .., A_{nd}$, where nd is the number of delays in the system - and/or with respect to parameters. The arguments for this function are given below:

```
function f_deriv(f,u,pars,nd;nx=[],np=[],v=[],h=1e-5,k=1e-5)
    ##inputs:
    #f is the system (equations)
    #u is the point the derivatives are being taken at
    #pars are the system parameters
    #nd is number of delays in the system
    #nx is the state derivative you want (empty=no state derivative wanted)
    #-> nx=1 is derivative wrt state at time t, nx=2 is derivative wrt state at
    t-tau_1, etc.
    #np is the index of the parameter derivative the user wants (empty = no
    parameter derivative wanted)
    #h is the small step for finite difference
    #k is the small steps for parameter (used only if nx and np both non-empty)

    ##outputs:
    #J is the matrix of partial derivatives (for chosen state and/or
    parameter derivative)
```

So to find the matrix A_0 you would set nx=1, for A_1 set nx=2, and so on for the remaining delays. This function is used in stab_func_matrix to determine the stability of equilibria.

8.1.5 create_ststfunc

The function `create_ststfunc` is used to obtain an initial guess of the equilibrium and creates a function that can either be used in `track_curve` to find the equilibria branch(es) or in `newton` to find an exact equilibrium point. Whether the function outputted by `create_ststfunc` is to be used in `track_curve` or `newton` is decided by the argument `par_indx`. If `par_indx` is left empty then the outputted function is to be used for `newton`, else if `par_indx=n` then the outputted function is to be used for `track_curve`, where the equilibria are found whilst varying the parameter with the index `n`. All the arguments of `create_ststfunc` are given below:

```
function create_ststfunc(f_DDE, u0, pars,nd; par_indx=[])#(f_DDE, f_tau,u0, pars,nd;
par_indx=[])
    ##inputs:
    #f_DDE is the RHS of the DDE system
    #u0 is the initial guess (for the states x_i's)
    #pars are parameters of the system (including the delays), ensure that p0 -
    the initial guess of the (varying) parameter - is in this vector
    #nd is the number of delays
    #par_indx is the index of the parameter the user wishes to vary
    ##Note: Leave par_indx empty to use the function output in newton function

    ##outputs:
    #y0 is a vector containing the initial guess of u0 and if par_indx is non empty also
    contains initial guess for the varying parameter (to be used in track_curve)
    #f_equilibrium is the function that is used to find equilibrium
```

8.1.6 stab_func_matrix

The function `stab_func_matrix` returns the stability and if the user wants it also returns the eigenvalues and eigenvectors of an equilibrium point. It finds these by looking at the eigenvalues of the large matrix in the Equations (7) and (9). If the equilibrium point is a Hopf bifurcation, the function can give the initial guesses of the ω value and the real and imaginary parts of the first eigenvalue of eigenvalue ωi if asked by the user. The arguments of `stab_func_matrix` are given below:

```
function stab_func_matrix(f_DDE,f_tau,x0::Vector,p0::Vector,pars,
par_indx::Vector,nd;doprint=1,
hopf=0,h=1e-6,m=100)
    ##inputs:
    #f_DDE is the DDE system
    #f_tau is the function for the delay(s)
    #x0 is the equilibirium point you're finding the stability of, it is to be
    given in vector form
    #p0 is the (varied) parameter value at which this equilibrium point occurs at,
    to be given in vector form
    #pars are the system parameters and include constant tau values
    #m in the number of steps you want to discretise over
    #pars_indx is the parameter that was varied when using track_curve function, to
    be given in vector form (even if only 1-dimensional)
    #nd is the number of delays
    #doprint=1 (the default) means that the eigenvalues and the lowest eigenvector is
```

```
    given, doprint=0 doesn't return these
    #hopf=0 is the default and the function doesn't return omega value, vrini
    or viini (see below)
    #hopf!=0 outputs the estimated information for a Hopf bifurcation if the user asks

    ##outputs:
    #stab=the stability of the equilibrium point given
    #eigvalsJ=the eigenvalues of the equilibrium point
    #eigvecs=the eigenvectors of the equilibrium point
    ##if hopf!=0 then the function can also output
    #vrini=initial guess of real part of first eigenvector of eigenvalue closest to
    being purely imaginary
    #viini=initial guess of imaginary part of first eigenvector of eigenvalue
    closest to being purely imaginary
    #omini=initial guess of omega (eigenvalue closest to being purely imaginary)
```

8.1.7 j_diff

The function `j_diff` returns the first-order differentiation matrix $D^{(1)}$ for the interpolation nodes x_j. It uses the formula for barycentric interpolation that was described in Section 4.3.1 of this report and in [BT04]. The arguments for this function are given below:

```
function j_diff(xbase)
    #input:
    #xbase are the interpolation points/nodes (x_j)

    #output:
    #D1 is the 1st-order differentiation matrix for the interpolation points
    (D_{ij}= l'j(x_i) for an interpolation point x_i)
```

The function `j_diff` is used in `j_eval`, which is introduced below.

