


# FYS-STK4155: APPLIED DATA ANALYSIS AND MACHINE LEARNING

## The diffusion equation & finding the eigenvalues of a symmetric matrix

Mohamad Mahmoud & Aleksandar Davidov

 [GitHub - click here -](#)

December 23, 2021

### Abstract

A neural network approach solving ordinary differential equations (ODEs) and partial differential equations (PDEs). We contrasted the results produced by the neural networks to analytical answers (when available) and to the explicit scheme as a numerical approach serving as a benchmark for accuracy and reliability. For the PDE case we studied the diffusion equation and how a neural network can be set up to solve it with the help of a trial function. The neural network constructed for this part yielded subpar results never fairing against the solution provided by the explicit scheme. As for the ODE case we constructed a neural network using Tensorflow set out to find the eigenvectors of a real symmetric matrix corresponding to their maximum and minimum eigenvalues. Our testing methodology has been found to be a reliable approach. Even though the results for the diffusion equation were poor. Through our analysis we manage to isolate the combination of number of hidden layers and number of hidden nodes that always resulted in reducing the error.

## Contents

|            |   |           |
|------------|---|-----------|
| <b>I</b>   | <b>Introduction</b>                         | <b>2</b>  |
| <b>II</b>  | <b>Theory</b>                               | <b>2</b>  |
| i          | The Diffusion Equation . . . . .            | 2         |
|            | In One Dimension . . . . .                  | 2         |
| ii         | The Neural Network approach . . . . .       | 3         |
| iii        | Eigenvalues of a Symmetric Matrix . . . . . | 4         |
|            | Euler's Method . . . . .                    | 5         |
| <b>III</b> | <b>Result</b>                               | <b>6</b>  |
| i          | The Diffusion Equation . . . . .            | 6         |
|            | Number of Hidden Nodes and Layers . . . . . | 6         |
| ii         | Eigenvalues of a Symmetric Matrix . . . . . | 8         |
|            | Number of Hidden Nodes and Layers . . . . . | 9         |
|            | First Test Case . . . . .                   | 11        |
|            | Second Test Case . . . . .                  | 12        |
| <b>IV</b>  | <b>Conclusion</b>                           | <b>13</b> |
| i          | The Diffusion Equation . . . . .            | 13        |
| ii         | Eigenvalues of a Symmetric Matrix . . . . . | 13        |
|            | <b>References</b>                           | <b>15</b> |
| <b>V</b>   | <b>Appendix</b>                             | <b>16</b> |
| i          | The Diffusion Equation . . . . .            | 16        |
|            | Analytical Solution . . . . .               | 16        |
|            | Explicit Scheme . . . . .                   | 16        |
|            | Trial Function . . . . .                    | 17        |
| ii         | Eigenvalues of a Symmetric Matrix . . . . . | 17        |
|            | Euler's Method . . . . .                    | 17        |

|     |   |    |
|-----|---|----|
|     | Number of Hidden Nodes and Layers . . . . . | 18 |
| iii | Figures & Tables . . . . .                  | 18 |
|     | The Diffusion Equation . . . . .            | 18 |
|     | Eigenvalues of a Symmetric Matrix . . . . . | 19 |
|     | First Test Case . . . . .                   | 20 |
|     | Second Test Case . . . . .                  | 21 |

## I. INTRODUCTION

PDEs or partial differential equations are powerful tools and they make up the building blocks of many scientific fields with their many applications. Used to determine the characteristics and behaviors of a certain physical problem such as sound waves, electromagnetic waves, heat or molecule diffusion. They make such an essential part of many sciences that often they get their own distinct staple names. The Schrödinger equation that governs the wave function of a quantum-mechanical system, Navier–Stokes equations to describe the motion of viscous fluid substances and Maxwell’s equations that describe the behavior of electric and magnetic fields etc.

Numerically any ODE or PDE for that matter is usually solved either through finite difference or finite element. We will be testing a neural network’s capabilities in producing reliable and accurate results as a PDE-/ and ODE-/ solver and contrast it to that of the explicit scheme [2]; a finite difference approach. Some PDEs have explicit analytical solution but for most cases a numerical approach has to be taken. For the diffusion/heat equation part of our report we will be focusing on the numerical side but we chose an analytically solvable case in order to verify our results. We will also dive into the work of Yi et al. in [5] on how an ODE can be used to determine the eigenvectors of a real symmetric  $6 \times 6$  matrix and from that calculate its corresponding eigenvalues using Rayleigh quotient. We also contrast the results to that of the numerical solution of the explicit scheme and the eigenvectors and eigenvalues obtained by NumPy’s `numpy.linalg.eig`.

One of the main reasons to consider solving differential equations using an NN (neural network), is that an NN produces a viable function that is the result of a fine-tuned trial function. The NN through it’s structure adjusts its parameters and learns with each iteration how to make alterations in order to arrive to an adequate solution. The final product is a trained function, given an input it can produce promising results. Unlike the discretized schemes that only fit for these given parameters at the start of the evaluation. Although one must not be overly optimistic, the solution from the neural network is best fit for similar applications.

## II. THEORY

### i. The Diffusion Equation

The diffusion equation describes the collective motion/variation in a confined space as a function of time. It could be used to describe particle density/motion, energy

density, temperature change and other processes as long as interactions between materials take place. Usually it’s written as:

$$\frac{\partial u(\mathbf{r}, t)}{\partial t} = \nabla \cdot [D(u, \mathbf{r}) \nabla u(\mathbf{r}, t)] \quad (1)$$

Where  $D(u, \mathbf{r})$  is the diffusion coefficient at a position vector  $\mathbf{r} = (x_1, x_2, x_3, \dots)$ ,  $x_i \in \mathbb{R}$  and  $u(\mathbf{r}, t)$  is the function we’re interested in solving. In the case where the diffusion coefficient is a constant eq. 1 collapses to:

$$\frac{\partial u(\mathbf{r}, t)}{\partial t} = D \nabla^2 u(\mathbf{r}, t) \quad (2)$$

Since we’re looking for a general numerical solution for the diffusion equation, some scaling could be done making the equation dimensionless. The scaling is done in *ch. 10.2* in [1], which rids us of the constant  $D$  entirely, so the dimensionless diffusion equation is given by

$$\frac{\partial u(\mathbf{r}, t)}{\partial t} = \nabla^2 u(\mathbf{r}, t) \quad (3)$$

Although the notation hasn’t changed the variables are now dimensionless. We are going to shift our focus to the one dimensional case from this point onward. Our main goal is to first solve the one dimensional diffusion equation using an explicit scheme also know as Forward Euler, then contrast the results to that of a neural network approach.

### In One Dimension

The problem we will be focusing on throughout our numerical analysis and testing is given by the following

$$\frac{\partial u(x, t)}{\partial t} = \frac{\partial^2 u(x, t)}{\partial x^2}, \quad t > 0, x \in [0, L] \quad (4)$$

with the initial condition:

$$u(x, 0) = \sin(\pi x) \quad 0 < x < L$$

where  $L = 1$  and the boundary conditions are set to

$$u(0, t) = 0 \quad t \geq 0$$

and

$$u(L, t) = 0 \quad t \geq 0$$

With the analytical solution

$$u(x, t) = \sin(\pi x) \exp(-\pi^2 t)$$

A detailed calculation of the analytical solution can be found in **V Appendix i**. We start with the diffusion equation itself

$$\frac{\partial u(x, t)}{\partial t} = \frac{\partial^2 u(x, t)}{\partial x^2} \quad (5)$$

or in a more compact notation

$$u_t = u_{xx}$$

In order to solve eq. 5 numerically we would need to approximate the derivatives at hand. We start by discretizing the space  $[0, L]$  and time  $[0, T]$  domain by replacing both with a set of uniformly spaced mesh points. Thus any point  $x$  inside the spatial domain is denoted as

$$x_i = i\Delta x, \quad i = 0, \dots, N_x$$

as for the time domain

$$t_n = n\Delta t, \quad n = 0, \dots, N_t$$

Where for uniform meshes  $\Delta x = L/(N_x + 1)$  and  $\Delta t = T/(N_t + 1)$ .

Now we turn to finite difference approximations in order to set up the numerical scheme. A straightforward and simple approach computationally, is obtained by applying a forward difference in time and a central difference in space [2]:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} \quad (6)$$

The PDE in eq. 5 is now expressed as a set of algebraic equations, often referred to as discrete equations. Hence why we call it the discrete form of the diffusion equation. A walkthrough to the notation in eq. 6 can be found in **V Appendix i**.

The assumption here is that  $u^n$  is known throughout the spatial domain and thus we aim to solve for  $u_i^{n+1}$ , this gives us

$$u_i^{n+1} = u_i^n + F(u_{i+1}^n - 2u_i^n + u_{i-1}^n) \quad (7)$$

where  $F = \frac{\Delta t}{\Delta x^2}$  is the Fourier number, a *dimensionless quantity*. Usually expressed with the diffusion coefficient  $D$  as follows  $F = D \frac{\Delta t}{\Delta x^2}$ , but because of the scaling of the equation  $D$  was omitted. The explicit scheme requires  $F \leq 1/2$  in order to produce meaningful results [2].

