Machine Learning: Using Forward Euler and Neural Networks to Solve Differential Equations

Vetle Nevland, Vetle Vikenes & Sigurd SÃÿrlie Rustad

FYS-STK4155 âĂŞ 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

To Do

Hovedrapport:

- Nevland: makefile

Ekstraoppgave:

- Nevland: Readme
- Nevland: makefile
- Nevland: docstrings

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 repository[1].

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 initiaites 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 applicablility. 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

[1]https://github.com/sigurdru/FYS-STK4155/tree/main/project3

considered is if neural networks can compete with state-of-the-art numerical methods for solving differential equations.

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

$$\frac{\partial u(\mathbf{r},t)}{\partial t} = \nabla \cdot \left[D(u,\mathbf{r})\nabla u(\mathbf{r},t)\right], \tag{2.1}$$

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

$$\frac{\partial u}{\partial t} = \nabla^2 u(\mathbf{r},t), \tag{2.2}$$

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

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t}, \tag{2.3}$$

with boundary conditions

$$u(x,0) = \sin(\pi x) \quad 0 \leq x \leq L, \tag{2.4}$$

$$u(0,t) = 0 \quad t \geq 0 \text{ and} \tag{2.5}$$

$$u(L,t) = 0 \quad t \geq 0. \tag{2.6}$$

2.2 Analytical solution

An analytical solution of the 1D diffusion equation can be dervied using the method of separation of variables. For a complete derivation, please refer to the Appendix 7.1. The analytical solution satisfying the given initial and boundary conditions is

$$u(x,t) = e^{-\pi^2 t}\sin(\pi x) \tag{2.7}$$

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.

To approximate the solution of equation (2.3), 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,

$$t_j = j\Delta t, \quad j \in \mathbb{N}_0 \quad \wedge \quad x_i = i\Delta x, \quad \{i \in \mathbb{N}_0 | i \leq N\}. \tag{2.8}$$

The algorithm for explicit forward Euler in one dimension (from [2] chapter 10.2.1) reads

$$u_{i,j+1} = \alpha\, u_{i-1,j} + (1 - 2\alpha)u_{i,j} + \alpha\, u_{i+1,j} \tag{2.9}$$

where

$$\alpha = \frac{\Delta t}{\Delta x^2}. \tag{2.10}$$

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.

$$\alpha \leq \frac{1}{2}. \tag{2.11}$$

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 [4].

2.4 Solving Differential equations with deep learning

Many of the concepts we use in this section are covered in a previous report [6]. 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. [5] and a Jupyter-Notebook by jan blechschmidt [1]. This method works with any partial differential equation (PDE), however we limit ourselves to the diffusion equation in one dimension (2.3) which we are going to study in this report.

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

$$u_{\hat{\theta}}(x, t) \approx u(x, t). \tag{2.12}$$

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

$$u_\theta(x,t) = u_0(x,t) + f\Big(x, N(x,\theta)\Big), \tag{2.13}$$

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 zero 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 (2.3)

$$r_\theta(x,t) \equiv \frac{\partial u_\theta}{\partial t} - \frac{\partial^2 u_\theta}{\partial x^2}. \tag{2.14}$$

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 minimized. For this we use the mean sum of squared residuals, including the initial and boundary points, across all data samples n

$$L(x,t,\theta) = \frac{1}{n}\sum_{i=1}^{n}[r_\theta(x_i,t_i)]^2 + \frac{1}{n_B}\sum_{i=1}^{n_B}(u_\theta(x_b,t_i) - u(x_b,t_i))^2 + \frac{1}{n_I}\sum_{i=1}^{n_I}(u_\theta(x_i,0) - u(x_i,0))^2. \tag{2.15}$$

Note, because we know the initial and boundary conditions we include them in the loss function, and they will act as regularization terms. n_B and n_I is the number of training points on the boundary and the initial time step, respectively. The boundary points are denoted by x_b. Optimization is then done through a form of gradient descent, minimizing the loss with respect to the parameters θ

$$\theta \leftarrow \theta - \eta\nabla_\theta L(x,t,\theta), \tag{2.16}$$

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, indicating that we have reached an optimal solution.

