

Machine Learning: Bra Tittel



Vetle Nevland, Vetle Vikenes & Sigurd S  yrli   Rustad

FYS-STK4155      Applied Data Analysis and Machine Learning

Autumn 2021

Department of Physics

University of Oslo

December 16, 2021

ABSTRACT: This project investigates two numerical techniques for solving differential equations, forward euler and neural network. Conventional finite difference methods are efficient and accurate for solving not too intricate differential equations. Stability is an issue though, particularly for explicit schemes such as forward euler. Results from numerical simulations suggest that the output of a neural network is able to converge to the true solution. In contrast to forward euler, neural network is more or less unaffected by the time step. Instead, hyperparameters such as learning rate and epochs determine the accuracy and convergence property. Neural networks have their limitation of an extensive learning process, but offer a flexible and robust alternative to finite difference methods that is presumeably independent of temporal discretization. This opens for new possibilities in the field of differential equations. Our results strongly suggest that neural networks are well suited for simulating differential equations over long time periods, being much less sensitive of small time steps which would increase efficiency and accuracy simultaneously.

Contents

1	Introduction	1
2	Theory	2
2.1	The diffusion equation	2
2.2	Analytical solution	2
2.3	Explicit forward Euler	2
3	Solving Differential equations with deep learning	3
3.1	Solving eigenvalue problems	5
4	Method	6
4.1	Neural network setup	6
4.2	Accuracy assessment	7
4.3	Finding eigenvalues	8
5	Results	9
5.1	Finding a suitable Neural network	9
5.2	Error of Forward Euler scheme	12
5.3	Error of Neural Network	12
5.4	Comparison of error	12
5.5	Eigenvalue problem	14
6	Discussion	19
6.1	Solving the diffusion equation with Forward Euler	19
6.2	Solving the diffusion equation with a neural network	19
6.3	Potential for solving general differential equations	22
6.4	Eigenvalue problems	23
7	Conclusion	23
8	Appendix	24

To do

- NN: Skrive metode
- NN: Finne beste NN (Resultater)
- Sammenlikne NN-FE feil og performance
- Konklusjon

1 Introduction

We will in no way answer all questions linked to the aforementioned methods. So that anyone can reproduce or continue our studies, we list all the code, results and instructions on running the code in our GitHub repository¹.

Neural network is a trending machine learning method due to its ability to solve a wide range of problems. By calculating the error at the output layer with an appropriate loss function, the weights and biases are updated accordingly to provide the best possible approximation to the true solution of the problem. The inner workings a neural network is considered a black box, difficult to justify theoretically, but its flexibility and wide applicability makes it a relevant method for approaching more unconventional tasks. This includes solving differential equations, which will be the main purpose for this project.

The benefit of solving differential equations with neural network is that we don't need to know the analytical solution. By starting with a suitable guess of a solution, the neural network will converge towards the analytical solution given an appropriate loss function that initiates gradient descent. Hence, neural networks are flexible enough to solve complex differential equations, including ODEs and PDEs. The focus in this project will be on solving a one dimensional diffusion equation. This is a PDE with a rather intricate analytical solution. It is of interest to investigate the potential for solving this PDE with a neural network.

A natural extension is to consider another differential equation with another applicability. We will consider a first order ODE whose solution is the eigenvector corresponding to the largest eigenvalue of a real, symmetric matrix. Solving this differential equation will reveal the potential for a neural network to find eigenvalues of symmetric matrices. The result will be compared with traditional numerical methods for solving differential equations, particularly the forward euler method. A critical discussion of the methods will then be presented with focus on flexibility and computational efficiency. An overarching question considered is if neural networks can compete with state-of-the-art numerical methods for solving differential equations.

¹<https://github.com/sigurdru/FYS-STK4155/tree/main/project3>

2 Theory

In the theory-section we aim to give a brief explanation of the main concepts and terminology used in this report.

2.1 The diffusion equation

The full diffusion equation reads

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

where \mathbf{r} is a positional vector and $D(u, r)$ the collective diffusion coefficient. If $D(u, \mathbf{r}) = 1$ the equation simplifies to a linear differential equation

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

In this report we are going to study a one dimensional rod of length $L = 1$. I.e. we need the one dimensional diffusion equation

$$\frac{\partial^2 u(x, t)}{\partial x^2} = \frac{\partial u(x, t)}{\partial t}, \quad (2.1)$$

with boundary conditions

$$\begin{aligned} u(x, 0) &= \sin(\pi x) \quad 0 \leq x \leq L, \\ u(0, t) &= 0 \quad t \geq 0 \text{ and} \\ u(L, t) &= 0 \quad t \geq 0. \end{aligned}$$

2.2 Analytical solution

An analytical solution of the 1D diffusion equation can be derived using the method of separation of variables. For a complete derivation, please refer to the Appendix. The analytical solution satisfying the given initial and boundary conditions is

$$u(x, t) = e^{-\pi^2 t} \sin(\pi x) \quad (2.2)$$

2.3 Explicit forward Euler

In this section we want to cover the explicit forward Euler. By explicit we mean that the value at the next grid point is determined entirely by known or previously calculated values.

In this report we are going to study the diffusion equation for a one dimensional rod of length $L = 1$, given in equation (2.1)

To approximate the solution, we have to discretize the position and time coordinates. We can choose $\Delta x = L/N$ and Δt as small steps in x -direction and time, respectively, where N are the number of discretized points in x -direction. Then we can define the value domain of t and x ,

$$t_j = j\Delta t, \quad j \in \mathbb{N}_0 \quad \wedge \quad x_i = i\Delta x, \quad \{i \in \mathbb{N}_0 | i \leq N\}.$$

The algorithm for explicit forward Euler in one dimension (from [?] chapter 10.2.1) reads

$$u_{i,j+1} = \alpha u_{i-1,j} + (1 - 2\alpha)u_{i,j} + \alpha u_{i+1,j} \quad (2.3)$$

where

$$\alpha = \frac{\Delta t}{\Delta x^2}.$$

This has a local approximate error of $O(\Delta t)$ and $O(\Delta x^2)$. Experiments show that the following bound on α ensures stability.

$$\alpha = \frac{\Delta t}{\Delta x^2} \leq \frac{1}{2}. \quad (2.4)$$

Even though $\alpha = 0.5$ is at the transition between stable and unstable solutions, experiments show that it is a safe choice for the diffusion equation that produces stable solutions [?]. Thus, the stability factor $\alpha = 0.5$ is used as default in this project.

3 Solving Differential equations with deep learning

Many of the concepts we use in this section are covered in a previous report [?]. There, in the theory section, we cover central concepts like deep neural networks, cost functions, gradient descent, etc. In this report we will follow closely the work by Maziar Raissi et.al. [?] and this Jupyter-Notebook². This method works with any partial differential equation (PDE), however for simplicity we will look at the diffusion equation in one dimension (2.1).

The idea is that we have some neural network N with parameters θ , including x and t as inputs, which returns $u_\theta(x, t)$. After training the neural network, one obtains the parameters $\hat{\theta}$ that best approximates its output with the actual solution, $u(x, t)$, of the PDE

$$u_{\hat{\theta}}(x, t) \approx u(x, t).$$

The process starts by defining a trial function $u_\theta(x, t)$ which is an initial guess of the actual solution $u(x, y)$, defined as

²https://colab.research.google.com/github/janblechschmidt/PDEsByNNs/blob/main/PINN_Solver.ipynb#scrollTo=1PlqQM9aZEkd

$$u_\theta(x, t) = u_0(x, t) + f(x, N(x, \theta)), \quad (3.1)$$

where u_0 is a function specifically designed to satisfy the initial and boundary conditions of the PDE. The function f is the part that is to be optimized by the algorithm. It depends on x and the parameters θ through the neural network N , and should return 0 for the initial and boundary conditions. Constructing the function f should beneficially incorporate prior knowledge about how the solution behaves, because this will accelerate the learning process.

We then define the residual $r_\theta(t, x)$, which is the output of the neural network inserted back into the PDE (2.1)

$$r_\theta(x, t) \equiv \frac{\partial u_\theta}{\partial t} - \frac{\partial^2 u_\theta}{\partial x^2}. \quad (3.2)$$

Ideally this should be zero, implying a perfect reproduction of the actual solution. The residual becomes our main object for training, as it gives a measure of how good the neural network approximates the actual solution. For the neural network to improve its predictions a loss function is required, which is to be minimized. For this we use the mean sum of squared residuals, including the initial and boundary points, across all data samples n .

$$L(x, t, \theta) = \frac{1}{n} \sum_{i=1}^n [r_\theta(x_i, t_i)]^2 + \frac{1}{n_B} \sum_{i=1}^{n_B} (u_\theta(x_b, t_i) - u(x_b, t_i))^2 + \frac{1}{n_I} \sum_{i=1}^{n_I} (u_\theta(x_i, 0) - u(x_i, 0))^2,$$

where n_B and n_I is the number of training points on the boundary and the initial time step, respectively. Boundary points are given by x_b . Optimization is then done through gradient descent, minimizing the loss with respect to the parameters θ

$$\theta \leftarrow \theta - \eta \nabla_\theta L(x, t, \theta),$$

where η is the learning rate, which can either be constant or variable, depending on the optimization strategy. The loss function converges towards zero for the optimal parameters θ , in which case the gradient of the loss function vanishes. As a result, the parameters will hardly update anymore through gradient descent, indicating that we have reached an optimal solution.