The initial condition in our scheme coincides with  $t_0 = 0$

$$u(x_i, t_0) = \sin(\psi x_i) = I(x_i)$$

and at the boundary conditions  $x_0 = 0$  and  $x_{N_x} = N_x \Delta x = L$  we have

$$u(x_0, t_n) = 0 \quad u(x_{N_x}, t_n) = 0$$

---

#### Setting up the algorithm

---

- opt.** set the boundaries (if needed) for  $u_i^0 = 0$  at  $i = 0$  and  $i = N_x$
  - 1. compute the initial condition:  $u_i^0 = I(x_i)$  for  $i = 1, 2, \dots, N_x - 1$
  - 2. for  $n = 1, 2, \dots, N_t - 1$ 
    - I. apply eq. 7 for all the internal spatial points,  $i = 1, 2, \dots, N_x - 1$
    - II. set the boundaries (if needed) for  $u_i^{n+1} = 0$  at  $i = 0$  and  $i = N_x$
- 

## ii. The Neural Network approach

*The basis of our theory and code are derived from Kristin Baluka Hein's lectures notes, [3].*

A general partial differential equation can be expressed as

$$f\left(\mathbf{x}, g(\mathbf{x}), \nabla_{x_1} g(\mathbf{x}), \dots, \nabla_{x_N} g(\mathbf{x}), \nabla_{x_1}^2 g(\mathbf{x}), \dots, \nabla_{x_N}^2 g(\mathbf{x}), \dots, \nabla_{x_1}^n g(\mathbf{x}), \dots, \nabla_{x_N}^n g(\mathbf{x})\right) \quad (8)$$

Where  $g(\mathbf{x})$  is the function (that we aim to find) of a vector  $\mathbf{x}$  containing all  $N$  variables,  $\mathbf{x} = (x_1, x_2, \dots, x_N)$ . The notation  $\nabla_{x_i}$  stands for the first order partial derivative wrt.  $x_i$ ,  $\nabla_{x_i} = \frac{\partial}{\partial x_i}$ . Subsequently,  $\nabla_{x_i}^n$  is the  $n$ -th order partial derivative wrt.  $x_i$ ,  $\nabla_{x_i}^n = \frac{\partial^n}{\partial x_i^n}$ . The variables denoted in the function  $f$  is just a way to write that the expression involves  $\mathbf{x}$ , a function  $g(\mathbf{x})$  and its partial derivatives for each point in  $\mathbf{x}$  up to the  $n$ -th order derivative.

We are trying to solve the diffusion equation in one dimensional space as shown in eq. 4. This means that the vector  $\mathbf{x}$  packs one spatial variable and one time variable,  $\mathbf{x} = (x, t)$ , and that the function we aim to find is  $g(\mathbf{x}) = u(x, t)$ . We start by discussing the general form for how the trial function  $g_t(\mathbf{x})$  is setup in order to be a viable solution for  $g(\mathbf{x})$  and the initial and boundary conditions of the PDE.

$$g_t(\mathbf{x}, P) = h_1(\mathbf{x}) + h_2(\mathbf{x}, N(\mathbf{x}, P)) \quad (9)$$

Here  $h_1(\mathbf{x})$  makes it so  $g_t(\mathbf{x})$  satisfies a set of given conditions,  $N(\mathbf{x}, P)$  a neural network with inputs  $\mathbf{x}$  and  $P$ ; where the parameter  $P$  contains all the weights and biases of the layers that makes up the structure of the neural network. The role of  $h_2(\mathbf{x}, N(\mathbf{x}, P))$  is to ensure that the output from  $N(\mathbf{x}, P)$  is set to zero, when the trial function  $g_t(\mathbf{x})$  is evaluated at a point  $\mathbf{x}$  where the conditions for the PDE must be satisfied.

The NN(neural network) optimizes  $P$  for each iteration using gradient descent and the mean squared error as the loss function so that it fits  $g_t(\mathbf{x})$  as close as possible to the actual function we are trying to solve for,  $g(\mathbf{x})$ . The optimization of the parameters is based on the fact that the lhs. and rhs. in eq. 5 are equal to each other and thus ideally the cost function should be equal to zero. This ultimately means

$$u_t(\mathbf{x}, P) - u_{xx}(\mathbf{x}, P) = 0$$

and that the cost function is defined as

$$\mathcal{C}(\mathbf{x}, P) = (u_t(\mathbf{x}, P) - u_{xx}(\mathbf{x}, P))^2$$

but since the input  $\mathbf{x}$  contains a cumulation of points,

$$\mathcal{C}(\mathbf{x}, P) = \frac{1}{N} \sum_{i=1}^N (u_t(\mathbf{x}, P) - u_{xx}(\mathbf{x}, P))^2$$

Where we have used  $u$  to denote  $g_t$  for the sake of brevity and clarity for the reader. Here  $N$  is number of elements in the input vector and in our case  $N_{input} = 2$ , since  $\mathbf{x} = (x_i, t_n)$  for a given point in the mesh.

The neural network takes in  $\mathbf{x}$  as an input and transfers it to the hidden layer without doing any alterations, in other words there is no weights and bias matrix for the input layer. As mentioned  $P$  contains the weights and biases of the layers making up the NN, so for an arbitrary number  $N_l$  of hidden layers plus an output layer,  $P = \{P_{H1}, P_{H2}, \dots, P_{HN_l}, P_{output}\}$ . Where  $P_{H1}$  denotes the weights and bias matrix for the first hidden layer,  $P_{H2}$  for the second hidden layer and so on.  $P_{H1}$  is an  $N_{H1} \times (N_{input} + 1)$  matrix, where  $N_{H1}$  is the number of neurons in the hidden layer  $H1$  and  $N_{input}$  is number of neurons in the input layer, +1 to account for the bias-terms which corresponds to the first column in  $P_{H1}$ . The subsequent weights and bias matrices  $P_{Hl}$  for a hidden layer  $Hl$ , where  $l = 2, 3, \dots, N_l$ , are as follows  $N_{Hl} \times (N_{H(l-1)} + 1)$ ; so for  $P_{H2}$  the matrix is given by  $N_{H2} \times (N_{H1} + 1)$ . As for the output layer  $P_{output}$  is an  $N_{output} \times (N_{HN_l} + 1)$  matrix, where for our case the number of neurons is  $N_{output} = 1$ . As always the first column in each of these matrices represents the bias of each neuron and the remaining columns represents the weights to each neuron.

The NN optimizes the set of weights and biases  $P$  such that the trial function eq. 9 satisfies eq. 5. In other words what we aim for is that our solver would optimize  $P$  such that

$$\frac{\partial g_t(x, t, P)}{\partial t} = \frac{\partial^2 g_t(x, t, P)}{\partial x^2} \quad (10)$$

The lhs. and rhs. are computed separately then the neural network must choose the weights and biases contained in  $P$ , such that each sides of eq. 10 are as equal to each other as possible. Thus, the neural network chooses  $P$  so that the mean squared cost function is as close as possible to zero.

$$\min_P \mathcal{C}(\mathbf{x}, P) \approx 0$$

For each iteration the neural network feeds forward the inputs. This means that  $\mathbf{x}$  is first passed through an input layer then through all  $N_l$  hidden layers before arriving to a generated weighted output. The next step is to utilize the weighted output to update and optimize

the weights and biases of the neural network through backpropagation. Backpropagation computes the gradient of the cost function wrt. the set of weights and biases  $P = \{P_{H1}, P_{H2}, \dots, P_{HN_l}, P_{output}\}$  and updates the matrices with their corresponding tweaked weights and biases in order to minimize the cost function.

$$\begin{aligned} P_{H1}^* &= P_{H1} - \eta \nabla_{P_{H1}} \mathcal{C}(\mathbf{x}, P) \\ P_{H2}^* &= P_{H2} - \eta \nabla_{P_{H2}} \mathcal{C}(\mathbf{x}, P) \\ &\vdots \\ P_{HN_l}^* &= P_{HN_l} - \eta \nabla_{P_{HN_l}} \mathcal{C}(\mathbf{x}, P) \\ P_{output}^* &= P_{output} - \eta \nabla_{P_{output}} \mathcal{C}(\mathbf{x}, P) \end{aligned}$$

These are the basic ideas behind feeding forward in a neural network and how backpropagation utilizes gradient descent to update the weights and biases. For a thorough explanation of both the feed forward and backpropagation schemes along with different implementation of gradient descent, we refer the reader to our previous work in [4] and the lecture notes [3].

A possible trial function that fits the diffusion equation and the conditions followed by eq. 4 is

$$g_t(x, t, P) = h_1(x, t) + x(1 - x)t \cdot N(x, t, P)$$

We have to chose  $h_1(x, t)$  so that the initial and boundary conditions are satisfied, a way of achieving just that would be by setting

$$h_1(x, t) = (1 - t) \sin(\pi x)$$

So the trial function for this problem is given by

$$g_t(x, t, P) = (1 - t) \sin(\pi x) + x(1 - x)t \cdot N(x, t, P)$$

Check **V Appendix i** for a more detailed calculation of the trial function.

### iii. Eigenvalues of a Symmetric Matrix

Studying the work of Yi et al. in [5] where they dive on how one can compute the eigenvector of any real symmetric matrix  $A$ . The neural network model proposed by Yi et al. is a nonlinear differential equation, given as

$$\frac{dx(t)}{dt} = -x(t) + f(x(t)) \quad (11)$$

for  $t \geq 0$  and where

$$f(x) = \left[ x^T x A + (1 - x^T A x) I \right] x \quad (12)$$

and where  $x(t)$  is a vector defined as  $x = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n$ . The neural networks model takes advantage of the fact that any given vector  $\xi \in \mathbb{R}^n$  is an equilibrium point of eq. 11 if and only if it satisfies the condition  $-\xi + f(\xi) = 0$ , meaning

A calculation deriving eq. 13 can be found in **V Appendix ii**.

$$\begin{aligned} -\xi + \left[ \xi^T \xi A + (1 - \xi^T A \xi) I \right] \xi &= 0 \\ -\xi + \xi^T \xi A \xi + \xi - \xi^T A \xi \xi &= 0 \\ \xi^T \xi A \xi - \xi^T A \xi \xi &= 0 \end{aligned}$$

Here we refer the reader to the thorough mathematical proof in [5], on how the set of equilibrium points, denoted  $E$ , is equal to the union of the eigenspace, denoted  $V_\lambda$  of the matrix  $A$  for a given eigenvalue  $\lambda$ . The condition for the equilibrium point is satisfied if  $\xi = \mathbf{0}$ , which is a trivial case that clearly always holds true. Thus, what we aim to solve for is the case where  $\xi \neq \mathbf{0}$  with an eigenvalue  $\lambda$  of  $A$  such that  $\xi \in V_\lambda$ .