2.5 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) [7] closely, using the same first order ODE as them and closely follow their methods for finding the eigenvalues. The first order ODE in question is given as

$$\frac{dx(t)}{dt} = -x(t) + f[x(t)], \tag{2.17}$$

where

$$f(x) = [x^T xA + (1 - x^T Ax)I]x \tag{2.18}$$

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 (2.17) 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 (2.17) span the eigenspace of A, implying that the equilibrium points are precisely the eigenvectors of A. That is, any $\tilde{x}$ that satisfies

$$\begin{aligned} \frac{d\tilde{x}(t)}{dt} &= -\tilde{x} + f[\tilde{x}(t)] \\ &= -\tilde{x} + [\tilde{x}^T \tilde{x} A + (1 - \tilde{x}^T A\tilde{x})I]\tilde{x} \\ &= \tilde{x}^T \tilde{x} A\tilde{x} - \tilde{x}^T A\tilde{x}\tilde{x} = 0, \end{aligned} \tag{2.19}$$

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

$$r = \frac{\tilde{x}^T \tilde{x}}{\tilde{x}^T A\tilde{x}}. \tag{2.20}$$

An essential question is how the neural network should be initialized. That is, what trial solution of the form (2.13) should be chosen? Convergence analysis reveal that any non-zero trial solution $x(0) \in \mathbb{R}^n$ will give a solution of (2.17) that converges to an eigenvector of A. Moreover, if $x(0)$ is not orthogonal to the subspace spanned by the largest eigenvector $\lambda_{\max}$ of A, then the solution $x(t)$ converges to the corresponding eigenvector.

The differential equation (2.17) is a first order nonlinear ODE. Generally, solving a non-linear 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 (2.17) numerically using the forward euler method. Discretizing the equation and solving for next time step $n + 1$ gives

$$\begin{aligned}\frac{x^{n+1}-x^n}{\Delta t} &= -x^n + f[x^n] \\ &= (x^n)^T x^n A x^n - (x^n)^T A x^n x^n, &\quad (2.21)\\ \Rightarrow x^{n+1} &= x^n + \Delta t\Big[(x^n)^T x^n A x^n - (x^n)^T A x^n x^n\Big]. &\quad (2.22)\end{aligned}$$

Starting with a non-zero initial condition $x^0 = I(x)$, convergence is fulfilled as long as the forward euler scheme is stable. The stability criteria depends on the discretization parameter Δt, and is to be approximated empirically.

3 Method

3.1 Unit testing

Before the forward Euler 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

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

The initial condition used is $u_*^0 = \Big(\sin(0), \sin(0.2\pi), \sin(0.4\pi), \sin(0.6\pi), \sin(0.8\pi), \sin(\pi)\Big)$. 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 4 in Appendix 7.2.

3.2 Neural network setup

Our first step when designing our neural network is to decide on a architecture. We have studied this more in-depth in a previous report [6]. Because of this we will only briefly investigate how the depth (number of hidden layers) affects the loss (2.15). We use similar architecture as Raissi et. al. (2017) [5], namely 20 nodes in each hidden layer, with hyperbolic tangent as activation function in each hidden node. The depth we will test for is $N_h \in \{2, 4, 8\}$ hidden layers, then choose the architecture with minimal loss after 500 training iterations. We also plot how the loss evolves as a function of iterations to see the evolution of training. The network is set up using TensorFlow with linearly stacked dense layers (sequential) with standard Glorot normal initialization (also called Xavier normal initializer). As optimizer we use Adam, which is a stochastic gradient decent method based on adaptive estimation of first- and second-order moments. Because of previous success with dynamic learning rate in the preceding project [6], we implement TensorFlow's piecewise constant decay schedule. Specifically we use learning rate 10^{-2} for the first 1000 steps, 10^{-3} from iterations 1000 to 3000 and $5 \cdot 10^{-4}$ for succeeding steps.