3.1 Solving eigenvalue problems

Deep neural networks are capable of solving more or less any differential equation. We will in this section follow the work of Yi et. al. (2004) [?] closely, using the same first ODE as them and closely following their methods for finding the eigenvalues. The first order ODE in question is given as

$$\frac{dx(t)}{dt} = -x(t) + f[x(t)], \quad (3.3)$$

where

$$f(x) = [x^T x A + (1 - x^T A x) I] x$$

where A is a real symmetric matrix of shape $(n \times n)$ and $t \geq 0$ represents time. The solution of the neural network is given by $x(t)$ with dimensions $(n_t \times n)$, where n_t is the number of time steps. The ODE (3.3) is applicable for finding the eigenvalues of A . An equilibrium point \tilde{x} of a differential equation is a solution that is stationary, that is a state where the solution doesn't change anymore, mathematically expressed as $\frac{dx}{dt} = 0$. It can be shown that the equilibrium points of (3.3) span the eigenspace of A , implying that the equilibrium points are precisely the eigenvectors of A . That is, any \tilde{x} that satisfies

$$\begin{aligned} \frac{d\tilde{x}(t)}{dt} &= -\tilde{x} + f[\tilde{x}(t)] \\ &= -\tilde{x} + [\tilde{x}^T \tilde{x} A + (1 - \tilde{x}^T A \tilde{x}) I] \tilde{x} \\ &= \tilde{x}^T \tilde{x} A \tilde{x} - \tilde{x}^T A \tilde{x} \tilde{x} = 0 \end{aligned}$$

is an eigenvector of A . The rayleigh quotient formula can then be used to find the corresponding eigenvalue, given by

$$r = \frac{\tilde{x}^T \tilde{x}}{\tilde{x}^T A \tilde{x}} \quad (3.4)$$

An essential question is how the neural network should be initialized. That is, what trial solution of the form (3.1) should be chosen? Convergence analysis reveal that any non-zero trial solution $x(0) \in \mathcal{R}^n$ will give a solution of (3.3) that converges to an eigenvector of A . Moreover, if $x(0)$ is not orthogonal to the subspace spanned by the largest eigenvector λ_{\max} of A , then the solution $x(t)$ converges to an eigenvector corresponding to λ_{\max} .

The differential equation (3.3) is a first order nonlinear ODE. Generally, solving a nonlinear differential equation numerically is a comprehensive process involving discretization in time followed by linearization through an iterative method (e.g Newton's method). Explicit schemes are an exception though, as all nonlinear terms are of the previous time step, hence are known. This allow us to easily solve (3.3) numerically using the forward euler method. Discretizing the equation and solving for next time step $n + 1$ gives

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

Starting with a non-zero initial condition $x^0 = I(x)$, iteration over time steps will make the subsequent solutions converge towards the eigenvector corresponding to the largest eigenvalue of A , as long as the forward euler scheme is stable. The stability criteria depends on the discretization parameter Δt , and is to be approximated empirically.

4 Method

Unit testing

Before the numerical explicit scheme is applied to a particular problem, it is important to test that the discretized equations return expected results. Unit tests are constructed to test the implementation. This is done by manually calculating the solution of the first two time steps given the initial condition, and comparing the result with that obtained by the numerical scheme. Since the same recursive formula is used for all time steps, it is sufficient to test the two first time steps. To avoid too much computations we choose five equally sized intervals between $x = 0$ and $x = L = 1$, that is $\Delta x = 0.2$. The time step is set to $\Delta t = 0.01$, ensuring stability. For time step j , that is $t = j\Delta t$, we have the following two boundary values and four interior points

$$\begin{aligned}
u_0^1 &= u_5^1 = 0 \\
u_i^1 &= \frac{\Delta t}{\Delta x^2} u_{i+1}^0 + \left(1 - 2\frac{\Delta t}{\Delta x^2}\right) u_i^0 + \frac{\Delta t}{\Delta x^2} u_{i-1}^0 \\
&= 0.05 u_{i+1}^0 + 0.9 u_i^0 + 0.05 u_{i-1}^0
\end{aligned}$$

The initial condition used is $u_*^0 = (\sin(0), \sin(0.2\pi), \sin(0.4\pi), \sin(0.6\pi), \sin(0.8\pi), \sin(\pi))$. If the results of the code match those manually calculated up to machine precision, it verifies the internal functionality of our implementation of forward euler. The results of the unit test are provided in Table 2 in Appendix.

4.1 Neural network setup

Sigurd: Skriv hvordan du setter opp neural network her. Skriv ogs   hva du kommer til    teste (dybde) og hva du ikke skal teste.

Vetle og Vetle tenker at det er fint Å bruke loss for Å finne beste NN, og nÅr det er funnet sÅ regner vi ut MSE o.l. med metoden under for sammenlikning.

4.2 Accuracy assessment

The unit test shows that the forward euler scheme works as expected, but to what accuracy can it reproduce the analytical solution and how does it compare to the neural network? To address these questions we will present some concrete methods for studying the solvers.

As discussed in the theory section, following equation (2.4), $\alpha = 0.5$ produce stable solutions for the diffusion equation. Throughout this report we will therefore use a time step of $\Delta t = \alpha \Delta x^2 = 0.5 \cdot \Delta x^2$ for different choices of spatial resolution, and not consider other stability factors. We will limit our study to two spatial resolutions, namely $\Delta x = 0.1$ and $\Delta x = 0.01$.

The first analysis we perform is the time evolution of the mean squared error (MSE) of the forward Euler solution. The MSE at a given time step, j , is given as

$$\text{MSE}_j = \frac{1}{N} \sum_{i=0}^N (u_i^j - u_{e,i}^j)^2 \quad (4.1)$$

where u_e denotes the analytical solution. We expect the MSE to be zero initially, as numerical solution for the first time step is simply the initial condition. Proceeding, we expect an increasing MSE up to a certain point. Our boundary conditions results in the diffusion equation approaching $u(x, t) = 0$ as t increases, causing the MSE to eventually decline after a certain time. Combined with the unit tests, this serves as a simple method of validating our result. Skal vi plotte dette for NN ogsÅ?

To proceed, we will study the solutions at two different times, t_1 and t_2 for both $\Delta x = 0.1$ and $\Delta x = 0.01$. We choose $t_1 = 0.1$ and $t_2 = 0.5$, where $u(x, t_1)$ is significantly curved and $u(x, t_2)$ is almost linear. These times will serve as the foundation for comparing the performance of our explicit numerical solver to that of the neural network. With the forementioned time periods and spatial resolutions, we compare the MSE of the two models, using equation (4.1).

As a final study, we would like to plot the absolute difference between the numerical solutions and the analytical one for both t_1 and t_2 .

$$\Delta u_i^n = u_i^n - u_{e,i}^n \quad (4.2)$$

Our main purpose of this is to see whether the neural network can circumvent one of the major drawbacks of the forward Euler scheme, namely the error caused by the discrete time steps advancing the solution more than it should. Solving the diffusion equation with forward Euler, we expect equation (4.2) to yield a negatively curved plot, with the boundaries being zero and the largest error occuring at $x = L/2$. This is due to the discretized solver advancing the solution by too much at each time step, affecting the

succeeding calculations throughout the simulation. The biased error of the forward Euler scheme will cause big problems for more complex differential equations, where solutions don't converge to a stationary state. One major advantage of using neural networks will thus be if it yields unbiased errors that doesn't propagate throughout the simulation. Kanskje litt dÅrlig forklart over, sÅ skriv gjerne om

4.3 Finding eigenvalues

The matrix we will consider for our initial analysis is a real symmetric matrix, A , with dimensions (3×3) ³. In the end we will perform one simulation with a (6×6) matrix to study differences for higher dimensions.

Using the neural network model (3.3) to find the eigenvalues of A requires a trial solution (3.1) to initiate the network. The trial solution must fulfill the convergence property to guarantee that we end up with the eigenvalues. Hence, we define the initial condition $x_0 \in \mathcal{R}^n$ to be a vector of randomly generated numbers between zero and one. Since we don't know the eigenspace corresponding to the largest eigenvalue a priori, we have no guarantee that the trial solution is not orthogonal to the eigenspace corresponding to the largest eigenvalue. The probability of non-orthogonality, though, can be increased by adding random perturbations.

Even though a non-zero trial solution is the only requirement for convergence, there are strategies for improving the *rate* of convergence. If we have some idea or preknowledge of a solution of the differential equation, incorporating this in the trial solution helps accelerate the learning of the neural network. The ODE (3.3) is of first order with a term involving $\frac{dx}{dt}$ on one side and a term involving x on the other. This is reminiscent of an exponential solution. Therefore, a smart initialization strategy of (3.1) is

$$\begin{aligned} g_\theta(t) &= g_0(x(t)) + f[x(t), N(x, \theta)] \\ &= e^{-t}x_0 + (1 - e^{-t})N(x(t), \theta) \end{aligned}$$

This expression fulfills the condition of a trial solution, namely that it must return the initial condition x_0 for time $t = 0$. Once we have a trial solution we feed it to the neural network, which will calculate the gradient $\frac{dx}{dt}$ and the resulting residual of (3.3). The MSE of the residual act as the loss function that is to be optimized through gradient descent. The accuracy of the final output of the neural network is assessed by comparing with the result of numpy's own eigenvalue solver.