So given that we can find the equilibrium point of eq. 11, the result would then be an arbitrary eigenvector  $\eta$  of  $A$ , where  $\eta \in E$ . Giving us

$$\begin{aligned} \eta^T \eta A \eta - \eta^T A \eta \eta &= 0 \\ \eta^T \eta A \eta &= \eta^T A \eta \eta \\ A \eta &= \frac{\eta^T A \eta}{\eta^T \eta} \eta \end{aligned}$$

Now for any symmetric real matrix its eigenvector  $v$  and corresponding eigenvalue  $\lambda$  must satisfy the condition

$$Av = \lambda v$$

Which leaves us with the fact that if the eigenvector  $v$  is predeterminedly defined

$$Av = \frac{v^T Av}{v^T v} v$$

Then the eigenvalue  $\lambda$  is given by

$$\lambda = \frac{v^T Av}{v^T v}$$

The solution of eq. 11 starting with any nonzero vector  $v \in \mathbb{R}^n$  will converge to an eigenvector of  $A$ , according to *Theorem 3 in [5]*. Additionally if a starting vector  $v \in \mathbb{R}^n$  is not orthogonal to the eigenspace  $V_{\lambda_1}$ , where  $\lambda_1$  is the smallest eigenvalue of  $A$ . Then the solution for eq. 11 converges to an eigenvector corresponding to the largest eigenvalue, according to *Theorem 4 in [5]*. Letting the algorithm solve for  $-A$  instead of  $A$  starting with any nonzero vector  $v \in \mathbb{R}^n$  that is not orthogonal to the eigenspace  $V_{\lambda_m}$ , where  $\lambda_m$  is the largest eigenvalue of  $A$ . Then the solution for eq. 11 converges to an eigenvector corresponding to the smallest eigenvalue, according to *Theorem 5 in [5]*.

### Euler's Method

In order to assess the validity of the model we will contrast the results of the neural network to that of the explicit scheme (Forward Euler). The discretized form of eq. 11 reads

$$x^{n+1} = x^n + \Delta t \left( (x^n)^T x^n A x^n - (x^n)^T A x^n x^n \right) \quad (13)$$

### III. RESULT

#### i. The Diffusion Equation

We set out to test how Feed Forward Neural Network (FFNN) based on the work of Kristine in [3] can be used to solve the diffusion equation in one spatial dimension. Here we resorted to the explicit scheme as another numerical approach, known to give reliable and accurate results for the problem at hand, to contrast it to the results produced by the neural network. We also contrasted both numerical approaches to the analytical solution of prob. 4, see **V Appendix i**. The latter two are used as a benchmark and a gauge of the neural network's accuracy and ability of producing reliable results.

We wanted to find an optimal preset for the general parameters and the parameters responsible of constructing the structure of the NN, i.e. the number of hidden layers NHL, number of hidden nodes NHN and number of epochs *epochs*. An epoch is a full iteration over the NN's Feed Forward and Backpropagation. The general parameters throughout our testing were set to

$$T = 1, \quad N_x = 10, \quad N_t = 20, \quad \text{num\_epochs} = 1000$$

The analytical solution was used to calculate the MSE (mean squared error) for a giving time step  $t_n$  and the average MSE of the numerical scheme for the entirety of the time domain  $t \in [0, T]$ , meaning

$$MSE = \frac{1}{N_x - 1} \sum_{i=1}^{N_x-1} (u_e(x_i, t_n) - u(x_i, t_n))^2$$

$$MSE_{avg} = \frac{1}{N_t} \sum_n MSE(t_n)$$

where  $u_e$  is the exact/analytical solution and  $u$  is the output from one of the numerical solutions either the explicit scheme or FFNN and where we excluded the boundary points (zero at  $x = 0$  and  $x = L$ ) for the calculation of the MSE. The propagation in time given by the explicit scheme together with the analytical result for this preset is

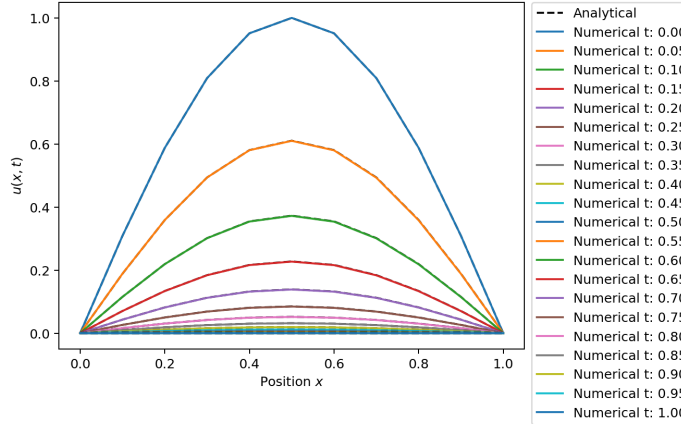


Figure 1. The diffusion equation solved using Euler's method for  $\Delta x = 0.1$ ,  $F = 0.01$  and  $\Delta t = 0.0001$ . Each line is a step in time, starting at  $t = 0$  blue solid line and stopping after a full time period at  $t = 1$ ; analytical result black dashed line.

shown in fig. 1. In addition, fig. 9 in **V Appendix iii** shows the results for the same solutions with a finer mesh  $N_x = 100$  and  $N_t = 2.5 \cdot 10^5$ . Just to demonstrate that the explicit scheme is capable of replicating the results with a higher level of precision. Where the calculated average MSE of the results shown in both figure was a resounding zero, which can be also observed from the figures themselves. Our aim is to construct the neural network in such a way that we reduce the offset from the analytical solution as much as possible, as we do not expect our NN to achieve the same level of accuracy.

#### Number of Hidden Nodes and Layers

Thus our goal is to find a combination of number of hidden layers and number of hidden nodes that achieves adequate results, mainly through data analysis i.e. using our metrics for error and the all too tedious but reliable approach of trial and error.

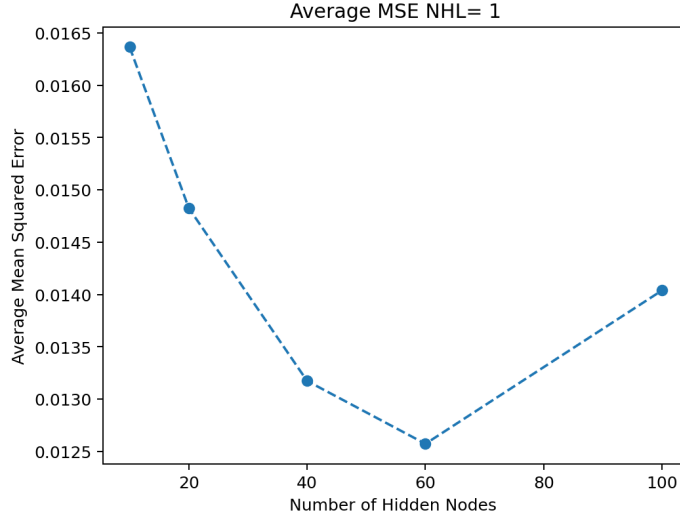


Figure 2. The average MSE for the entire numerical scheme of the FFNN as a result of the number of hidden nodes  $NHN = [10, 20, 40, 60, 100]$ .

Fig. 2 clearly shows that setting  $NHN = 60$  yields the best results in terms of average MSE across the time domain for the entire numerical scheme. Thus, we chose to highlight the performance of this initial preset of the NN in fig. 3.

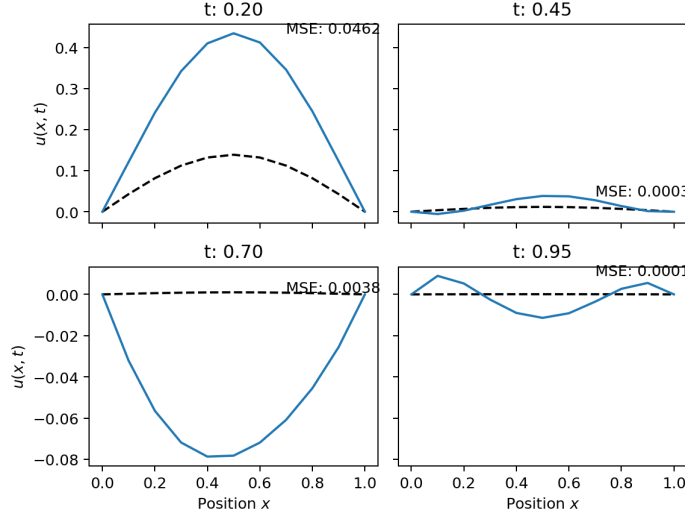


Figure 3. The NN's solution (solid blue line) for one hidden layer and  $NHN = 60$  contrasted against the analytical solution (dashed black line) together with the MSE at different time steps  $t$ .

Given these results we decided to set the number of hidden nodes in the first layer to 60 for our testing of the neural network with two hidden layers.

Following the same type of analysis as for one hidden layer the results for the two hidden layers case are shown in fig. 10 and fig. 11 in **V Appendix iii**. Fig. 10 shows that setting  $NHN = 60$  results in the lowest average MSE. The results found until now indicate that the best possible structure for the neural network is a symmetric structure. Thus, for our final testing we naturally tested for  $NHN = 60$  in the third layer.