8.1.8 j_eval

The function `j_eval` is used find the value of the n-th derivative of a function f at some desired points x_e for $n = 0, 1, ...$ etc. The value of n is given by the user. The function `j_eval` returns a matrix than when multiplied with the vector of f_j values it returns $f(x_e)$ to the n-th derivative. The arguments of this function are given below:

```
function j_eval(xj,xe;diff=0)
    ##inputs:
    #xj=the interpolation points/nodes
    #xe=the points you want to evaluate at
    #diff= what derivative you want to find points (xe) at (0=evaluation, 1= 1st
    derivative,2= 2nd derivative, etc.)

    ##output:
    #Dk is matrix such that when multiplied by f (the values f_j), the user gets the
    values of f (when diff=0), f' (when diff=1), etc. at the desired points, xe
```

The function `j_eval`, along with `j_diff`, is used in function `stab_func_DDE` to find the satbility of a point using the Breda et al. 2009 method.

8.1.9 stab_func_DDE

The function stab_func_DDE finds the stability of an equilibrium point by using the Breda et al. 2009 method, which looks at the eigenvalues of the spectral differentiation matrix $\mathcal{A}_N$ to determine stability. The arguments of the function are given below:

```
function stab_func_DDE(A,taus, N; eigvecs=0)
    ##inputs:
    #A is a vector of matrices, where A_0,...,A_d are the partial derivative matrices
    or the matrices in a linear DDE system
    #taus are the delays (in vector form)
    #N are the number of nodes for interpolation (so you get N+1 nodes)
    #eigvecs tells function whether or not to output the eigenvectors (0 means no
    eigenvectors outputted, any other value means they are outputted)

    ##outputs:
    #stab is the stability of the (equilibrium) point (already incorporated in the
    A matrices)
    #sm_eigvals are the eigenvalues of the point
    #if asked: sm_eigvecs are the eigenvectors (in matrix form) of the point - ith
    column of matrix contains the eigenvectors of the ith eigenvalue
```

It is interesting to note that the function stab_func_DDE is much quicker finding the stability of all equilibrium points over equilibria branches than stab_func_matrix, especially for large number of equilibria.

8.1.10 create_hopffunc

The function create_hopffunc is similar to create_ststfunc as it produces an initial guess and function, however, rather than finding the equilibria it is concerned with finding Hopf bifurcations. The initial guess is the guess for the Hopf bifurcation values and the function outputted is used for finding Hopf bifurcations when put into newton or track_curve. The arguments for create_hopffunc are:

```
function create_hopffunc(f_DDE,f_tau, pars, x0,p0::Vector,par_indx::Vector,nd;m=100)
    ##inputs:
    #f_DDE is the system function(s)
    #f_tau is the delay equation/function
    #pars are the parameters values of the system
    #x0 is the initial guess of the (equilibrium) state - the Hopf bifurcation
    #p0 is the initial parameter guess for the Hopf bifurcation (note this always a
    vector (even if only varying one parameter))
    #par_indx is the index of parameter(s) you're varying (again it should be given as
    a vector even if only of length 1)
    #nd is the number of delays
    #m is the number of discretised steps to be used stab_func_matrix; default is 100

    ##outputs:
    #y0 are the initial guesses of x, vr, vi, om, p
    #Note:x are the states, vr and vi are the real and imaginary parts of the (first)
    eigenvector for initial guess of eigenvalue om and where p is the initial guess of
    Hopf (varied) parameter p
```

```
    #fhopf is the function that finds the Hopf and is in a form that can be used in
    newton (for finding exact parameter values) or track_curve (to continue
    in 2-parameter plane)
```

8.1.11 `create_foldfunc`

The function `create_foldfunc` is again similar to `create_ststfunc` and `create_hopffunc` as it produces an initial guess and function but is instead concerned with finding fold bifurcations. It finds the values such that eigenvalue $\lambda = 0$. The initial guess is for the state, `x`, the fold happens at; the real, `vr`, and imaginary parts, `vi`, of the eigenvector for the eigenvalue closest to being purely imaginary and the value of the (varied) parameter, `p`, when the fold occurs. The function, `ffold`, outputted is used for finding the true fold bifurcation when put into `newton` or `track_curve`. The arguments for `create_foldfunc` are:

```
function create_foldfunc(f_DDE, f_tau,pars,x0,p0::Vector,par_indx::Vector,nd;m=100)
    ##inputs: are the same as create_hopffunc but for fold bifurcation rather than for
    a Hopf bifurcation

    ##outputs:
    #y0 is the initial guesses for x, vr,vi and p
    #ffold is the function that finds the fold bifurcation
```

This function can find the values for which $\lambda = 0$, however, it can't distinguish between a true fold bifurcation and other bifurcation with $\lambda = 0$ such as a pitchfork bifurcation.