For comparison purposes, the forward euler scheme is initiated with the same initial condition x_0 as for the neural network model, and using the same temporal domain. A total time of $T = 5$ is consistently used for the simulations, as this seems to be sufficient for

³We didn't realize that it should have been a (6×6) matrix until after the report was finished. This has been cleared with Morten

convergence and at the same time short enough to illustrate the details of the convergence. Unless specified, the learning rate and the number of epochs of the neural network is set to 0.005 and 2000, respectively.

For each simulation we perform, we find the eigenvector, $\mathbf{v}_{\max} = (v_1, v_2, v_3)$, with the largest corresponding eigenvalue, λ_{\max} . We then plot the computed value of the three eigenvector components, v_1 , v_2 and v_3 as a function of time, for both Forward Euler and Neural network, and include the corresponding value obtained from numpy for comparison. For a given simulation we also plot the time evolution of the Rayleigh quotient, r from equation (3.4), for the two forementioned methods, using numpy once again for comparison.

5 Results

5.1 Finding a suitable Neural network

Figure 1 shows the data points used for training the network. One vertical line represents the spatial data points at one point in time.

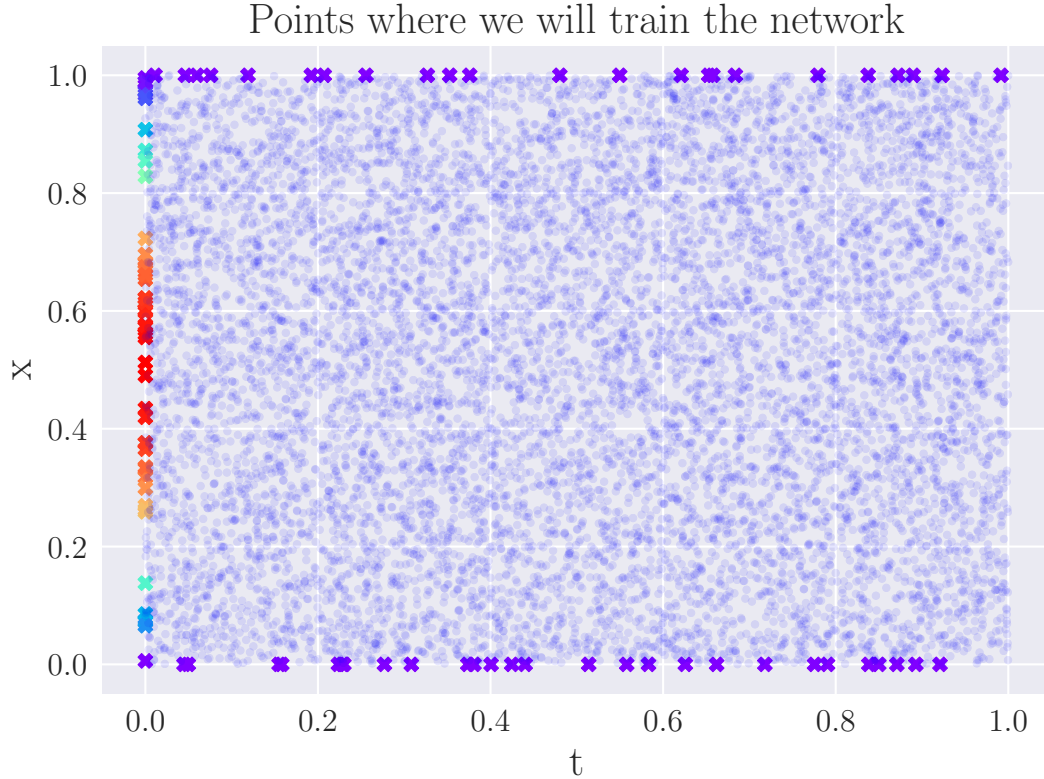


Figure 1. In this plot we highlight the data points used for training the neural network. Points on the boundary are highlighted, and we use 50 points for $t = 0$, and a total of 50 points at $x = 0$ and $x = 1$, resulting in a total of 100 boundary points.

The loss of the network as a function of iterations is shown in figure 2 for three different architectures. Each network has 20 nodes in each hidden layer, where the upper left panel

has 2 hidden layers, the upper right panel has 4 hidden layers and the lower left panel has 8 hidden layers.

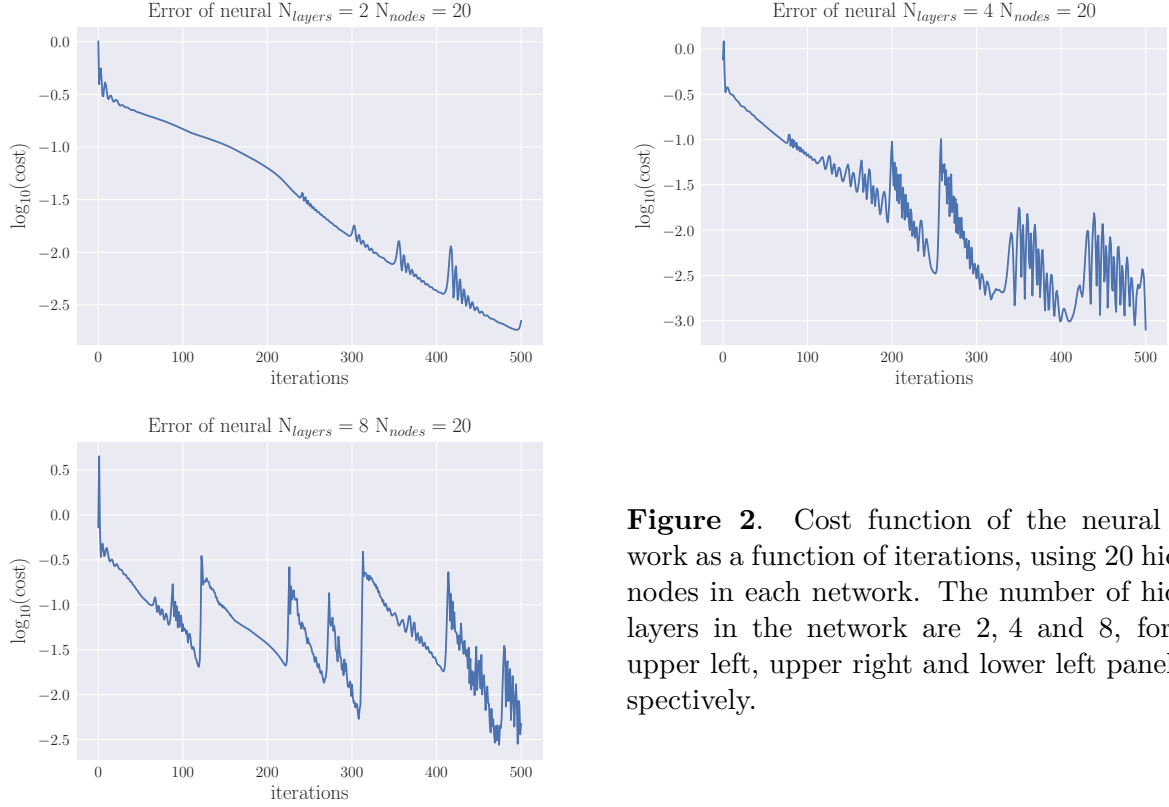


Figure 2. Cost function of the neural network as a function of iterations, using 20 hidden nodes in each network. The number of hidden layers in the network are 2, 4 and 8, for the upper left, upper right and lower left panel, respectively.

Although there are large fluctuations of the loss for the network with 4 hidden layers, we choose this architecture for further analysis. Using logarithmic scales on the y-axes in figure 2, the amplitudes of the oscillating losses appear bigger than they are in reality. We only consider the network after the final iteration, and this architecture provided the lowest loss and MSE. A detailed study of different architectures is outside the scope of this report, and is nonetheless dependent of the particular PDE in question. Unless specified, the following results are all obtained with a neural network with 4 hidden layers after 500 iterations.

Figure 3 shows the output of the neural network. We clearly see the exponential decay of the initial sine function, as expected, eventually flattening out, as $t \rightarrow 1$.