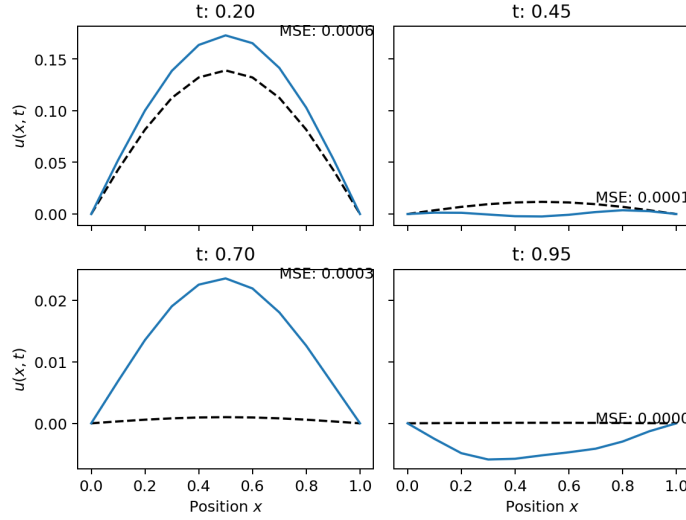


Figure 4. The NN's solution (solid blue line) for three hidden layers and  $\text{NHN} = [60, 60, 60]$  contrasted against the analytical solution (dashed black line) together with the MSE at different time steps  $t$ .

| $t$     | MSE    |        |        |        | Avg MSE        |
|---------|--------|--------|--------|--------|----------------|
|         | 0.20   | 0.45   | 0.70   | 0.95   | $t \in [0, 1]$ |
| NHL = 1 | 0.0462 | 0.0003 | 0.0038 | 0.0001 | 0.0126         |
| NHL = 2 | 0.0072 | 0.0009 | 0.0001 | 0.0000 | 0.0031         |
| NHL = 3 | 0.0006 | 0.0001 | 0.0003 | 0.0000 | 0.0012         |

Table I. A complete analysis of the mean squared error of all the symmetric NN structure with  $\text{NHN} = 60$  at time step  $t$ . In addition to the corresponding average MSE for the entire time domain,  $t \in [0, 1]$ , for every NHL.

Tab. I summarizes the results of the MSE for each optimal case we arrived at every step of the way. The last one being the neural network constructed with three hidden layers and  $\text{NHN} = [60, 60, 60]$ . These are the results that we have discussed and can be seen in figures such as fig. 3, fig. 4 and fig. 11.

In addition fig. 12 in **V Appendix iii** shows how the final solution produced by the 3-layered neural network behaves contrasted against the analytical solution. Which is the same type of time propagation seen in fig. 1, although fails in comparison to the output from the explicit scheme.

## ii. Eigenvalues of a Symmetric Matrix

We based our approach on the findings of Yi et al. in [5]. We studied the efficacy of our FFNN, built using [TensorFlow Core v2.7.0](#), in finding the eigenvectors that corresponds to the smallest and largest eigenvalues of a  $6 \times 6$  matrix. We contrasted our results to that of Euler's method solving eq. 11 and [NumPy's numpy.linalg.eig](#) to obtain the eigenvalues and eigenvectors of the matrix. Where the former two were used as benchmark for accuracy and convergence and also gauge how different parameters can effect the end results of the NN.

The fixed layers in the NN are the *input* and the *output* layers, where both contain 6 nodes. Since the initial guess naturally is a 6 component vector and the result produced is a 6 component eigenvector. As for the hidden layers that number along with the number of the hidden nodes is specified by the user. Our analysis based itself on a randomly generated (but seeded)  $6 \times 6$  matrix  $A$  and an initial vector  $x_0 \in \mathbb{R}^6$ . We chose so in order to be able to establish an *optimal* preset, in terms of time period  $T$ , number of mesh points  $N_t$ , number of hidden layers NHL, number of hidden nodes NHN and number of epochs. We start by introducing the problem at hand and the matrix and initial vector we solve for

$$x_0 = \begin{bmatrix} 0.3117 \\ 0.5201 \\ 0.5467 \\ 0.1849 \\ 0.9696 \\ 0.7751 \end{bmatrix} \quad A = \begin{bmatrix} 0.4967 & 0.7205 & 0.4448 & 0.3075 & -0.3893 & -0.4179 \\ 0.7205 & 0.7674 & -1.1914 & -0.4349 & -0.1762 & 0.6933 \\ 0.4448 & -1.1914 & -1.7249 & 0.4517 & -1.0819 & 0.1504 \\ 0.3075 & -0.4349 & 0.4517 & -0.2258 & 0.2216 & -1.2412 \\ -0.3893 & -0.1762 & -1.0819 & 0.2216 & -0.6006 & 0.2654 \\ -0.4179 & 0.6933 & 0.1504 & -1.2412 & 0.2654 & -1.2208 \end{bmatrix}$$



With eigenvectors and corresponding eigenvalues

$$\begin{aligned}
\mathbf{v}_1 &= \begin{bmatrix} -0.1344 \\ 0.3054 \\ 0.715 \\ -0.2571 \\ 0.3724 \\ -0.4154 \end{bmatrix} & \mathbf{v}_2 &= \begin{bmatrix} 0.1143 \\ 0.7897 \\ -0.3068 \\ -0.4048 \\ 0.0591 \\ 0.3191 \end{bmatrix} & \mathbf{v}_3 &= \begin{bmatrix} -0.8056 \\ -0.2445 \\ -0.1691 \\ -0.3293 \\ 0.2954 \\ 0.2586 \end{bmatrix} & \mathbf{v}_4 &= \begin{bmatrix} 0.5603 \\ -0.4068 \\ -0.0417 \\ -0.3597 \\ 0.591 \\ 0.2003 \end{bmatrix} & \mathbf{v}_5 &= \begin{bmatrix} 0.0729 \\ -0.2121 \\ 0.4024 \\ -0.5558 \\ -0.6254 \\ 0.2963 \end{bmatrix} & \mathbf{v}_6 &= \begin{bmatrix} -0.0245 \\ 0.1132 \\ 0.4499 \\ 0.4725 \\ 0.1735 \\ 0.7285 \end{bmatrix} \\
\lambda_1 &= -3.1308 & \lambda_2 &= 1.8243 & \lambda_3 &= 1.2113 & \lambda_4 &= -0.8169 & \lambda_5 &= 0.1522 & \lambda_6 &= -1.7481
\end{aligned}$$

The max and min eigenvalues with their corresponding eigenvectors are then

$$\begin{aligned}
\mathbf{v}_{\min} &= \begin{bmatrix} -0.1344 \\ 0.3054 \\ 0.715 \\ -0.2571 \\ 0.3724 \\ -0.4154 \end{bmatrix} & \mathbf{v}_{\max} &= \begin{bmatrix} 0.1143 \\ 0.7897 \\ -0.3068 \\ -0.4048 \\ 0.0591 \\ 0.3191 \end{bmatrix} \\
\lambda_{\min} &= -3.1308 & \lambda_{\max} &= 1.8243
\end{aligned}$$

The results for the eigenvectors and eigenvalues were obtained using `numpy.linalg.eig` and will be used to assess the validity of the results generated by the FFNN and the explicit scheme for a multitude of presets.

### Number of Hidden Nodes and Layers

Our aim is to arrive at an optimal preset to be used in subsequent testing. We started by studying how the number of nodes in the first hidden layer effects the results at hand and if any gain comes from increasing the number of nodes. In order to do that we applied a rigorous presets to all the other parameters. Where we chose, what we think is a sufficiently large time period  $T$ , number of mesh points  $N_t$  for both the NN and the explicit scheme and finally number of epochs. So no bottleneck would occur. Our parameters throughout our testing were set to

$$T = 10, \quad \mathbf{NN}: N_t = 101, \quad \mathbf{Euler}: N_t = 1001, \quad \mathbf{num\_epochs}: 2000$$

We did our testing for the maximum and minimum case with equivalent analysis. To determine how each preset performed we paired the results from the NN and the explicit scheme against that obtained by NumPy. We calculated the absolute error (AE) of the eigenvalues and the mean absolute error (MAE) of the eigenvectors, meaning

$$\begin{aligned}
\text{AE}_\lambda &= |\lambda_{\mathbf{num}} - \lambda_{\mathbf{np}}| \\
\text{MAE}_v &= \frac{1}{N} \sum_i^N |v_i^{\mathbf{num}} - v_i^{\mathbf{np}}|
\end{aligned}$$

where **num** denotes numerical either the explicit scheme or FFNN and **np** denotes NumPy. The error in the explicit scheme for this preset is

| Error for the Explicit Scheme |            |                        |            |                         |
|-------------------------------|------------|------------------------|------------|-------------------------|
|                               | $v_{\max}$ | $\lambda_{\max}$       | $v_{\min}$ | $\lambda_{\min}$        |
| $\epsilon$                    | 0.0026     | $1.8475 \cdot 10^{-8}$ | 0.0018     | $4.5990 \cdot 10^{-12}$ |

Table II. The absolute error of the eigenvalues and the mean squared error of the eigenvectors for the explicit scheme, with  $N_t = 1001$  mesh points.

Ideally the neural network if constructed correctly should yield results close to that of tab. II if not better.

| NHL = 1          | NHN    |        |        |        |
|------------------|--------|--------|--------|--------|
| AE for           | 25     | 50     | 100    | 200    |
| $\lambda_{\max}$ | 0.0027 | 0.0025 | 0.0023 | 0.0072 |
| $\lambda_{\min}$ | 0.0473 | 0.0260 | 0.0011 | 0.0105 |

Table III. The absolute error of the eigenvalues  $\lambda_{\max}$  and  $\lambda_{\min}$  as a result of the number of hidden nodes  $\text{NHN} = [25, 50, 100, 200]$  for one hidden layer.