8.2 Equilibria branch and stability

As discussed in Section 2.3, the interaction between two neurons can be modelled using the DDE system:

$$\begin{aligned}\dot{x_1}(t) &= -\kappa x_1(t) + \beta \tanh(x_1(t-\tau_s)) + a_{12}\tanh(x_2(t-\tau_2))\\ \dot{x_2}(t) &= -\kappa x_2(t) + \beta \tanh(x_2(t-\tau_s)) + a_{21}\tanh(x_1(t-\tau_1))\end{aligned}$$

For this example the parameters are $(\kappa, \beta, a_{12}, a_{21}, \tau_1, \tau_2, \tau_s)$ and the above system is given by the function `neuronfunc`, along with the constant delays given in `neurontau`. For this example, the parameter values taken are $\kappa = 0.5, \beta - 1, a_{12} = 1, a_{21} = 2.34, \tau_1 = 0.2, \tau_2 = 0.2, \tau_s = 1.5$.

To initiate finding the equilibria branches, the parameters and an initial equilibrium is given. Below finds an initial guess of an equilibrium point:

```
k=0.5
beta=-1
a12=1
a21=2.34
tau1=0.2
tau2=0.2
taus=1.5

nd=3 #there are 3 delays
y01,feq=create_ststfunc(neuronfunc,[0.0,0],[k,beta,a12,a21,tau1,tau2,taus],
nd, par_indx=4)
#varying paramter a21 so parameter index (par_indx) is 4
```

The output is a vector comprised of the equilibrium point (0,0) and a value for the varied parameter, a_{21}, that this equilibrium occurs at:

```
([0.0, 0.0, 2.34], var"#f_equilibrium#25"{Int64, typeof(neuronfunc), Vector{Float64},
Int64, Int64}(4, Main.neuronfunc, [0.5, -1.0, 1.0, 2.34, 0.2, 0.2, 1.5], 3, 2))
```

To check this x is in fact an equilibrium it is subbed back into the `neuronfunc` function to check the output of $f(x) = 0$

```
#check equilibrium (f(x*,...,x*)=0)
xvec=[fill(0.0,2) for _ in 1:nd+1] #creates a vector of vectors [[x*],[x*]]
neuronfunc(xvec,[k,beta,a12,a21,tau1,tau2,taus]) #should output [0,0]
```

The output:

```
2-element Vector{Float64}:
 0.0
 0.0
```

As discussed before, finding the stability of equilibria of DDEs is more complex than finding the stability of equilibria of ODEs due to the infinite dimensional past. One way the stability can found is by using a large matrix that can be used to create a finite-dimensional system that can approximate the stability of the equilibria. The function `stab_func_matrix` finds the stability using this matrix method, described in Section 4.2. The function's first output argument is `1` if the equilibrium point is stable and is `0` if it is unstable. Below we find the stability, using `stab_func_matrix`, of one equilibrium point (0,0) for the parameters given before:

```
#Test stability of initial point (it should be unstable)
x0=y01[1:2] #equilibrium x* given by create_ststfunc
p0=y01[3] #value of a21 that x* occurs at
stabi,eigvalsi,eigvecs1=stab_func(neuronfunc,neurontau,x0,[p0],[k,beta,a12,a21,tau1,
tau2,taus],[4],nd,doprint=1)
#stabi is the stability, eigvalsi are the eigenvalues of the equilibrium x0, eigvecs1 are
the eigenvectors of the equilibrium x0

unstabindx=findfirst(isequal(1),real(eigvalsi).>0) #finds the first eigenvalue whose
real part is greater than 0 (indicating instability)
unstabeigi=eigvalsi[unstabindx] #outputs the first eigenvalue that has real part
greater than 0

println("The stability is: $stabi")
println("The eigenvalues are: $eigvalsi")
println("The unstable eigenvalue $unstabeigi") #this values seems to match that given
in the neuron DDE-Biftool example
```

The output is:

```
The stability is: 0
The eigenvalues are: ComplexF64[-968.7660348034071 + 0.0im, -968.5375195436628
- 14.653396899307069im, -968.5375195436628 + 14.653396899307069im,
-967.8521945592739 - 29.292579296872145im, -967.8521945592739 + 29.292579296872145im, ...,
-0.0968461005550756 + 0.0im, 0.3069969848590207 + 0.0im]
The unstable eigenvalue 0.3069969848590207 + 0.0im
```

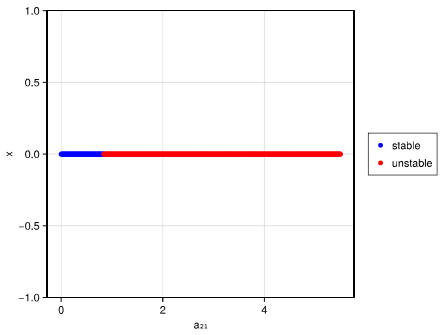
The stability indicator (first output of `stab_func_matrix`) is `0` so this equilibrium point is unstable. This is further supported by the fact that the rightmost eigenvalue $\lambda \approx 0.307$ has a real part greater than 0 - emphasizing that this equilibrium point is unstable.