Architecture	Loss
8×20	0.00476
4×20	0.00079
2×20	0.00224

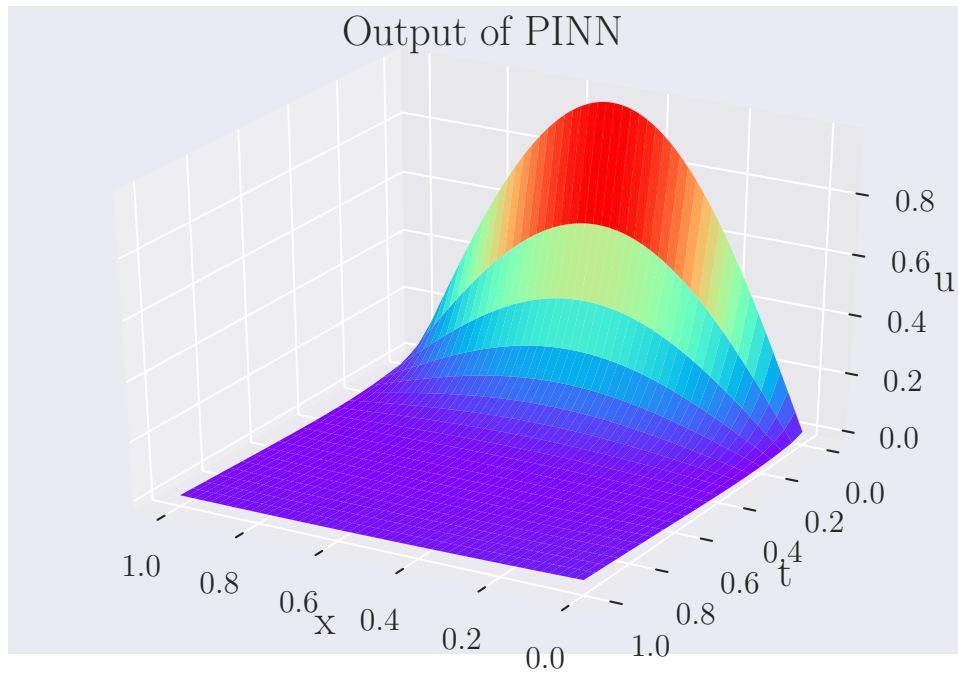


Figure 3. The solution of the diffusion equation of the final neural network.

5.2 Error of Forward Euler scheme

To assess the accuracy of our implemented numerical scheme, we need to test how well the produced solution fits the analytical solution. In this project we use the mean squared error to compare the approximated solutions with the analytical solution, both for forward euler and the neural network. Figure 4 shows the MSE as function of time for the forward euler scheme. The time step is chosen to give a stability factor $\alpha = 0.5$. The error increases significantly for early time steps, reaches a peak at around $t = 0.1$ and quickly reduces thereafter. Already at $t = 0.5$ the MSE is converging towards zero.

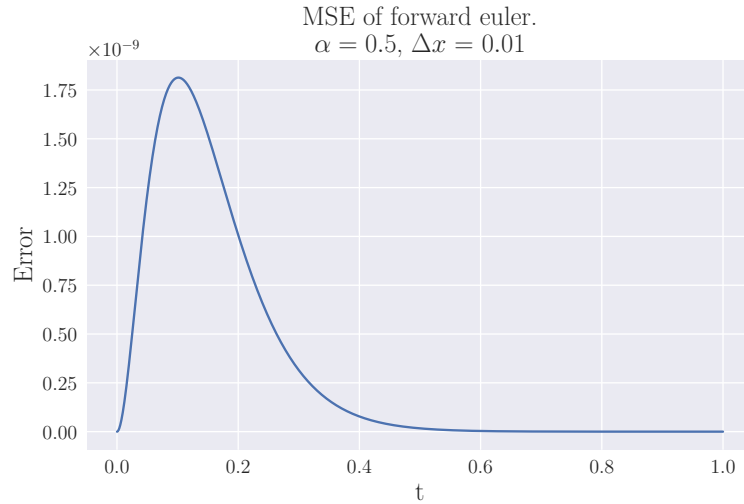


Figure 4. Mean squared error of approximated solution by forward euler, using $\Delta x = 0.01$ and time step Δt dictated by the stability criterion (2.4).

Figure ?? shows the MSE from the neural network with the architecture that provided best results. **Sigurd: Legg til MSE plottet her for det beste nettverket.**

5.3 Error of Neural Network

Basert paa gammel figur. **Sigurd: kanskje dette maa skrives om, avhengig av de endelige resultatene dine.** Figure ?? shows the error of the neural network when approximating the solution u_θ to the actual solution u as a function of training iterations. Initially, the error decreases considerably - it reduces by a factor of ten for just a couple of iterations, and the approximated solution improves significantly. Afterwards, the error decreases by a much slower rate with some minor fluctuations. Some time after 100 iterations the error is subject to a large jump, but quickly diminishes again. For later iterations the error slowly converges towards zero, possessing some irregular, minor fluctuations.

5.4 Comparison of error

Table 1 compares the MSE obtained for forward euler and neural network at two selected time steps t and two different spatial steps Δx .

Figure 5. The difference between the analytical solution of the diffusion equation and the solution obtained by forward euler at two different points in time.

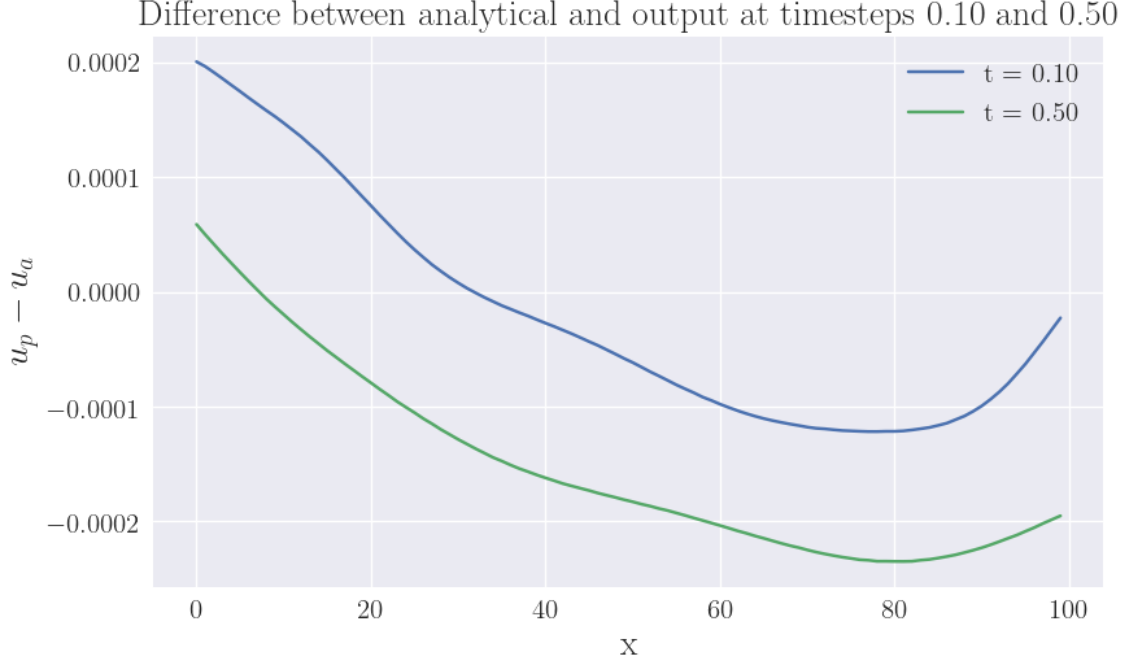


Figure 6. In this plot ...

Number of training iterations	Mean Square error	
	t = 0.1	t = 0.5
500	0.0017	0.0070
1000	0.015	0.00055
10^4	$1.4 \cdot 10^{-4}$	$9.5 \cdot 10^{-5}$
$5 \cdot 10^4$	$1.0 \cdot 10^{-6}$	$3.0 \cdot 10^{-6}$

Figure 5 shows the plot of equation (4.2), i.e. the numerical solution of the diffusion equation from forward Euler, subtracted by the analytical solution in equation (2.2). The time points shown in the plot are $t_1 = 0.1$ and $t_2 = 0.5$. The same plot when using $\Delta x = 0.1$ has identical shape, and differs only by larger error values on the y-axis, and we have therefore chosen to omit it.

Using equation

	$\Delta x = 0.1$		$\Delta x = 0.01$	
	FE	NN	FE	NN
$t_1 = 0.1$	$1.72 \cdot 10^{-5}$		$1.81 \cdot 10^{-9}$	
$t_2 = 0.5$	$1.50 \cdot 10^{-7}$		$1.69 \cdot 10^{-11}$	

Table 1. MSE as function of spatial step Δx for two different time levels, for forward euler scheme and neural network. Forward euler is abbreviated as FE and neural network as NN.