To train the network we need data, and we will use random data points uniformly distributed. In figure 1 we display the data points used for training. Here we choose 50 data points for $t = 0$ and $x \in [0, 1]$, and another 50 points for $t \in [0, 1]$ and $x \in \{0, 1\}$. This is to make sure we also test for boundary points when calculating the loss (2.15), and will act as a regularization. Next we also choose 10^4 data points (t_i, x_i), where $t_i, x_i \in [0, 1]$.

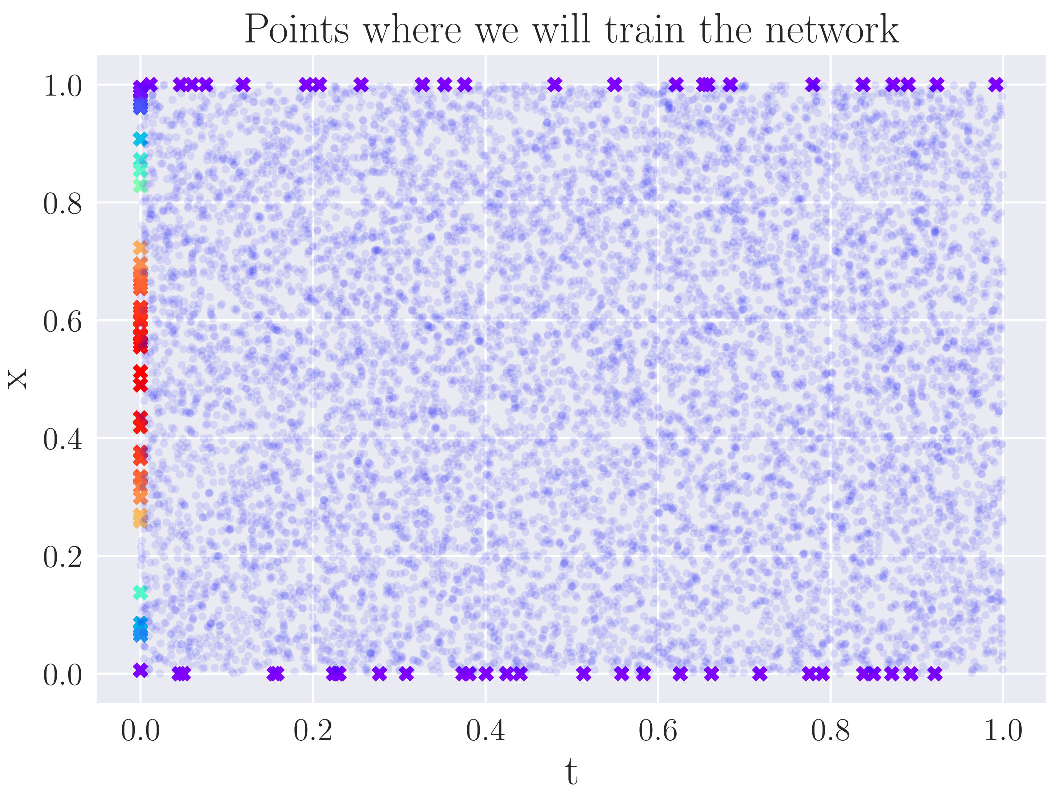


Figure 1. In this plot we highlight the data points used for training the neural network. Points on the boundary are highlighted, and we use 50 points for $t = 0$, and a total of 50 points at $x = 0$ and $x = 1$, resulting in a total of 100 boundary points. We use 10^4 data points not at the boundary.