We now find the equilibria branch whilst varying the parameter a_{21} and then find stability of these equilibrium points, using `stab_func_matrix`. The code below finds the equilibria branch and the stability of each equilibrium point.

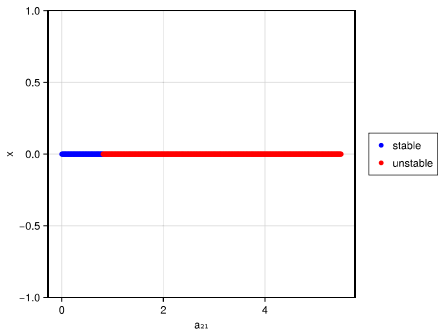
```
y02,feq2=create_ststfunc(neuronfunc,[0.0,0],[k,beta,a12,0,tau1,tau2,taus],
nd, par_indx=4) #we set a21 to 0 and finding starting point for equilibria branching
alist1,ytan1=track_curve_new(feq2,y02,[0.0,0,1],nmax=550)#tracks equilibria to give
equilibria branch as parameter a21 is varied

xmat1=hcat([u[1] for u in alist1],[u[2] for u in alist1]) #finds the values of x1 and
x2 for the equilibria (x_1*,x_2*)
xlist1=[xmat1[i,:] for i in 1:size(xmat1,1)] #creates a vector of vectors of all the
equilibria points


plist1=[u[3] for u in alist1] #A vector of the values of a21 for the equilibria points

m=100 #number of discretised steps
n=2 #number of states (x1,x2)
neq=length(plist1) #number of equilibrium points we're finding the stability of
stab=fill(NaN,neq)
eigvals1=[fill(0.0+0.0*im,n*(1+nd*m)) for _ in 1:neq]

for i in 1:neq
    stab[i],eigvals1[i],=stab_func_matrix(neuronfunc,neurontau,xlist1[i],[plist1[i]],
    [k,beta,a12,a21,tau1,tau2,taus],[4],nd,doprint=1,m=m) #outputs the stability (1 for
    stable, 0 for unstable) and also gives the eigenvalues for each point
end

stable=(stab.==1.0) #highlights indices of stable points (indicated by a 1 in
stable vector)
unstable=(stab.==0.0)#highlights indices of unstable points (indicated by a 1 in
unstable vector)

#Below plots the equilibria and shows their stability
fig1=Figure()
ax1=Axis(fig1[1,1],xlabel="a_{21}",ylabel="x")
scatter!(ax1,plist1[stable],[u[1] for u in xlist1[stable]],label="stable",color="blue")
scatter!(ax1,plist1[unstable],[u[1] for u in xlist1[unstable]],label="unstable",
color="red")
Legend(fig1[1,2], ax1, merge=true)
fig1
```

The figure produced by this code is shown in Figure 3a, with the stable equilibria points indicated in blue and the unstable equilibria points coloured red.

A second way to find stability of an equilibrium point is to use `stab_func_DDE` (which uses the Breda et al 2009 method). To use `stab_func_DDE` for this example, the partial derivative



(a) Stability method: Large matrix approximation



(b) Stability method: Breda et al. 2009 [BMV09]

Figure 3: Plot to show the equilibria branch for the neuron DDE system and the stability of the points. These points were found by tracking the system while the parameter a_{21} was varied. The stable equilibria are in blue while the unstable equilibria are in red

matrices A_0, A_1, A_2, A_3 need to be found for the equilibrium point(s). These matrices can be found either manually or by using `f_deriv`. In this example we'll use `f_deriv` so we can loop over all equilibrium points. The code for finding the stability of each equilibrium point is given below:

```
N=30 #so there are 31 interpolation points
neq=length(plist1) #number of equilibrium points we're finding the stability of
stab_DDE=fill(NaN,neq)

for i in 1:neq
    A0=DDETools.f_deriv(neuronfunc, xlist1[i], [k,beta,a12,plist1[i],tau1,tau2,taus],
    nd,nx=1) #finds partial derivative matrix with respect to x(t)
    A1=DDETools.f_deriv(neuronfunc, xlist1[i], [k,beta,a12,plist1[i],tau1,tau2,taus],
    nd,nx=2) #finds partial derivative matrix with respect to x(t-tau_1)
    A2=DDETools.f_deriv(neuronfunc, xlist1[i], [k,beta,a12,plist1[i],tau1,tau2,taus],
    nd,nx=3) #finds partial derivative matrix with respect to x(t-tau_2)
    A3=DDETools.f_deriv(neuronfunc, xlist1[i], [k,beta,a12,plist1[i],tau1,tau2,taus],
    nd,nx=4) #finds partial derivative matrix with respect to x(t-tau_s)
    stab_DDE[i],=DDETools.stab_func_DDE([A0,A1,A2,A3],[tau1,tau2,taus],N) #gives
    stability of each equilibrium point
end

stableDDE=(stab_DDE.==1.0) #highlights indices of stable points (indicated by a 1 in
stable vector)
unstableDDE=(stab_DDE.==0.0)#highlights indices of unstable points (indicated by a 1 in
unstable vector)

#Below plots the equilibria and shows their stability
fig2=Figure()
ax2=Axis(fig2[1,1],xlabel="a_21",ylabel="x")
scatter!(ax2,plist1[stableDDE],[u[1] for u in xlist1[stableDDE]],label="stable",
color="blue")
scatter!(ax2,plist1[unstableDDE],[u[1] for u in xlist1[unstableDDE]],label="unstable",
```

```
color="red")
Legend(fig2[1,2], ax2, merge=true)
fig2
```