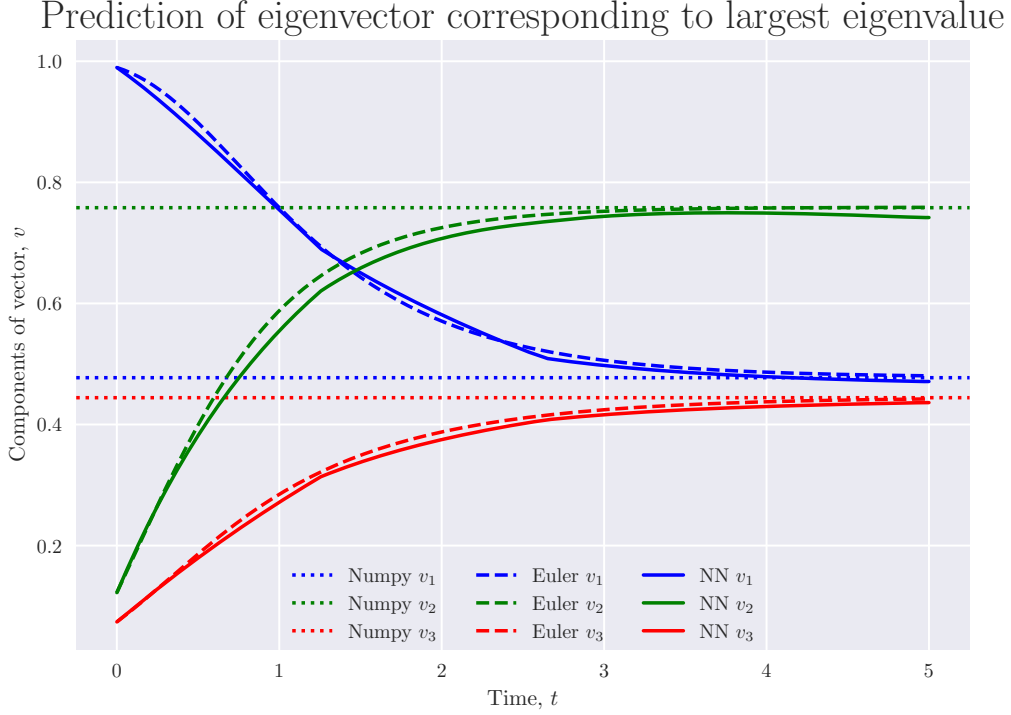


Figure 7. Predictions by forward euler (FE) and neural network (NN) of the eigenvector corresponding to the largest eigenvalue of a real, symmetric 3×3 matrix. The horizontal dotted lines are the components of the eigenvector as found by numerical diagonalization by `numpy.linalg.eig`.

5.5 Eigenvalue problem

Figure 7 shows predictions of the eigenvector corresponding to the largest eigenvalue of a symmetric (3×3) matrix A of randomly assigned values between 0 and 1. The simulation is run for a total time of $T = 5$ with $N = 1000$ estimations in time, providing a time step of $\Delta t = 0.005$. The prediction from both forward euler and neural network converges to the true eigenvector, as found by diagonalization with `numpy.linalg.eig`. The largest eigenvalue is shown in Figure 8 together with the rayleigh quotients estimated by forward euler and the neural network.

Figure 9a and 10a shows predictions of the same eigenvector of the same matrix A in Figure 7 but with a time step of $\Delta t = 0.05$ and $\Delta t = 0.5$, respectively. The largest eigenvalue and rayleigh quotients are shown in Figure 9b and 10b. The graphs are less smooth due to larger timesteps and fewer estimates.

Adjusting hyperparameters of the neural network has also been investigated. Figure 11a shows the predicted eigenvector using a time step of $\Delta t = 0.005$ and a learning rate of 0.1. The associated rayleigh quotients is shown in Figure 11b. Predictions of the same eigenvector is shown in Figure 12a, with the learning rate retained at 0.1 but the number of epochs reduced from 2000 to 50. The associated rayleigh quotients are shown in Figure 12b.

Results of a further increase in the time step is shown in Figure 13, using $\Delta t = 0.83$

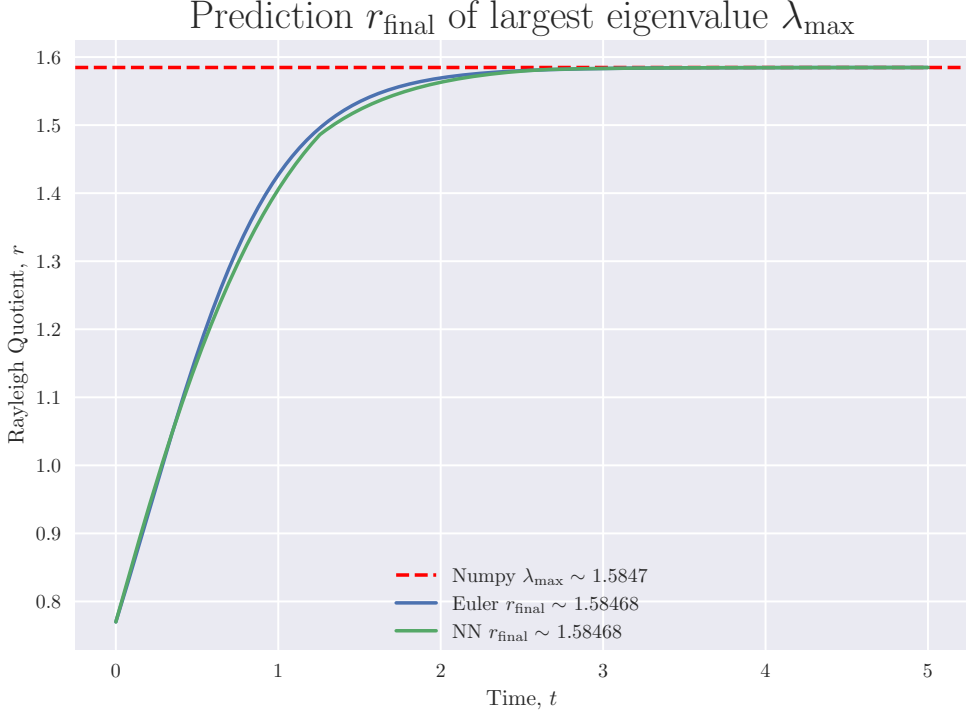


Figure 8. Rayleigh quotients of forward euler (FE) and neural network (NN) representing predictions of the largest eigenvalue of a real, symmetric 3×3 matrix. The horizontal dotted line is the largest eigenvalue as found by numerical diagonalization by `numpy.linalg.eig`.

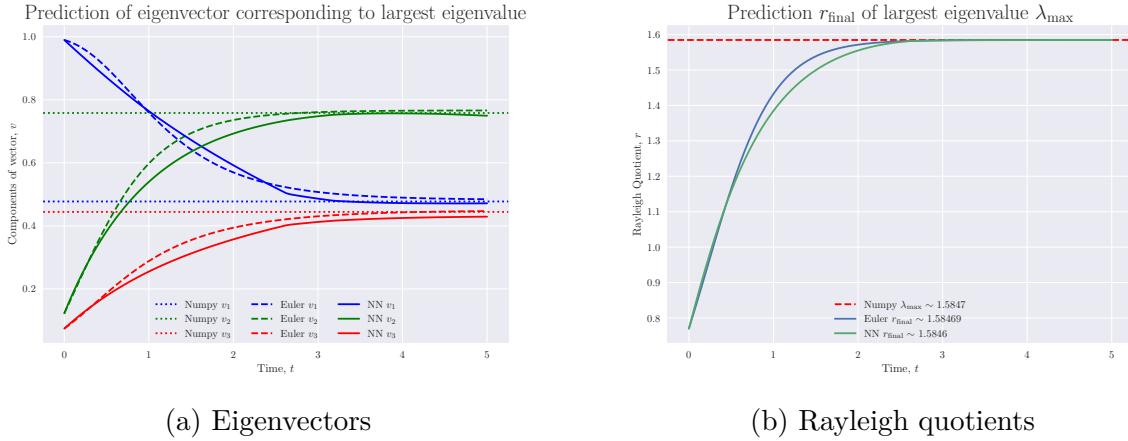
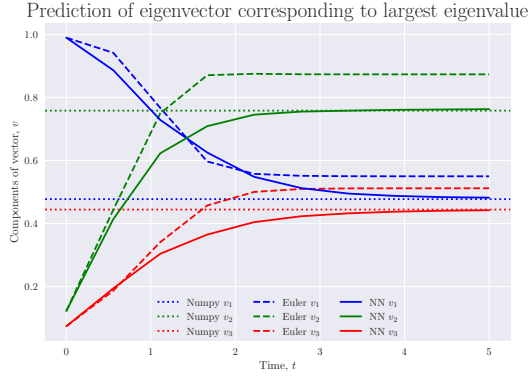


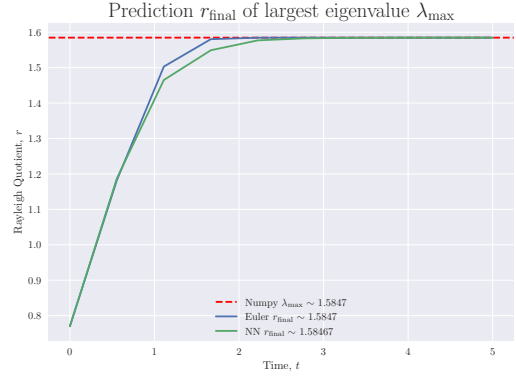
Figure 9. Predictions by forward euler (FE) and neural network (NN) of largest eigenvalue and corresponding eigenvector of a real, symmetric 3×3 matrix, using a time step $\Delta t = 0.05$. The x and y axes are identical to those in figure 7 and 8

corresponding to $N = 6$ estimations, where the left panel shows the eigenvector components and the right panel shows the associated Rayleigh quotients.