3.3 Forward Euler

As discussed in the theory section, following equation (2.11), $\alpha = 0.5$ 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 particular time step, j, is given as

$$\mathrm{MSE_j} = \frac{1}{N} \sum_{i=0}^{N-1} (u^j_{p,i} - u^j_{a,i})^2 \tag{3.1}$$

where u_e denotes the analytical solution and N is the number of spatial points. By construction the MSE should be zero at $t = 0$. Subsequently, we expect an increasing MSE. 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 test, this serves as a simple method of validating our result.

3.4 Comparing numerical results

To proceed, we will study the results of the two numerical solvers at two different times, and 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.

Using the aformentioned time points, we will compare the resulting MSE of the two methods, where we use both $\Delta x = 0.1$ and $\Delta x = 0.01$ for forward Euler. For the neural network we will only consider $\Delta x = 0.01$, but we will calculate the final MSE after four different traing iterations, namely $N_t \in \{500,\ 1000,\ 10^4,\ 5 \cdot 10^4\}$, to see how increasing iterations affect the score.

As a final study, we are going to plot the difference between the analytical solution and the two numerical solutions at t_1 and t_2, as a function of $x \in [0,\ 1]$

$$\Delta u(t) = u_p(t) - u_a(t), \tag{3.2}$$

where $u_p(t)$ is the numerical solution and $u_a(t)$ is the analytical solution. Our main purpose of this is to see whether the neural network can circumvent one of the major drawbacks of the forward Euler scheme, giving an output that is either consistently above or below the actual solution. For more volitale differential equations, such errors can propagate throughout the simulation, resulting in a poor fit. One potential advantage of using neural networks will thus be if it yields unbiased errors.

3.5 Finding eigenvalues

The matrix we will consider for our initial analysis is a real symmetric matrix, A, with dimensions (3×3) [2]. In the end we will perform one simulation with a (6×6) matrix to study differences for higher dimensions.

Using the neural network model (2.17) to find the eigenvalues of A requires a trial solution (2.13) to initiate the network. The trial solution must fulfill the convergence property to guarantee that we end up with the eigenvalues, as mentioned in the theory section. Hence, we define the initial condition $x_0 \in \mathbb{R}^n$ 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 non-orthogonal to the eigenspace corresponding to the largest eigenvalue. To increase the probability of non-orthogonality, we can add 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 preknowledge of a solution of the differential equation, incorporating this in the trial solution helps accelerate the learning of the neural network. The ODE (2.17) is of first order with a term involving 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 (2.13) is

$$\begin{aligned} g_\theta(t) &= g_0(x(t)) + f[x(t), N(x,\theta)] \\ &= e^{-t}x_0 + (1-e^{-t})N(x(t),\theta). \end{aligned} \tag{3.3}$$

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, dx/dt, and the resulting residual of (2.17). 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. Unless specified, a total time of $T = 5$ is used, with a learning rate and number of epochs set to 0.005 and 2000 respectively, for the neural network.

For each simulation, we find the eigenvector, $\mathbf{v}_{\max} = (v_1, v_2, v_3)$, with the largest corresponding eigenvalue, $\lambda_{\max}$. We then plot the computed value of the three eigenvector components as a function of time, for both Forward Euler and Neural network. For a given simulation we also plot the time evolution of the Rayleigh quotient (2.20), for the two forementioned methods.

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

4 Results

4.1 Finding a suitable Neural network

Table 1 shows the loss of the neural network after 500 iterations for the three different network architectures. We see that 4 hidden layers clearly yields the lowest value for the final loss.

Architecture	Loss
8×20	0.0048
4×20	0.00079
2×20	0.0022

Table 1. Computed loss, equation (2.15), of the neural network after 500 training iterations. The first number in the first column represents the number of hidden layers of the network, while the second number is the number of hidden nodes in each layer, which is 20 for all three networks. $N_h = 4$ yields the lowest MSE.

The loss of the network as a function of iterations is shown in figure 2 for three different architectures, using 2, 4 and 8 hidden layers for the upper left, upper right and lower left panel, respectively.