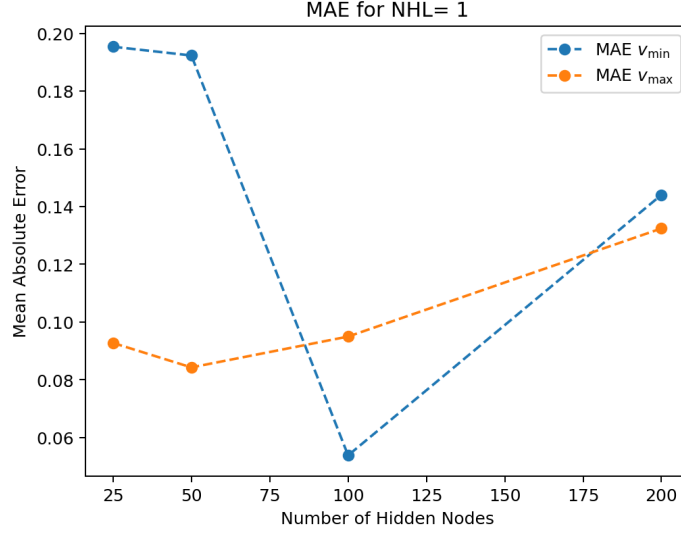


Figure 5. The mean absolute error of the eigenvectors  $v_{\max}$  and  $v_{\min}$  as a result of the number of hidden nodes  $\text{NHN} = [25, 50, 100, 200]$  for one hidden layer.

Tab. III and fig. 5 clearly show that setting  $\text{NHN} = 100$  yields the best results in terms of error for both the eigenvalues and eigenvectors; when both the case for max and min is taken under consideration. Since the mean absolute error of  $v_{\max}$  did increase for  $\text{NHN} = 100$ . Concluding with these results and venturing off to the case with two hidden layers, the number of hidden nodes in the first layer is set to  $\text{NHN} = 100$  from now on.

Following the same type of analysis as for one hidden layer, the results obtained using two hidden layers are shown in Tab. IV and fig. 6

| NHL = 2          | NHN                    |                         |        |        |
|------------------|------------------------|-------------------------|--------|--------|
| AE for           | 10                     | 25                      | 50     | 100    |
| $\lambda_{\max}$ | $4.5244 \cdot 10^{-5}$ | 0.0003                  | 0.0004 | 0.0039 |
| $\lambda_{\min}$ | $3.9429 \cdot 10^{-5}$ | $8.5990 \cdot 10^{-05}$ | 0.0003 | 0.0301 |

Table IV. The absolute error of the eigenvalues  $\lambda_{\max}$  and  $\lambda_{\min}$  as a result of the number of hidden nodes  $\text{NHN} = [10, 25, 50, 100]$  in the second hidden layer, for two hidden layers.

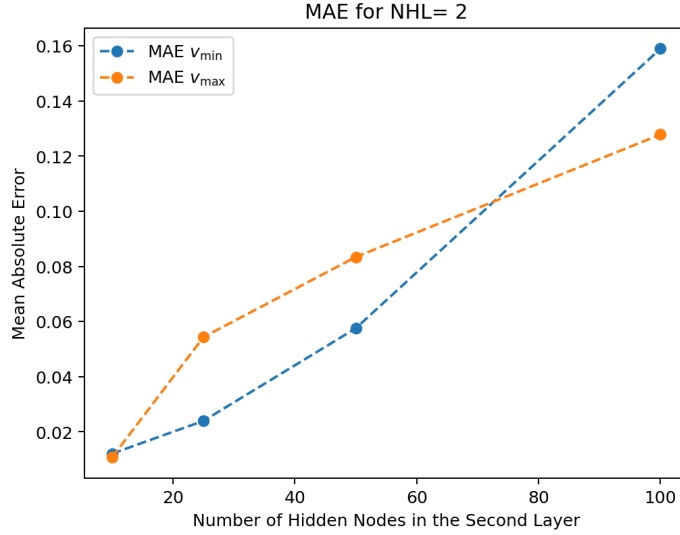


Figure 6. The mean absolute error of the eigenvectors  $v_{\max}$  and  $v_{\min}$  as a result of the number of hidden nodes  $NHN = [10, 25, 50, 100]$  in the second hidden layer, for two hidden layer.

What we obtained shows that adding an extra layer decreases the error significantly. Additionally, given the results from fig. 6 we can see that the error increases almost proportionally with the number of hidden nodes, as  $NHN$  approaches the number of hidden nodes in the first layer. The best results are obtained by setting the number of hidden nodes in the second layer to  $NHN = 10$ . With an AE for the eigenvalues of  $-5$  order of magnitude and a MAE lower than 0.02, which is a clear improvement over the one hidden layer case.

Since the neural network achieved better results by adding a hidden layer, we went ahead and tested the case of three hidden layers, tab. VI and fig. 13 in V Appendix iii. The results obtained weren't very promising but the trend of error increasing as  $NHN$  approaches the number of hidden nodes in the prior layer continued. With  $NHN = 20$  being somewhat of an outlier as the error decreased but ultimately none of the combinations for the three hidden layers case yielded any concrete and reliable improvements. Although, setting  $NHN = 3$  in the third layer did give better results they weren't of the order of magnitude we had hoped for, see tab. IV for  $NHN = 10$  and tab. VI for  $NHN = 3$ . In addition, it's worth mentioning that the output layer contains 6 nodes since we're working with  $6 \times 6$  matrix and naturally the eigenvector produced is  $v \in \mathbb{R}^6$ . So we want to structure the NN so as the number of hidden nodes in the hidden layer prior to the output is  $NHN \geq 6$ .

This made us rethink how we should structure our neural network. Clearly  $NHN = 100$  in the first hidden layer yields the best results and we observed that  $NHN$  in the following layer must not approach the number of hidden nodes in the proceeding layer. Thus, we went a head and tested for the case where we half the number of hidden nodes in each following layer in the NN, meaning  $NHN = [100, 50, 25]$ . The results obtained are shown in tab. VII in V Appendix iii and it yielded no improvements contrasted to that of the two hidden layers case. Given these results for our further testing with different matrices and initial vectors we will be working with 2 hidden layers with 100 nodes in the first layer and 10 nodes in the second layer, in other words  $NHL = 2$  and  $NHN = [100, 10]$ .

### First Test Case

The first test case preset and values are all highlighted in the section called **First Test Case** in V Appendix iii. For this test case we wanted to see how well the neural network performs and confirm that it can be used for other combinations of  $A$  and  $x_0$ , showing that our optimization analysis indeed works for other cases.

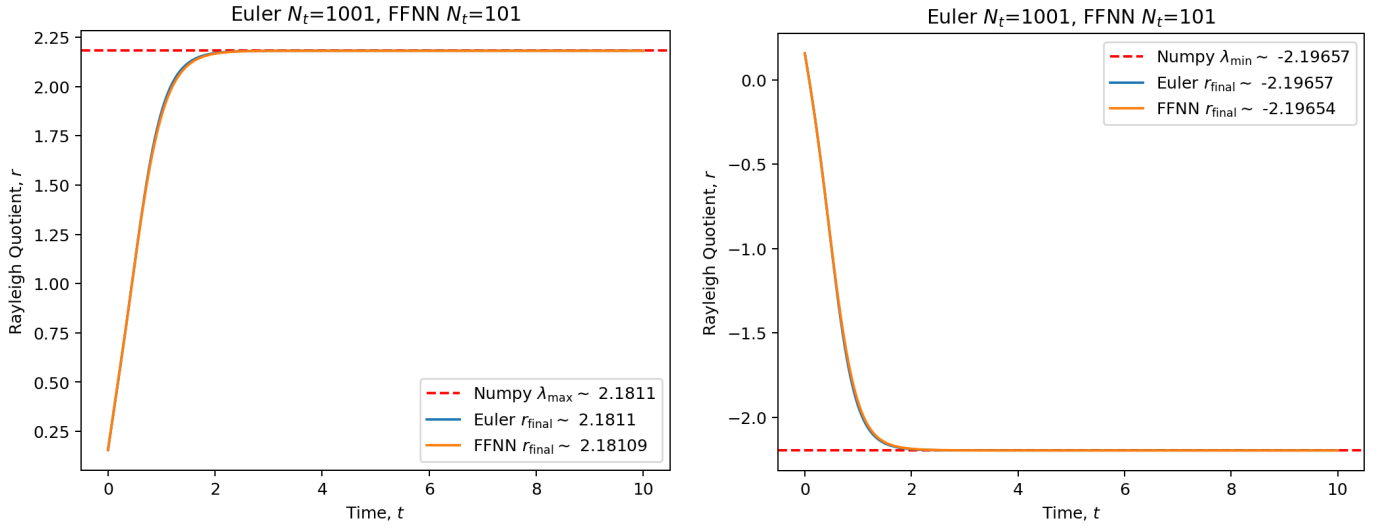


Figure 7. Lhs. max eigenvalue and rhs. min eigenvalue. NumPy dashed red line, Euler solid blue line, FFNN solid orange line.

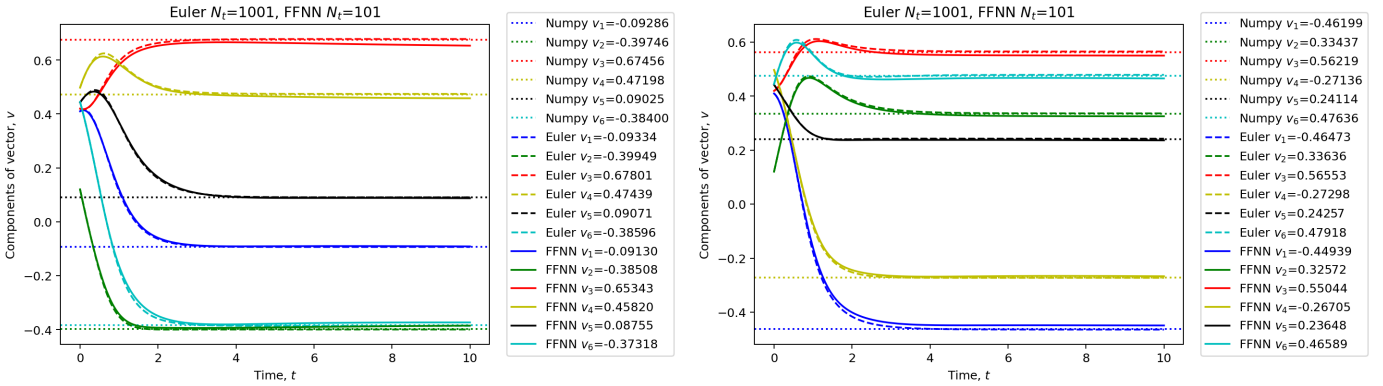


Figure 8. Eigenvector components: lhs. corresponds to max eigenvalue and rhs. corresponds to min eigenvalue. NumPy vector components dotted line, Euler vector components dashed line, FFNN vector components solid line.