An additional simulation using a 6×6 matrix has also been conducted. Figure 14 shows the predicted eigenvector for forward euler (14a) and neural network (14b), and Figure ?? the rayleigh coefficient corresponding to the largest eigenvalue of a 6×6 symmetric matrix. A time step of 0.005 have been used and a total time of $T = 2$. The reason for the latter choice is based on the previous simulations with $\Delta t = 0.005$, as both forward euler and

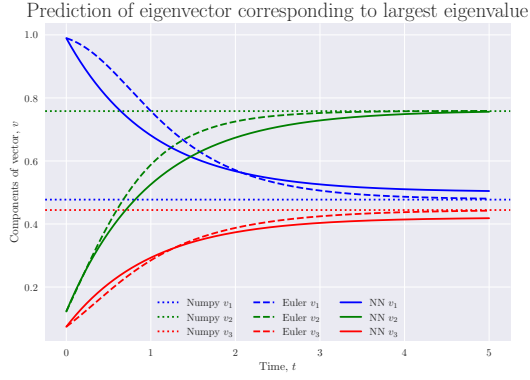


(a) Eigenvectors

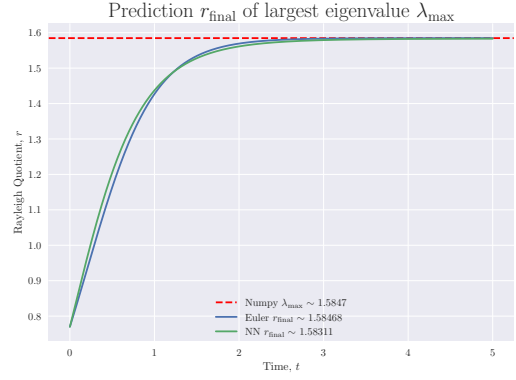


(b) Rayleigh quotients

Figure 10. Predictions by forward euler (FE) and neural network (NN) of largest eigenvalue and corresponding eigenvector of a real, symmetric 3×3 matrix, using a time step $\Delta t = 0.5$. The x and y axes are identical to those in figure 7 and 8

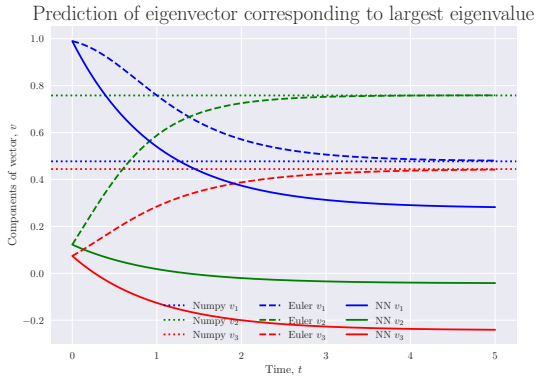


(a) Eigenvectors

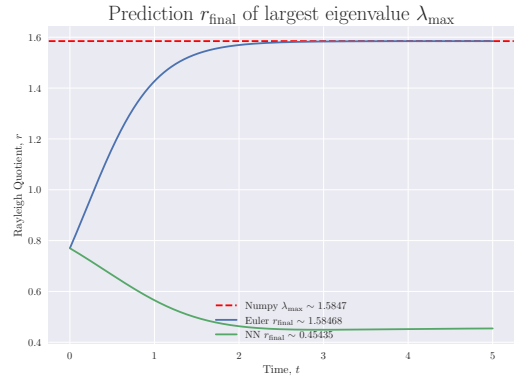


(b) Rayleigh quotients

Figure 11. Predictions of largest eigenvalue and corresponding eigenvector by forward euler and neural network for a real, symmetric 3×3 matrix, using a learning rate of 0.1 and a time step $\Delta t = 0.005$. The x and y axes are identical to those in figure 7 and 8

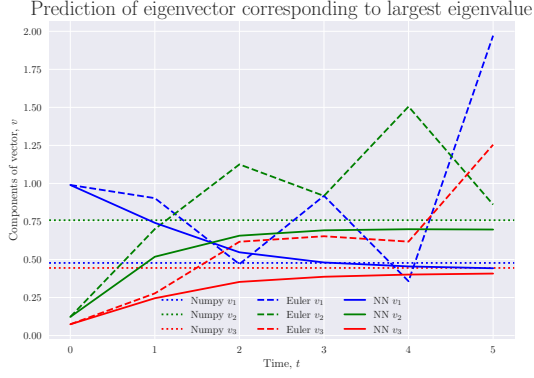


(a) Eigenvectors

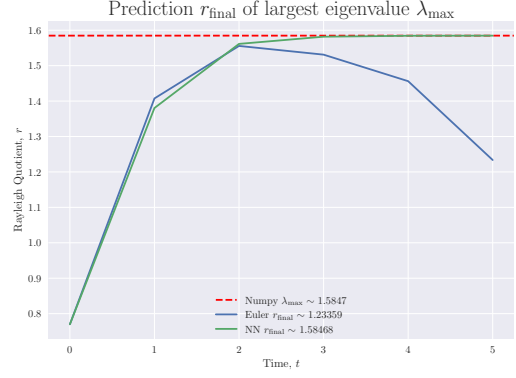


(b) Rayleigh quotients

Figure 12. Predictions of largest eigenvalue and corresponding eigenvector by forward euler and neural network for a real, symmetric 3×3 matrix, using a learning rate of 0.1, 50 epochs and a time step $\Delta t = 0.005$. Other than negative eigenvector components in the left panel, and lower Rayleigh quotient in the right panel, the x and y axes are identical to those in figure 7 and 8

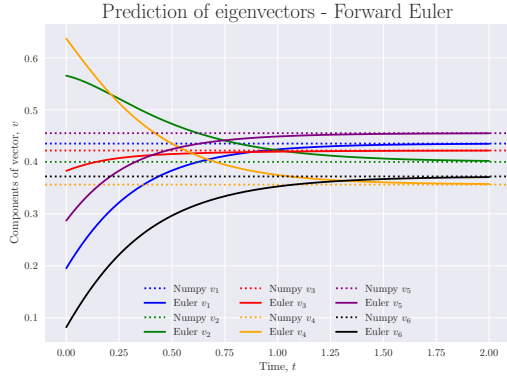


(a) Eigenvectors

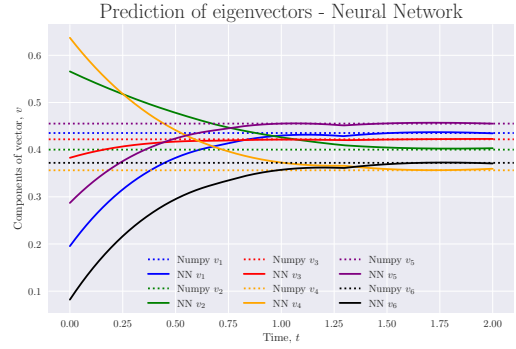


(b) Rayleigh quotients

Figure 13. Predictions of largest eigenvalue and corresponding eigenvector by forward euler and neural network for a real, symmetric 3×3 matrix. A time step of $\Delta t = 0.83$ is used. Other than increased eigenvector component values for forward Euler on the left panel, the x and y axes are identical to those in figure 7 and 8



(a) Forward Euler



(b) Neural network

Figure 14. Predictions of eigenvector corresponding to the largest eigenvalue for a real, symmetric 6×6 matrix. A time step of $\Delta t = 0.005$ and a total time $T = 2$ is used.

neural network converge long before $T = 5$. Reducing the simulation time makes it easier to illustrate the convergence property of the six eigenvector components.

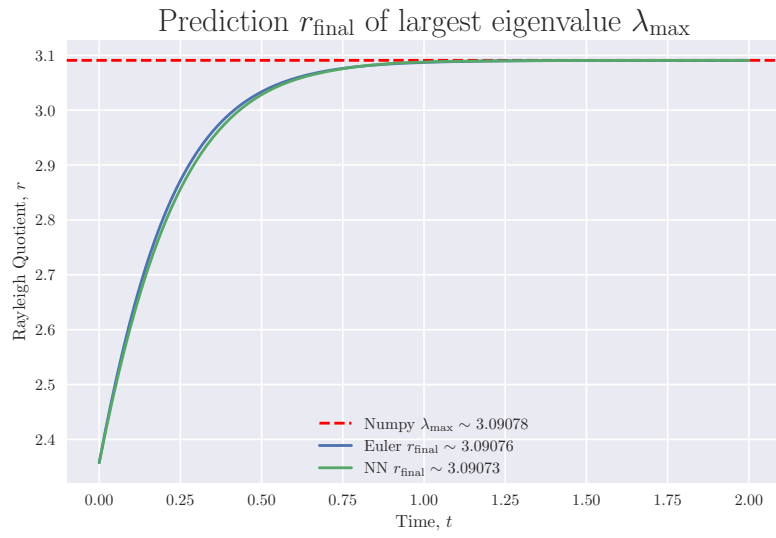


Figure 15. Rayleigh quotient corresponding to the largest eigenvalue of a real, symmetric 6×6 matrix.

6 Discussion

6.1 Solving the diffusion equation with Forward Euler

Figure 4 verifies that simulating the forward euler scheme of the diffusion equation with spatial step $\Delta x = 0.01$ yields an excellent correspondance between the numerical and analytical solution. The maximum error is obtained at an early time step, around $\Delta t = 0.1$. For time levels beyond this, the MSE decreases exponentially and eventually converges towards zero. Notice that the error is exactly zero initially, because the numerical scheme has enforced the solution at the initial time step to satisfy the initial condition from the analytical solution. The maximum error is on the order of 10^{-9} , yielding an excellent fit to the exact solution. One would not be able to distinguish the numerical solution from the exact solution if they were plotted together.