The plot of equilibria and their stability is shown in Figure 3b, where again the stable equilibria are coloured blue and the unstable equilibria are coloured red. We can see by comparing both Figures 3a and 3b that the two stability functions return very similar stabilities for the equilibria.

8.3 Hopf bifurcation

The value of the rightmost (largest real part) eigenvalue for each equilibrium can be plotted, see Figure 4, to look for the occurrence of a Hopf bifurcation. As mentioned in Section 5, the Hopf bifurcation occurs when a pair of complex conjugate eigenvalues cross the imaginary axis and becomes purely imaginary. This condition is represented by the blue dashed line in Figure 4 at the value $\mathbb{R}(\lambda) = 0$. The code for finding the rightmost eigenvalues of each equilibrium point is given below:

```
avals=[u[3] for u in alist1] #list of a21  values
realeigs=real(eigvals1) #gets real parts of all eigenvalues for each tracked point
n_eq=length(avals) #number of equilibrium points

maxrealeigs=fill(NaN,n_eq) #blank array for largest real part of an eigenvalue for
each equilibrium point

for i in 1:n_eq
    realindx=argmax(realeigs[i]) #finds index of largest real part of eigenvalues
    for a21 value
    maxrealeigs[i]=realeigs[i][realindx] #the largest real part of eigenvalues
    for the a21 value (needed as largest real part dictates if a point is stable (<0)
    or unstable (>0))
end
```

Looking at Figure 4, we see that the rightmost eigenvalues do cross the imaginary axis and become purely imaginary for roughly the a_{21} value of 0.8. To find the true a_{21} parameter value for the Hopf bifurcation, a more accurate initial guess is found. Note that while the initial guess of a_{21} for a Hopf bifurcation can be found using both `stab_func_matrix` and `stab_func_DDE`, the true Hopf bifurcation can only be found when using the stability function `stab_func_matrix` as `create_hopffunc`, currently, only works for `stab_func_matrix`. The code for finding the initial guesses of a_{21} for the two different stability methods is given below:

```
#We aim to find where the eigenvalues cross the imaginary axis (and become
purely imaginary)
#Finds initial a21 value for Hopf bifurcation from stab_func_matrix results
unstab_indx=findfirst(isequal(0.0),stab) #find index of stability change
a21unstab=avals[unstab_indx] #initial guess of a21 value for Hopf bifurcation
```

Giving an initial guess of a_{21}:

```
0.8400000000000005
```

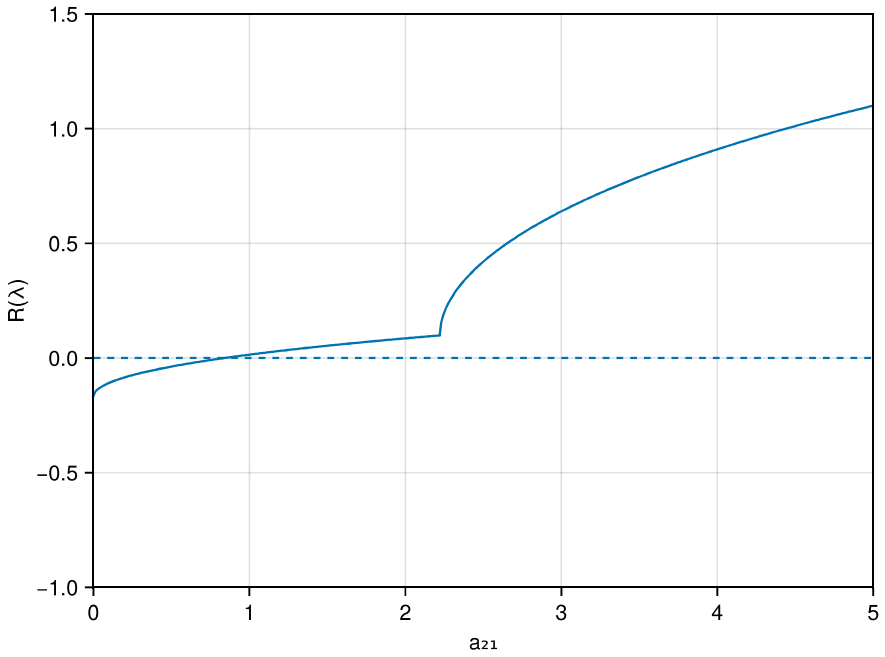


Figure 4: The real part of the rightmost eigenvalue for the different a_{21} parameters. The imaginary axis is indicated by the blue dashed line at $\mathbb{R}(\lambda) = 0$

```
#Finds initial a21 value for Hopf bifurcation from stab_func_DDE results
unstab_DDEindx=findfirst(isequal(0.0),stab_DDE) #finds index of Hopf bifurcation (for
stability vector given by stab_func_DDE)
a21unstabDDE=plist1[unstab_DDEindx] #finds initial guess for parameter a21 value for
a Hopf bifurcation
```