Both fig. 7 and fig. 8 show how well the NN performs for a different  $6 \times 6$  matrix  $A$  and initial vector  $x_0$ . It also shows that the eigenvalues and their corresponding eigenvectors had long converged at  $t = 4$ . Which in itself shows that the preset we chose is adequate when it came to optimizing the neural network and that the neural network itself yield reliable results. But exceptions do occur and they're mainly the cause of a bad initial guess for  $x_0$ , which we will dive deeper into in the following section. Many of the behaviors that the solution can have is discussed in [5]; check the theorems listed in the article.

### Second Test Case

The second test case preset and values are all highlighted in the section called **Second Test Case** in **V Appendix iii**. For this test case we wanted to inspect how different initial guesses of the initial vector effect the end result. The testing was done for the same matrix  $A$  with different initial vector each time, denoted  $x_0, x_1$ .

| NHL = 2    | FFNN NHN = [100, 10] |                        |
|------------|----------------------|------------------------|
| $\epsilon$ | $\lambda_{\max}$     | $\lambda_{\min}$       |
| $x_0$      | 0.0007               | $1.3368 \cdot 10^{-5}$ |
| $x_1$      | 3.4819               | $1.1576 \cdot 10^{-5}$ |

Table V. The absolute error of the eigenvalues and the mean squared error of the eigenvectors for the neural network with two hidden layers,  $\text{NHN} = [100, 10]$ , as a result of multiple initial vectors  $x$ .

Looking at tab. V we notice although an initial guess might yield bad results for the maximum case, it is not necessarily a bad initial guess for the minimum case. The initial guess had severe impact on the result for the maximum eigenvalue but as for the minimum eigenvalue the results were within the margins, with absolute errors that are more or less alike. In addition, from fig. 14 in V Appendix iii we can see that the NN behaves similarly to that of Euler and follows the same trend as before and that is the convergence happens well before  $t = 4$ . The only outlier is  $x_1$  as the start vector for  $\lambda_{\max}$ , where we see that the NN converges at the same rate as before but to a completely different value from  $\lambda_{\max}$ . This could be due to the fact that the optimizer (ADAM in this case) responsible of updating the weights and biases of this preset with  $x_1$  as the initial guess converged to a so called local minimum and thus no further alterations has been made to the parameters of the neural network.

## IV. CONCLUSION

### i. The Diffusion Equation

The results from the neural network were subpar if we can describe it as such and also extremely time demanding. Where on average a computation involving one hidden layer took around 50 minutes and 1 hour 40 minutes for the two and three layers case. This doesn't necessarily mean that neural networks can't be used to solve partial differential equations related problems. It only points to that a lot of alterations to our code is needed in order for it to be reliable, accurate and somewhat fast. Through our testing the neural network only managed to be fully accurate at the initial time step  $t = 0$  but it never came close to being as accurate as the results produced by the explicit scheme, fig. 12. Although, what we can take from this is that our approach for data analysis in deciding the structure of the neural network is indeed reliable. Since our analysis always resulted in an error decrease as can be seen in tab. I. What we had in mind is to build on the neural network code with TensorFlow used to find the eigenvalue of a symmetric matrix. We had started altering the TensorFlow neural network to be able to solve the diffusion equation, but we didn't manage to make a reusable code before the deadline. We do believe that the NN built using TensorFlow for the eigenvector problem, if adjusted for the diffusion equation, would yield significantly better results than our neural network and while being considerably less time demanding. This conclusion is derived from the results of our testing of the eigenvector problem where the accuracy and reliability of the TensorFlow based neural network can be seen in the tables and figures. As for the CPU-time we have only measured the runtime of our own neural network, but a test run should suffice in showing that the TensorFlow code never took over 20 minutes to produce results and that is with 2000 epochs.

### ii. Eigenvalues of a Symmetric Matrix

Our testing for building up the structure of the neural network proved to be reliable for other cases than the case used for optimization, although with the need for a more rigorous testing. Perhaps with matrices and initial vectors combinations that are known to be hard to solve numerically, following the work of Yi et al. in [5] and especially focusing on what have been done in Section 4 & 5. Our optimization analysis showed that increasing the number of hidden nodes to a certain threshold is key in obtaining adequate and precise results. For the first hidden layer alone  $\text{NHN} = 100$  yielded the best and most reliable results, when both taking the absolute error of the eigenvalues and the mean absolute error of the eigenvectors under considerations. Our findings also showed that adding a hidden layers does increase the accuracy of the results produced to a certain extent. Firstly, the number of hidden nodes in the following layer must be considerably lower than the number of hidden nodes in the prior layer in order to achieve the best results. Secondly, adding layers could improve the results but with no guaranteed reliability and definitely not needed when studying a matrices of the size  $6 \times 6$ . We concluded that an NN with two hidden layers, with 100 and 10 hidden nodes respectively, yielded the best results for our case. Although, eigenvector problems such as these with  $N \times N$  matrices that are relatively large in size (bigger than what we have tested for) would require a different network structure. A higher number of nodes per hidden layer and more hidden layers in order to be accurate and reliable. We would have used the same testing methodology to determine the number of hidden layers and number of hidden nodes per layer and construct our initial NN that way. The

results generated by the neural network were accurate and precise enough, as shown by tab. [IV](#), fig. [6](#) and fig. [8](#). Except for when the initial guess at hand doesn't quite fit the test case, as can be seen in tab. [V](#) for  $x_1$  as the initial guess.

## REFERENCES

- [1] M. Hjort-Jensen. Computational Physics [Lecture Notes](#), (2015)
- [2] Linge, Svein and Langtangen, Hans Petter. Finite Difference Computing with PDEs: A Modern Software Approach [Springer International Publishing](#), (2017)
- [3] K. Baluka Hein. Data Analysis and Machine Learning: Using Neural networks to solve ODEs and PDEs [Department of Informatics](#), University of Oslo (2018)
- [4] M. Mahmoud and A. Davidov. Classification, Regression and FFNN. [PDF](#), (2021)
- [5] Z. Yi, Y. Fu, H. Jin Tang, Neural networks based approach for computing eigenvectors and eigenvalues of symmetric matrix, Computers & Mathematics with Applications, Volume 47, Issues 8–9, 2004, Pages 1155-1164, ISSN 0898-1221, [ScienceDirect](#), (2004)



## V. APPENDIX

### i. The Diffusion Equation

#### Analytical Solution

$$\frac{\partial u(x, t)}{\partial t} = \frac{\partial^2 u(x, t)}{\partial x^2}, \quad t > 0, \quad x \in [0, L] \quad (14)$$

with the initial condition:

$$u(x, 0) = \sin(\pi x) \quad 0 < x < L$$

where  $L = 1$  and the boundary conditions are set to

$$u(0, t) = 0 \quad t \geq 0 \quad u(L, t) = 0 \quad t \geq 0$$

Assuming that the solution for eq. 14 takes the form

$$u(x, t) = f(x)g(t)$$

We have that for the initial condition

$$u(x, 0) = f(x)g(0) = \sin(\pi x)$$

meaning

$$f(x) = \sin(\pi x) \quad g(0) = 1$$

The general expression for  $f(x)$  is actually

$$f(x) = \sin\left(\frac{\pi}{L}x\right)$$

but since in our case  $L = 1$

$$f(x) = \sin(\pi x)$$

Clearly  $f(x)$  satisfies the boundary conditions. Now for the derivative in eq. 14

$$\frac{\partial u(x, t)}{\partial t} = f(x)g'(t) = \sin(\pi x)g'(t), \quad \frac{\partial^2 u(x, t)}{\partial x^2} = \frac{\partial}{\partial x} \pi \cos(\pi x)g(t) = -\pi^2 \sin(\pi x)g(t)$$

Solving for  $g(t)$

$$\begin{aligned} \frac{\partial u(x, t)}{\partial t} &= \frac{\partial^2 u(x, t)}{\partial x^2} \\ \sin(\pi x)g'(t) &= -\pi^2 \sin(\pi x)g(t) \\ g'(t) &= -\pi^2 g(t) \rightarrow g(t) = \exp(-\pi^2 t) \end{aligned}$$

The analytical expression for  $u(x, t)$  is then given by

$$u(x, t) = \sin(\pi x) \exp(-\pi^2 t)$$

#### Explicit Scheme

$$\frac{\partial u(x, t)}{\partial t} = \frac{\partial^2 u(x, t)}{\partial x^2}$$

Discretization forward in time and centered in space

$$\frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} = \frac{u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t)}{\Delta x^2}$$

at a mesh point  $(x_i, t_n)$

$$\frac{u(x_i, t_n + \Delta t) - u(x_i, t_n)}{\Delta t} = \frac{u(x_i + \Delta x, t_n) - 2u(x_i, t_n) + u(x_i - \Delta x, t_n)}{\Delta x^2}$$

a more compact notation

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}$$

Solving for  $u_i^{n+1}$

$$u_i^{n+1} - u_i^n = \frac{\Delta t}{\Delta x^2} (u_{i+1}^n - 2u_i^n + u_{i-1}^n)$$