The shape of the plot is in good correspondance with what was initially expected, as elaborated on in Methods. The large initial increase of the MSE is a result of the curved initial solution. The spatial gradients are significant and the approximation with finite differences becomes less accurate. In the beginning of the diffusion process, for low t , the solution changes rather quickly. The forward euler scheme is first order accurate in time, $O(\Delta t)$, and second order accurate in space, $O(\Delta x^2)$, implying that the numerical solution is more sensitive to the temporal evolution than the spatial evolution. Hence, the forward euler scheme will produce larger errors for the initial time levels due to the rapid change in solution [?].

After time $t = 0.1$ the MSE decreases. This can be explained by the discretization parameters Δt and Δx . The magnitude of the gradients are diminished because the solution becomes more linear and less curved. As a result, the solution changes more slowly, allowing the finite difference scheme to approximate the changes more accurately both spatially and temporally. Eventually, the exact solution is approximately constant at zero. In this case, the gradients of the solution are indiscernible and the numerical scheme is able to approximate the solution with machine precision. Figure 4 illustrates this by the MSE converging towards zero sufficiently deep into the diffusion process.

6.2 Solving the diffusion equation with a neural network

Et forsÅyk paa aa forklare fluktuasjonene i MSE for NN: The reason that the error fluctuates instead of decreasing monotonically is that we use the Adam optimizer [<https://arxiv.org/abs/1412.6980>]. This is a type of stochastic optimization method, providing more favorable features than traditional stochastic methods. The stochasticity allows the neural network to explore more of the convex parameter space by trying out new, potentially worse solutions. The hope is that, after escaping the local optima, that we find an even better solution at some later stage. The apparent jump in Figure ?? is an example of escaping a local optima at the expense of further exploration. In this case, it does not seem like the new solutions lead

to better performance than the previous optima. It eventually converges to zero, which is something the solutions prior to the jump did as well.

Using Forward Euler and Neural Network to find eigenvalues

Forward euler and neural networks are two different numerical methods capable of finding the eigenvalues of a real, symmetric matrix A by solving the ODE in equation (3.3). The two algorithms for solving differential equations are quite different, and will therefore yield different predictions of the largest eigenvalue. The result of forward euler is sensitive to the time step Δt . Thus, it is of interest to compare the accuracy of prediction of the two methods when the time step is altered.

Comparing Figure 7, Figure 9a and Figure 10a for time steps $\Delta t = 0.005$, $\Delta t = 0.05$ and $\Delta t = 0.5$, respectively, we observe that the accuracy of forward euler consistently decreases for larger time steps. The result for $\Delta t = 0.5$ is particularly biased with all components of the predicted eigenvector consistently missing their respective targets. The corresponding rayleigh quotients, shown in Figure 8, Figure 9b and Figure 10b, converge precisely to the largest eigenvalue calculated by `numpy.linalg.eig`. Thus, despite the lower accuracy - particularly for forward euler - both methods succeed in returning the largest eigenvalue with four decimal precision.

It may appear surprising that the significant bias of the estimated eigenvector for forward euler for $\Delta t = 0.5$ is not replicated in the associated rayleigh quotient, which converges precisely to the largest eigenvalue. To explain this, observe that the *relation* between the components of the estimated eigenvector seems to replicate that of the true eigenvector, only shifted towards higher values. If v is the eigenvector of matrix A , the associated eigenvalue is given by

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

Then, if s a constant representing the shift, the rayleigh quotient becomes

$$\begin{aligned} r &= \frac{(sv)^T A (sv)}{(sv)^T (sv)} = \frac{s^2 v^T A v}{s^2 v^T v} \\ &= \frac{v^T A v}{v^T v} \end{aligned}$$

This implies that the associated eigenvalue is unaffected by scaling the components of an eigenvector by a constant amount. Geometrically, it means that the eigenvector is extended or shrunk, but the spanned eigenspace is the same, hence the same eigenvalue. **Gir dette mening, Vikenes, Sigurd? Vikenes: Ja.** Figure 11 is an exception, though. For this particular simulation, the relation between the first and third predicted eigenvectors are different from the relation of the true eigenvectors, whereas the rayleigh quotient converges to the largest eigenvalue. This does not comply with the argument above, and we find it difficult to interpret this particular result. It should incentivate for further analysis.

The neural network on the other hand, apparently looks unaffected by the increase in time step. In fact, it seems like the neural network has a more accurate convergence for $\Delta t = 0.5$ than $\Delta t = 0.05$ and $\Delta t = 0.005$. Still, this does not imply that the accuracy increases with larger time step in general, but it substantiates the fact that the convergence property of a neural network is more sensitive to other features than the time step.

In project 2 we studied some of the most sensitive parameters for a neural network to provide accurate predictions, including the learning rate, number of epochs and loss function. The dependence on the learning rate is illustrated by comparing Figure 7 with 11a. Increasing the learning rate from 0.005 to 0.1 has consequences for the result. The neural network now predicts an eigenvector whose components notably deviate from that of the true eigenvector. From Figure 11b the rayleigh quotients corresponds quite well with the largest eigenvalue, just slightly less accurate. Additionally, if the number of epochs is reduced to 50, the consequences are even more severe, as shown in Figure 12a. Now the predicted components completely misses the true components of the eigenvector. Figure 12b shows that the rayleigh quotients of the neural network apparently converges to a different value than the largest eigenvalue. Because forward euler only depends on Δt , it is unaffected by a change in the learning rate and number of epochs. Conclusively, the results indicate that a learning rate of 0.1 and 50 number of epochs is not a good architecture of a neural network for solving equation (3.3).

Figure 13a shows that the eigenvector components do not converge but rather oscillate in an inconsistent pattern when we use Forward Euler with a time step $\Delta t = 0.83$. There is a strong indication of diverging values for the eigenvector components, which would be more apparent if we either increased the duration of the simulation, or decreased the time step even further.

The corresponding rayleigh quotients, shown in Figure 13b, verify that forward euler is not able to converge to the true solution of equation (3.3) as Δt is too large. All simulations with $\Delta t \leq 0.5$ have given a relative error less than 1% between the final rayleigh quotient and the largest eigenvalue using forward euler. In comparison, the relative error for $\Delta t = 0.83$ is 22.1%.

All simulations so far have been conducted for a symmetric 3×3 matrix. It is of interest to compare the two methods for a higher dimensional matrix as well. This is an important step towards generalization of the ability of forward euler and neural network to find eigenvalues of symmetric matrices of arbitrary shape. Figure 14 and Figure ?? simulates for a 6×6 symmetric matrix. The plot shows that neither forward euler nor neural network have problems with convergence. Apparently, the predicted components of the eigenvector are near perfect for both methods after time $T = 2$. The stochastic initialization of the neural network will yield different results for different simulations, some that may give notable deviations. Nonetheless, the rayleigh quotient converges to the largest eigenvalue with forth decimal precision for both neural network and forward euler. The only noticable difference is that forward euler has a smoother convergence, while the neural network seems to possess some minor low-frequency fluctuations. The simulation illustrates that

forward euler is slightly more consistent for small time steps Δt , even for higher dimensional matrices.

Taking all simulations into account, the neural network has excellent accuracy for $\Delta t < 1$. This indicates weak dependency on the time step, but this is completely deafed by the sensitivity to other hyperparameters such as learning rate and number of epochs.

6.3 Potential for solving general differential equations

The forward euler method and neural networks are two quite different methods for solving differential equations. The former relies on discretization of the derivative to arrive at an explicit recursive set of equations to solve for each time step. The latter uses the residual error from the approximation as a basis for backpropagation to update weights and biases to improve the approximation to the actual solution. Figure ?? visualizes how bad the approximated solution of the initial run of the neural network is. Just after a few iterations though, the error decreases tremendously and goes below 0.083 (which is the total error obtained from forward euler) after around 100 iterations. After this, the total error fluctuates mildly but eventually converges towards zero.

The forward euler method clearly has an advantage in its simple recursive formula and fast computation time. The proposed solution is calculated within a few milliseconds, with a total error of only 0.083. On the other hand, it is limited by a stability requirement in order to produce realistic results. This effectively puts a restriction on applicable mesh resolutions.

A notable drawback of the neural network is the demanding training process. To compute an approximated solution of (2.1) requires calculating gradients and accumulating chained derivatives backwards through the hidden layers. This is a significantly larger computational effort than calculating the simple recursive formula of the forward euler scheme. As a result, the neural network suffers from a large computational runtime. Still, if training sufficiently long, the neural network will produce a better solution than forward euler and eventually yield a near perfect approximation to the exact solution.

The forward euler scheme has proven to obtain good accuracy on the numerical solution. However, forward euler has an inherent disadvantage in that it is conditionally unstable with stability criteria given by (2.4). In general it means that the forward euler scheme is not a stable method for solving differential equations as it requires that we carefully assign the mesh discretization steps. Neural networks are not limited to any kind of discretization. Neural networks benefit from good convergence properties. That is, given enough time to train it is able to approximate the actual solution of a differential equation with an error of approximately zero. Moreover, the neural network model (3.1) has the additional flexibility in that the function f can be tweaked to give a better initial guess of the solution.