Giving an initial guess of $a21$:

```
0.8100000000000005
```

Looking at the Figures 3a and 3b, and the value of a_{21} where the stability appears to change, we see that the initial guesses are pretty good. Despite this, the true a_{21} value for the Hopf bifurcation needs to be found. To do this the function `create_hopffunc` is used. The initial guess of the value of a_{21} (taking the one given by `stab_func_matrix`), and its associated equilibrium point $(0, 0)$ are put into the `create_hopffunc` to aid its outputted initial guess. This result is then put into `newton` to find the true a_{21} value for the Hopf bifurcation. The commands are given below:

```
yh01,fhopf=DDETools.create_hopffunc(neuronfunc,neurontau,[k,beta,a12,0,tau1,tau2,taus],
xlist1[unstab_indx],[a21unstab],[4],nd) #outputs initial guess of the Hopf bifurcation
(including x, omega and a21 estimates) and a function that contains the method/formula
to find the Hopf bifurcation
println("The initial guess for Hopf info: $yh01")
yh1,conv1,=DDETools.newton(fhopf,yh01) #putting initial guess and Hopf function into
newton we get the true parameter values and information for the Hopf bifurcation
println("The true information for the Hopf bifurcation: $yh1")
println("converged?: $conv1")
a_hopf=yh1[end]
```

```
om_hopf=yh1[end-1]
x_hopf=yh1[1:2]
println("The omega value for Hopf bifurcation: $om_hopf")
println("The a21 value for Hopf bifurcation: $a_hopf")
```

The output for this is:

```
The initial guess for Hopf info: [0.0, 0.0, -0.7372097807744701, -0.6756639246921932,
0.0, 3.526043721116034e-14, 0.7668092696644265, 0.8400000000000005]
The true information for the Hopf bifurcation: [0.0, 0.0, -0.7438855391134424,
-0.6683070437290806, 1.7528156689931815e-14, 1.5751721548947158e-14, 0.7819651620459527,
0.8071232251830753]
converged?: true
The omega value for Hopf bifurcation: 0.7819651620459527
The a21 value for Hopf bifurcation: 0.8071232251830753
```

To compare the two stability methods (and their functions), the following is written:

```
println("The estimated a21 value for Hopf bifurcation using large matrix stability
(stab_func_matrix) is: $a21unstab")
println("The estimated a21 value for Hopf bifurcation using stab_func_DDE is:
$a21unstabDDE")
println("True a21 value for Hopf bifurcation given by stab_func_matrix: $a_hopf")
```

which outputs:

```
The estimated a21 value for Hopf bifurcation using large matrix stability
(stab_func_matrix) is: 0.8400000000000005
The estimated a21 value for Hopf bifurcation using stab_func_DDE is: 0.8100000000000005
True a21 value for Hopf bifurcation given by stab_func_matrix: 0.8071232251830753
```

From looking at the above output, we can see that `stab_func_DDE` is more accurate than `stab_func_matrix` as the initial guess of the Hopf a_{21} value obtained from `stab_func_DDE` (≈ 0.81) is closer to the true a_{21} Hopf value (≈ 0.807) than the initial guess of a_{21} when using `stab_func_matrix` (≈ 0.84). The true value of a_{21} for a Hopf bifurcation, represented by a black diamond, is shown on Figure 5.

We can test the above Hopf values by checking that the following hold (which have been obtained by manipulating from the characteristic equation with $\lambda = +\omega i$, as seen in Section 5.3):

$$-\omega^2 - 2\omega\beta\sin(\omega\tau_s) - 2\kappa\beta\cos(\omega\tau_s) + \kappa^2 + \beta^2\cos(2\omega\tau_s) - a_{21}a_{12}\cos(\omega(\tau_1+\tau_2)) = 0$$
$$2\omega\kappa - 2\omega\beta\cos(\omega\tau_s) + 2\kappa\beta\sin(\omega\tau_s) - \beta^2\sin(2\omega\tau_s) + a_{21}a_{12}\sin(\omega(\tau_1+\tau_2)) = 0$$

Below is my code for testing my found a_{21} and ω Hopf values (`a_hopf` and `om_hopf`):

```
eq3ans=-(om_hopf)^2 -2*om_hopf*beta*sin(om_hopf*taus) - 2*k*beta*cos(om_hopf*taus)+
k^2+(beta^2)*cos(2*om_hopf*taus) -a_hopf*a12*cos(om_hopf*(tau1+tau2))

eq4ans=2*om_hopf*k - 2*om_hopf*beta*cos(om_hopf*taus)+2*k*beta*sin(om_hopf*taus)-
(beta^2)*sin(2*om_hopf*taus)+a_hopf*a12*sin(om_hopf*(tau1+tau2))
```

The ouputs for these are:

```
-3.687616878522704e-11

-4.159839139816768e-11
```

These are both reasonably close to 0 so the functions `create_hopffunc` and `newton` are working well together to find the true Hopf bifurcation values.