$F = \frac{\Delta t}{\Delta x^2}$  the Fourier number

$$u_i^{n+1} - u_i^n = F(u_{i+1}^n - 2u_i^n + u_{i-1}^n)$$

Finally

$$u_i^{n+1} = u_i^n + F(u_{i+1}^n - 2u_i^n + u_{i-1}^n)$$

### **Trial Function**

$$g_t(\mathbf{x}, P) = h_1(\mathbf{x}) + h_2(\mathbf{x}, N(\mathbf{x}, P))$$

Setting in for  $\mathbf{x}$ :

$$g_t(x, t, P) = h_1(x, t) + h_2(x, t, N(x, t, P))$$

$h_1$  should alone satisfy the initial and boundary conditions. A possible trial solution for this PDE:

$$g_t(x, t, P) = h_1(x, t) + x(1-x)t \cdot N(x, t, P)$$

A way of fulfilling the conditions at hand can be achieved by setting:

$$h_1(x, t) = (1-t) \left( u(x) - ((1-x)u(0) + xu(1)) \right) = (1-t)u(x) = (1-t)\sin(\pi x)$$

The final expression for the trial function:

$$g_t(x, t, P) = (1-t)\sin(\pi x) + x(1-x)t \cdot N(x, t, P)$$

## **ii. Eigenvalues of a Symmetric Matrix**

### **Euler's Method**

$$\begin{aligned} \frac{dx(t)}{dt} &= -x(t) + f(x(t)) \\ \frac{x^{n+1} - x^n}{\Delta t} &= -x^n + f(x^n) \end{aligned}$$

where  $x^n = x(t_n) = x(n\Delta t)$

$$\begin{aligned} \frac{x^{n+1} - x^n}{\Delta t} &= (x^n)^T x^n A x^n - (x^n)^T A x^n x^n \\ x^{n+1} &= x^n + \Delta t \left( (x^n)^T x^n A x^n - (x^n)^T A x^n x^n \right) \end{aligned}$$

## Number of Hidden Nodes and Layers

### iii. Figures & Tables

#### The Diffusion Equation

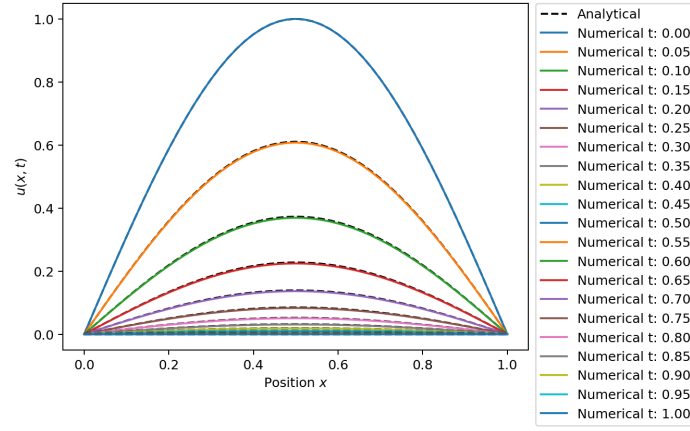


Figure 9. The diffusion equation solved using Euler's method for  $\Delta x = 0.02$ ,  $F = 0.01$  and  $\Delta t = 4 \cdot 10^{-4}$ . Each line is a step in time, starting at  $t = 0$  blue solid line and stopping after a full time period at  $t = 1$ ; analytical result black dashed line.

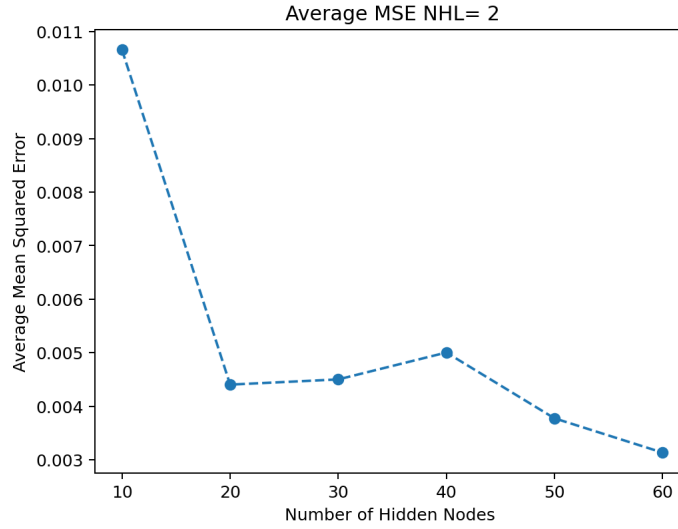


Figure 10. The average MSE for the entire numerical scheme of the FFNN as a result of the number of hidden nodes in the second layer  $NHN = [10, 20, 30, 40, 50, 60]$ .

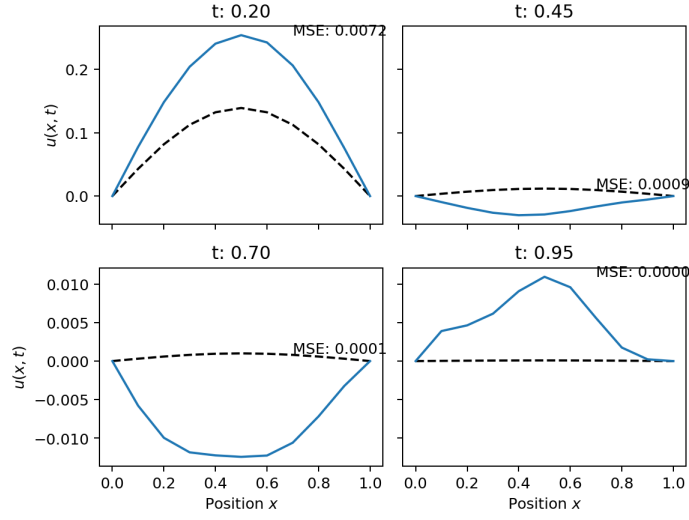


Figure 11. The NN's solution (solid blue line) for two hidden layer and  $\text{NHN} = [60, 60]$  contrasted against the analytical solution (dashed black line) together with the MSE at different time steps  $t$ .

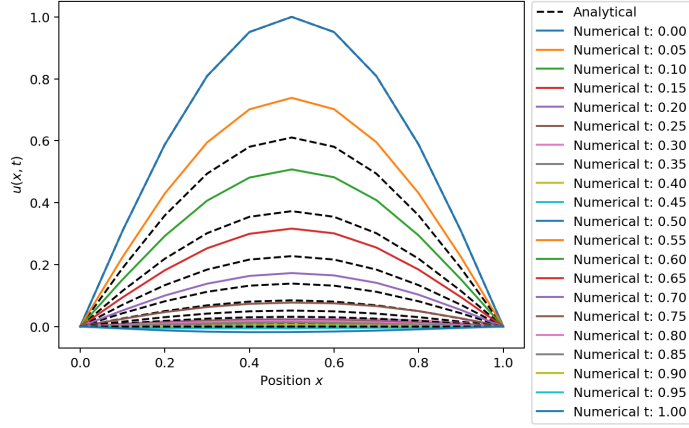


Figure 12. The diffusion equation solved using FFNN with three hidden layers and  $\text{NHN} = [60, 60, 60]$ . Each line is a step in time, starting at  $t = 0$  blue solid line and stopping after a full time period at  $t = 1$ ; analytical result black dashed line.

### Eigenvalues of a Symmetric Matrix

| NHL = 3          | NHN                    |                        |        |                        |
|------------------|------------------------|------------------------|--------|------------------------|
|                  | 3                      | 6                      | 10     | 20                     |
| $\lambda_{\max}$ | $2.9238 \cdot 10^{-5}$ | $9.1230 \cdot 10^{-5}$ | 0.0005 | $8.9394 \cdot 10^{-5}$ |
| $\lambda_{\min}$ | $6.4527 \cdot 10^{-5}$ | $8.2438 \cdot 10^{-5}$ | 0.0001 | $1.7894 \cdot 10^{-5}$ |

Table VI. The absolute error of the eigenvalues  $\lambda_{\max}$  and  $\lambda_{\min}$  as a result of the number of hidden nodes  $\text{NHN} = [3, 6, 10, 20]$  in the third hidden layer, for three hidden layers.

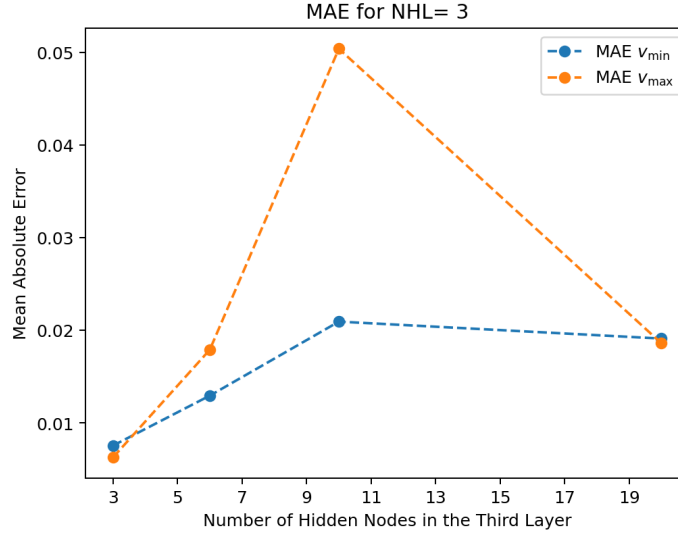


Figure 13. The mean absolute error of the eigenvectors  $v_{\max}$  and  $v_{\min}$  as a result of the number of hidden nodes  $\text{NHN} = [3, 6, 10, 20]$  in the third hidden layer, for three hidden layer.

|            | FFNN $\text{NHN} = [100, 50, 25]$ |                        |            |                  |
|------------|-----------------------------------|------------------------|------------|------------------|
|            | $v_{\max}$                        | $\lambda_{\max}$       | $v_{\min}$ | $\lambda_{\min}$ |
| $\epsilon$ | 0.0271                            | $3.6145 \cdot 10^{-5}$ | 0.0162     | 0.0001           |

Table VII. The absolute error of the eigenvalues and the mean squared error of the eigenvectors for the neural network with three hidden layers and  $\text{NHN} = [100, 50, 25]$  in each layer respectively.