Overall, the method of choice for solving differential equations is a tradeoff between the accuracy of the approximated solution and the computational runtime. Forward euler is

definitely the desired method if we want a quick representation of the solution. If the importance is how precise the representation is, a neural network is a better choice.

Det meste av diskusjonen over kan beholdes da det gjelder generelt, men de faktiske tallene nevnt maa oppdateres med resultatene fra MSE.

6.4 Eigenvalue problems

The potential of solving differential equations spans a large domain of mathematical problems. Particularly, the model (3.3) can be represented by a neural network or a forward euler scheme to approximate the eigenvalues of a real, symmetric matrix. Our experiments have shown that the neural network has no trouble finding an accurate approximation to the largest eigenvalue and the corresponding eigenvector given a suitable trial solution and network architecture. The parameter space of a neural network is large, and optimizing a solution requires an infeasible search over all parameters. Training for more epochs gives a more accurate prediction, but comes at the expense of a longer runtime, in which case forward euler can produce equally accurate results for a much shorter runtime. Nonetheless, the optionality to tune the various parameters of a neural network makes it a flexible method for finding approximate solutions.

The forward euler scheme gives a more accurate prediction then the neural network, in addition to converging faster, as long as the time step is sufficiently low. In this case, forward euler is the prevailing method for finding the largest eigenvalue of a real, symmetric matrix. However, the method has an inherent disadvantage in terms of stability. Our experiments indicate that forward euler diverges for $\Delta t \geq 1$. This marks a transition where the results of forward euler are deceptive and the neural network is the method of choice.

Most of the differential equations frequently applied in natural science, such as the diffusion equation, are simple enough for finite difference schemes to handle efficiently and accurately. However, for more intricate differential equations finite differencing may not give accurate approximations of gradients. Refining the mesh is a solution, but this is computationally expensive. For situations where we want to simulate for a very long period refining the mesh may not be a feasible option. For these particular cases, it seems like neural network is the prominent method and a great alternative to conventional numerical schemes. Hence, the method of choice is problem specific.

Vikenes: Burde kommentere at NN vil være bedre/raskere for mer komplekse modeller, og at NN er best dersom man skal simulere over lange perioder.

7 Conclusion

In this project we have particularly studied the diffusion equation, a rather simple PDE. It has a closed form expression for the analytical solution, so strictly speaking it is not

necessary to solve the equation numerically. Still, it is a good representation for comparing numerical methods. Our results have proven that forward euler clearly outperforms neural network in terms of MSE between approximated and analytical solution, as long as it fulfills the stability requirement (2.4). The potential for finding eigenvalues by solving the first order ODE (3.3) has also been studied, in which case forward euler and neural network perform comparably well, although forward euler generally runs faster.

The two differential equations studied in this project are linear and rather simple differential equations. In a general case, differential equations can be much more intrinsic. In such cases, numerical schemes and particularly forward euler encounter problems with accurate finite differences. Neural networks, on the other hand, can according to the universal approximation theorem approximate any function to arbitrary accuracy without any stability constraint on the time step. Hence, neural network is a more flexible numerical technique with a wide applicability.

8 Appendix

Appendix

Analytical solution of the diffusion equation

By using the concept of separation of variables, the solution can be expressed as

$$u(x, t) = X(x)T(t)$$

The solution is separated into a function X only depending on the independent variable x , and a function T only depending on the independent variable t . Equation (2.1) can then be rewritten as

$$\begin{aligned}\frac{\partial^2 X(x)T(t)}{\partial x^2} &= \frac{\partial X(x)T(t)}{\partial t} \\ T(t)\frac{\partial^2 X(x)}{\partial x^2} &= X(x)\frac{\partial T(t)}{\partial t} \\ \frac{1}{X}\frac{\partial^2 X(x)}{\partial x^2} &= \frac{1}{T}\frac{\partial T(t)}{\partial t}\end{aligned}$$

The core of the method now becomes clear. The independent variables are separated and put on either side of the equation. Because x and t are independent we may fix one of them, say x , while letting the other (t) vary. The left side of the equation is thus constant, and since we have equality the expression on the right side must equal the same constant, for arbitrary t . Therefore, we can set the left side and right side equal to a constant $-k^2$.

The reason for defining a negative constant is to prevent a growing solution, which will be clear on the derivation.

$$\frac{1}{X} \frac{\partial^2 X(x)}{\partial x^2} = \frac{1}{T} \frac{\partial T(t)}{\partial t} = -k^2$$

This is solved for the functions X and T separately.

$$\begin{aligned} \frac{1}{T} \frac{dT}{dt} &= -k^2 \\ \int \frac{1}{T} dT &= - \int k^2 dt \\ \ln T &= -k^2 t + \hat{T}_0 \\ T &= e^{-k^2 t + \hat{T}_0} \\ &= T_0 e^{-k^2 t} \end{aligned}$$

$$\frac{1}{X} \frac{\partial^2 X}{\partial x^2} = -k^2 \tag{8.1}$$

$$\frac{d^2 X}{dx^2} = -k^2 X \tag{8.2}$$

$$X = X_0 \sin(kx) + X_1 \cos(kx) \tag{8.3}$$

Here we have used the fact that a differential equation of the form $X'' = -X$ has a trigonometric solution. The general solution becomes

$$u(x, t) = X(x)T(t) = T_0 e^{-k^2 t} [X_0 \sin(kx) + X_1 \cos(kx)]$$

To obtain a specific solution we use the initial and boundary conditions to find the undetermined coefficients T_0 , X_0 and X_1 . At $x = 0$ we have

$$\begin{aligned} u(0, t) &= 0 \\ T_0 e^{-k^2 t} X_1 &= 0 \end{aligned}$$

The only possibility is $X_1 = 0$ because if $T_0 = 0$ then the general solution becomes time-independent, and does not describe the temporal variability of (2.1). At $x = L$ we have

$$\begin{aligned} u(L, t) &= 0 \\ T_0 e^{-k^2 t} X_0 \sin(kL) &= 0 \end{aligned}$$

Following the previous argument we can't have $X_0 = 0$ as this would imply independence of space. Instead, we must have

$$\begin{aligned} \sin(kL) &= 0 \\ kL &= n\pi \\ k &= \frac{n\pi}{L}, \quad n = 0, 1, 2, \dots \end{aligned}$$

Defining a new constant $A = T_0 X_0$ we get the following solution for an eigenfrequency n .

$$u(x, t) = A e^{-k^2 t} \sin\left(\frac{n\pi x}{L}\right)$$

Because any linear combination of a solution is also a solution, the full specific solution is given by the linear combination of all eigenfrequencies n .

$$u_n(x, t) = \sum_{n=1}^{\infty} A_n e^{-k_n^2 t} \sin\left(\frac{n\pi x}{L}\right)$$

The coefficients A_n are determined by the initial condition.

$$\begin{aligned} u_n(x, 0) &= \sin(\pi x) \\ \sum_{n=1}^{\infty} A_n \sin\left(\frac{n\pi x}{L}\right) &= \sin(\pi x) \end{aligned}$$

This is a Fourier sine series with the following expression for the coefficient.

$$\begin{aligned} A_n &= \frac{2}{L} \int_0^L \sin(\pi x) \sin\left(\frac{n\pi x}{L}\right) dx \\ &= 2 \int_0^1 \sin(\pi x) \sin(n\pi x) dx \end{aligned}$$

Note that for $n \neq 1$, the two factors in the integrand are mutually orthogonal functions on domain $x \in [0, 1]$. Therefore, the integral vanishes. For $n = 1$ we have

$$\begin{aligned}
A_1 &= 2 \int_0^1 \sin^2(\pi x) dx \\
&= 2 \cdot \frac{1}{2} \int_0^1 [1 - \cos(2\pi x)] dx \\
&= [x - \frac{1}{2\pi} \sin(2\pi x)]_0^1 \\
&= 1
\end{aligned}$$

This gives

$$A_n = \begin{cases} 1, & n = 1 \\ 0, & n \neq 1 \end{cases}$$

In other words, the specific solution only contains one eigenfunction, $n = 1$, with wave number given by

$$k_1 = \frac{1 \cdot \pi}{1} = \pi$$

Finally, this gives the following analytical solution of (2.1)

$$u(x, t) = e^{-\pi^2 t} \sin(\pi x)$$

Forward Euler unit test results

The results of the unit test for the Forward Euler scheme is given in table 2.

	$j = 1$		$j = 2$	
	Manual	Code	Manual	Code
u_0^j	0	0	0	0
u_1^j	0.531656755	0.53165676	0.480888053	0.48088805
u_2^j	0.8602387	0.8602387	0.778093215	0.77809321
u_3^j	0.8602387	0.8602387	0.778093215	0.77809321
u_4^j	0.531656755	0.53165676	0.480888053	0.48088805
u_5^j	0	0	0	0

Table 2. Unit test of implementation of Forward Euler scheme. Results from manual calculations are compared with the output of forward euler implementation, for time steps $j = 1$ ($t = \Delta t$) and $j = 2$ ($t = 2\Delta t$).