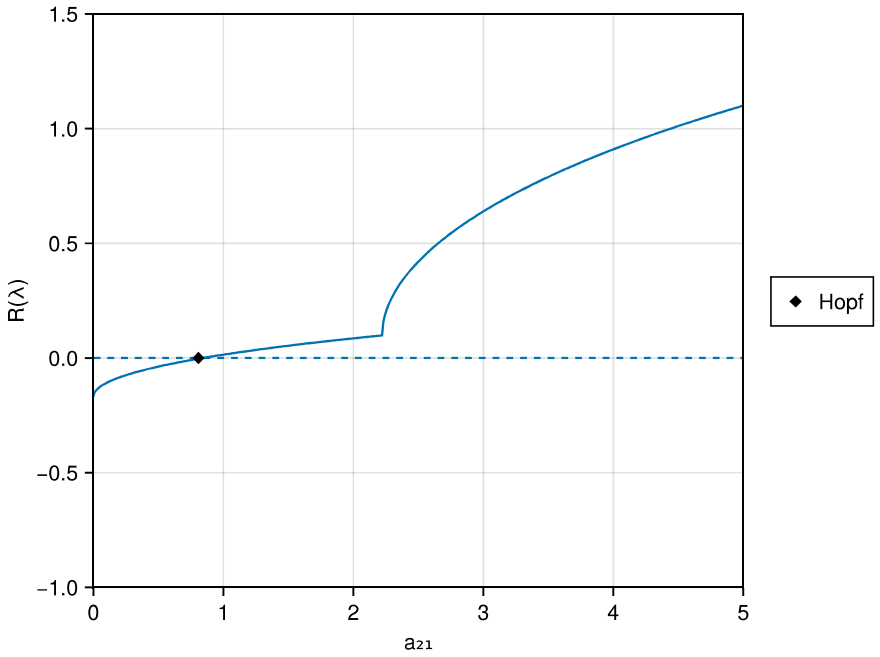


Figure 5: The real part of the rightmost eigenvalue for the different a_{21} parameters and the true a_{21} value for the Hopf bifurcation (represented by a black diamond)

8.4 Continuation in (a_{21}, τ_s)-plane

To finish this example, the Hopf bifurcation points over the (a_{21}, τ_s) parameter space are found. Below is the code that finds them and plots them, which are shown in Figure 6.

```
yheq01,fhopfeq1=DDETools.create_hopffunc(neuronfunc,neurontau,[k,beta,a12,a_hopf,tau1,
tau2,taus],x_hopf,[a_hopf,taus],[4,7],nd) #initialises Hopf function and initial guess
for hopf information with a21 approx. 0.81 and tau_s=1.5
yheqlist1,heqtan1=DDETools.track_curve(fhopfeq1,yheq01,[0.0,0,0,0,0,0,0,1,-1],nmax=1250)
#tracks a21 and tau_s over (a21, tau_s) plane

#Plot of continuation/tracking in (a21,tau_s) plane
fig6=Figure()
ax6=Axis(fig6[1,1],xlabel="a21",ylabel="tau_s")
scatter!(ax6,[u[end-1] for u in yheqlist1],[u[end] for u in yheqlist1],color="green",
markersize=:7)
ylims!(ax6,[1,11])
xlims!(ax6,[-0.5,2.5])
ax6.xticks=-0.5:0.5:2.5 #creates x-axis increments that match those given in DDE-Biftool
(makes comparsion between the two easier)
ax6.yticks=1:1:11 #creates y-axis increments that match those given in DDE-Biftool
fig6
```

9 Conclusion

Delay differential equations (DDEs) are becoming more readily used to model dynamical systems, especially those that include delays [BMV14a]. Analysis of these systems, particularly