#### First Test Case

$$x_0 = \begin{bmatrix} 0.6908 \\ 0.2029 \\ 0.7073 \\ 0.8384 \\ 0.7455 \\ 0.7494 \end{bmatrix} \quad A = \begin{bmatrix} -0.4 & -0.086 & 0.4304 & -0.609 & 0.3063 & 0.7928 \\ -0.086 & -0.3445 & -1.2878 & -0.0195 & -0.3014 & 0.2778 \\ 0.4304 & -1.2878 & 0.3413 & 0.8435 & -0.3063 & -1.0382 \\ -0.609 & -0.0195 & 0.8435 & 0.5769 & 0.4451 & -0.2178 \\ 0.3063 & -0.3014 & -0.3063 & 0.4451 & 0.3742 & -0.1777 \\ 0.7928 & 0.2778 & -1.0382 & -0.2178 & -0.1777 & -0.4315 \end{bmatrix}$$

With eigenvectors and corresponding eigenvalues

$$\begin{aligned} v_1 &= \begin{bmatrix} -0.462 \\ 0.3344 \\ 0.5622 \\ -0.2714 \\ 0.2411 \\ 0.4764 \end{bmatrix} & v_2 &= \begin{bmatrix} -0.0929 \\ -0.3975 \\ 0.6746 \\ 0.4720 \\ 0.0902 \\ -0.3840 \end{bmatrix} & v_3 &= \begin{bmatrix} -0.4253 \\ -0.7335 \\ -0.2497 \\ 0.0437 \\ -0.0745 \\ 0.4597 \end{bmatrix} & v_4 &= \begin{bmatrix} -0.4232 \\ -0.1496 \\ -0.1497 \\ -0.5507 \\ 0.3241 \\ -0.6065 \end{bmatrix} & v_5 &= \begin{bmatrix} 0.5148 \\ -0.2539 \\ 0.0085 \\ -0.1017 \\ 0.7842 \\ 0.2126 \end{bmatrix} & v_6 &= \begin{bmatrix} -0.3911 \\ 0.3247 \\ -0.3796 \\ 0.623 \\ 0.4563 \\ -0.0353 \end{bmatrix} \\ \lambda_1 &= -2.1966 & \lambda_2 &= 2.1811 & \lambda_3 &= -1.0362 & \lambda_4 &= -0.1689 & \lambda_5 &= 0.5636 & \lambda_6 &= 0.7735 \end{aligned}$$

The max and min eigenvalues with their corresponding eigenvectors are then

$$\begin{aligned} v_{\min} &= \begin{bmatrix} -0.462 \\ 0.3344 \\ 0.5622 \\ -0.2714 \\ 0.2411 \\ 0.4764 \end{bmatrix} & v_{\max} &= \begin{bmatrix} -0.0929 \\ -0.3975 \\ 0.6746 \\ 0.4720 \\ 0.0902 \\ -0.3840 \end{bmatrix} \\ \lambda_{\min} &= -2.1966 & \lambda_{\max} &= 2.1811 \end{aligned}$$

## Second Test Case

$$x_0 = \begin{bmatrix} 0.3739 \\ 0.5731 \\ 0.9174 \\ 0.8190 \\ 0.1632 \\ 0.0242 \end{bmatrix} \quad x_1 = \begin{bmatrix} 0.7387 \\ -0.4097 \\ 0.5774 \\ 0.4166 \\ 0.5215 \\ -1.4737 \end{bmatrix} \quad A = \begin{bmatrix} -1.2641 & -0.099 & -0.8674 & -0.8067 & -1.502 & -0.3982 \\ -0.099 & 1.5851 & -0.3204 & -0.0845 & 1.5976 & -0.6041 \\ -0.8674 & -0.3204 & 1.3838 & 0.3773 & 0.5699 & 0.6744 \\ -0.8067 & -0.0845 & 0.3773 & -0.4675 & -0.5439 & 0.0099 \\ -1.502 & 1.5976 & 0.5699 & -0.5439 & 0.985 & -1.1948 \\ -0.3982 & -0.6041 & 0.6744 & 0.0099 & -1.1948 & -0.8528 \end{bmatrix}$$

With eigenvectors and corresponding eigenvalues

$$\begin{aligned} v_1 &= \begin{bmatrix} 0.6949 \\ -0.0992 \\ -0.0235 \\ 0.3390 \\ 0.4840 \\ 0.3969 \end{bmatrix} & v_2 &= \begin{bmatrix} 0.2147 \\ -0.6449 \\ -0.0865 \\ 0.0562 \\ -0.6840 \\ 0.2440 \end{bmatrix} & v_3 &= \begin{bmatrix} 0.3357 \\ 0.2692 \\ -0.8385 \\ -0.2007 \\ -0.1401 \\ -0.2279 \end{bmatrix} & v_4 &= \begin{bmatrix} -0.4451 \\ 0.0252 \\ -0.3531 \\ -0.2198 \\ 0.1415 \\ 0.7799 \end{bmatrix} & v_5 &= \begin{bmatrix} 0.3951 \\ 0.2146 \\ 0.3801 \\ -0.7753 \\ -0.1209 \\ 0.1940 \end{bmatrix} & v_6 &= \begin{bmatrix} -0.0643 \\ -0.6747 \\ -0.1406 \\ -0.4384 \\ 0.4936 \\ -0.2917 \end{bmatrix} \\ \lambda_1 &= -2.8875 & \lambda_2 &= 3.5056 & \lambda_3 &= 2.2028 & \lambda_4 &= -1.1699 & \lambda_5 &= -0.3053 & \lambda_6 &= 0.0240 \end{aligned}$$

The max and min eigenvalues with their corresponding eigenvectors are then

$$\begin{aligned} v_{\min} &= \begin{bmatrix} 0.6949 \\ -0.0992 \\ -0.0235 \\ 0.3390 \\ 0.4840 \\ 0.3969 \end{bmatrix} & v_{\max} &= \begin{bmatrix} 0.2147 \\ -0.6449 \\ -0.0865 \\ 0.0562 \\ -0.6840 \\ 0.2440 \end{bmatrix} \\ \lambda_{\min} &= -2.8875 & \lambda_{\max} &= 3.5056 \end{aligned}$$

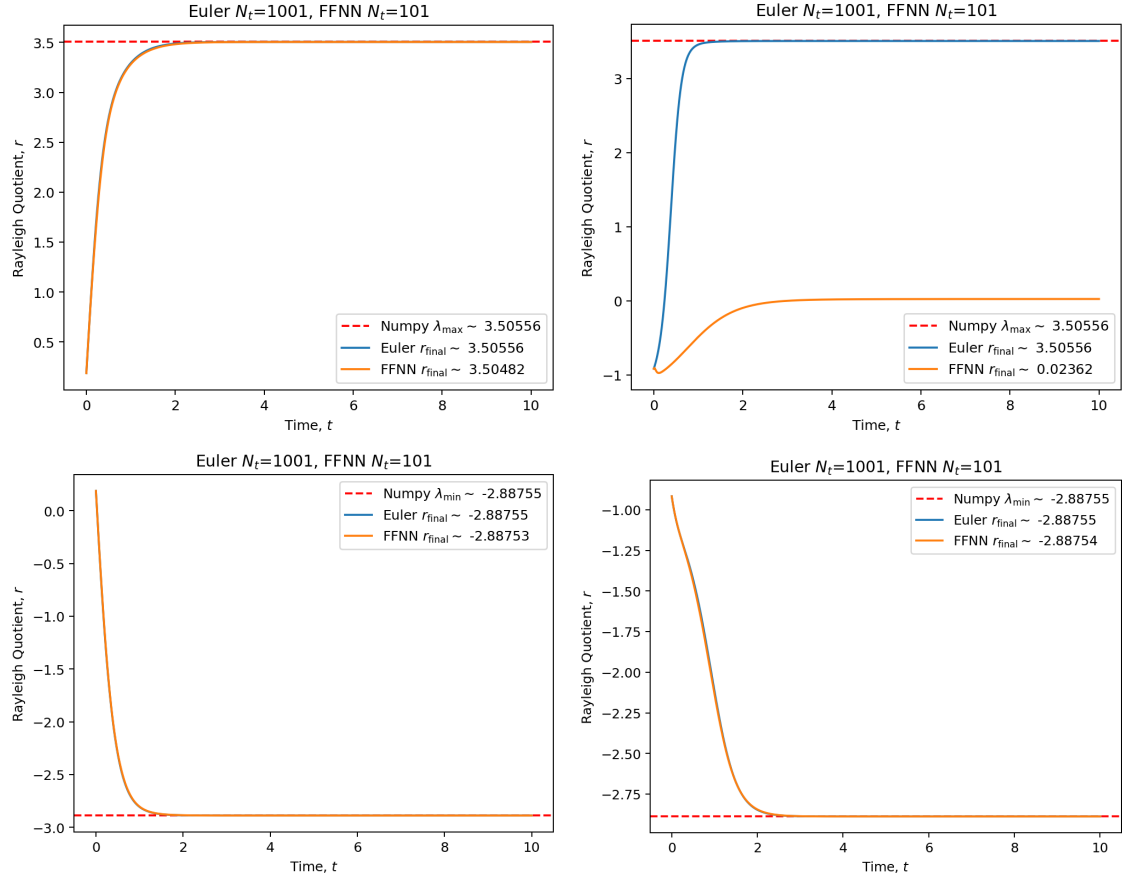


Figure 14. Lhs. max(top) min(bottom) eigenvalues for  $x_0$ , rhs. max(top) min(bottom) eigenvalues for  $x_1$ . NumPy dashed red line, Euler solid blue line, FFNN solid orange line.