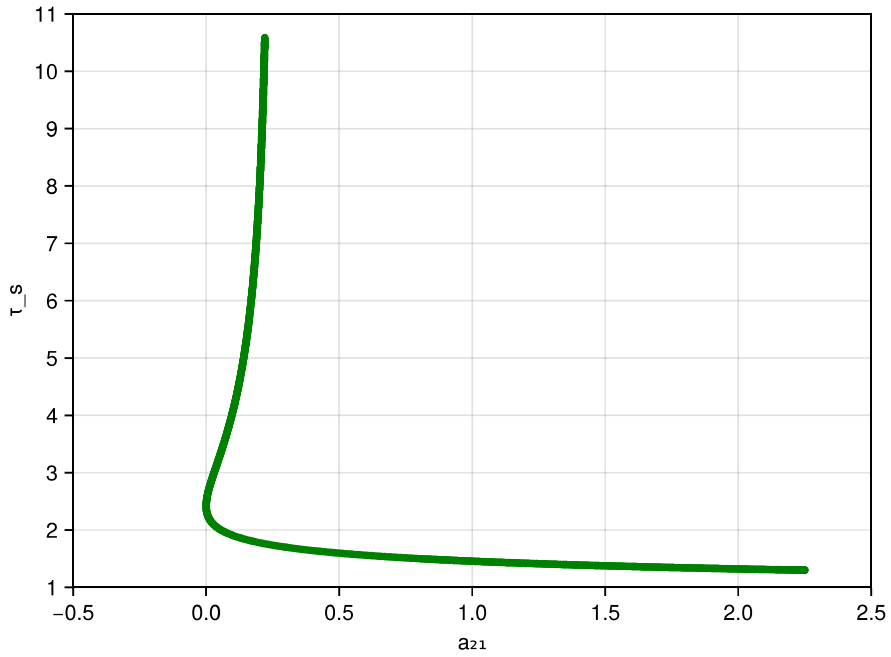


Figure 6: Plot to show the Hopf bifurcation points continued over the (a_{21}, τ_s) space

their stability, can be complicated and complex due the infinite-dimensional property [Smi11a] of the system's dependence on its history. However, there are ways to approximate the stability of DDE systems by transforming the DDE system into a more manageable form (via the Linearisation Principle) and writing the characteristic equation in a finite-dimensional form. During this project, I developed functions in Julia for common tools used in analysing DDE systems, focusing on those with constant delays, having found DDE code in Julia not as readily used as in MATLAB, with its DDE-Biftool package. Having researched the background theory of delay differential equations, I created a function that could be used to find the equilibria branches of DDE systems and looked into two different methods of determining the stability of these equilibria points. I looked at approximating the stability using a large matrix equation and using a method put forward by Breda et al in their 2009 paper [BMV09] before writing them as functions to be implemented in Julia. I found that while both were good methods, the [BMV09] method was more accurate and computationally quicker.

Alongside this I also developed functions to aid finding Hopf and fold bifurcations, both for a state-parameter space and for continuation of the bifurcation points over a 2-parameter space. To test all my functions and their accuracy I explored three examples of DDE systems with varying dimensions and number of delays and compared my results to published findings and theoretical formula. My functions build a good foundation of DDE tools that I believe could be built on to develop further tools, that can be used in Julia, such as a function for finding periodic orbits.

References

[BMV09] D. Breda, S. Maset, and R. Vermiglio. *TRACE-DDE: a Tool for Robust Analysis and Characteristic Equations for Delay Differential Equations*, volume 388, pages 145–155. 08 2009.

[BMV14a] D. Breda, S. Maset, and R. Vermiglio. *Stability of Linear Delay Differential Equations: A Numerical Approach with MATLAB*, chapter Preface, pages vii–viii. Springer, 2014.

[BMV14b] D. Breda, S. Maset, and R. Vermiglio. *Stability of Linear Delay Differential Equations: A Numerical Approach with MATLAB*, chapter 3: Stability of Linear Autonomous Equations, pages 23–33. Springer, 2014.

[BT04] J.-P. Berrut and L.N. Trefethen. Barycentric lagrange interpolation. *SIAM Review*, 46(3):501–517, 2004.

[GB24] D. Gupta and S. Bhalekar. Analysis of stability, bifurcation and chaos in generalized mackey-glass equations. *arXiv:2411.02865*, 2024.

[KS04] B. Krauskopf and J. Sieber. Bifurcation analysis of an inverted pendulum with delayed feedback control near a triple-zero eigenvalue singularity. 2004.

[KS23] B. Krauskopf and J. Sieber. Bifurcation analysis of systems with delays: Methods and their use in applications. In D. Breda, editor, *Controlling Delayed Dynamics: Advances in Theory, Methods and Applications*, pages 195–245. Springer International Publishing, Cham, 2023.

[MR14] S. Manjunath and G. Raina. Stability and hopf bifurcation analysis of the mackey-glass and lasota equations. *The 26th Chinese Control and Decision Conference (2014 CCDC)*, pages 2076–2082, 2014.

[Smi11a] H. Smith. *An Introduction to Delay Differential Equations with Applications to the Life Sciences*, chapter 4: Linear Systems and Linearization, pages 41–60. Springer, 2011.

[Smi11b] H. Smith. *An Introduction to Delay Differential Equations with Applications to the Life Sciences*, chapter 3: Existence of Solutions, pages 25–40. Springer, 2011.

[Smi11c] H. Smith. *An Introduction to Delay Differential Equations with Applications to the Life Sciences*, chapter 1: Introduction, pages 1–12. Springer, 2011.