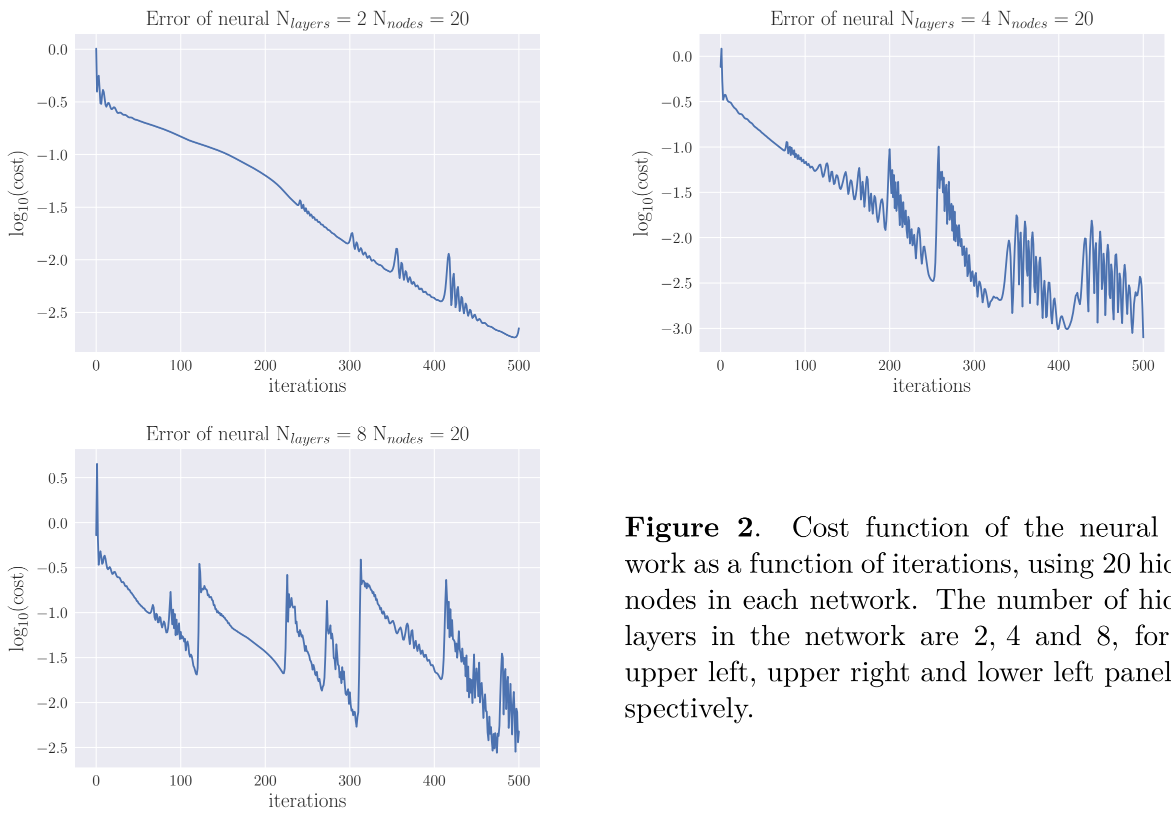


Figure 2. Cost function of the neural network as a function of iterations, using 20 hidden nodes in each network. The number of hidden layers in the network are 2, 4 and 8, for the upper left, upper right and lower left panel, respectively.

We notice large fluctuations of the loss for the network with 4 hidden layers. Using logarithmic scales on the y-axes in figure 2, the amplitudes of the oscillating losses appear bigger than they are in reality. As planned, we choose the network with $N_h = 4$ for further analysis, as it yields the lowest MSE after 500 iterations. A detailed study of different architectures is outside the scope of this report, and is nonetheless dependent of the particular PDE in question.

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 $t \to 1$.

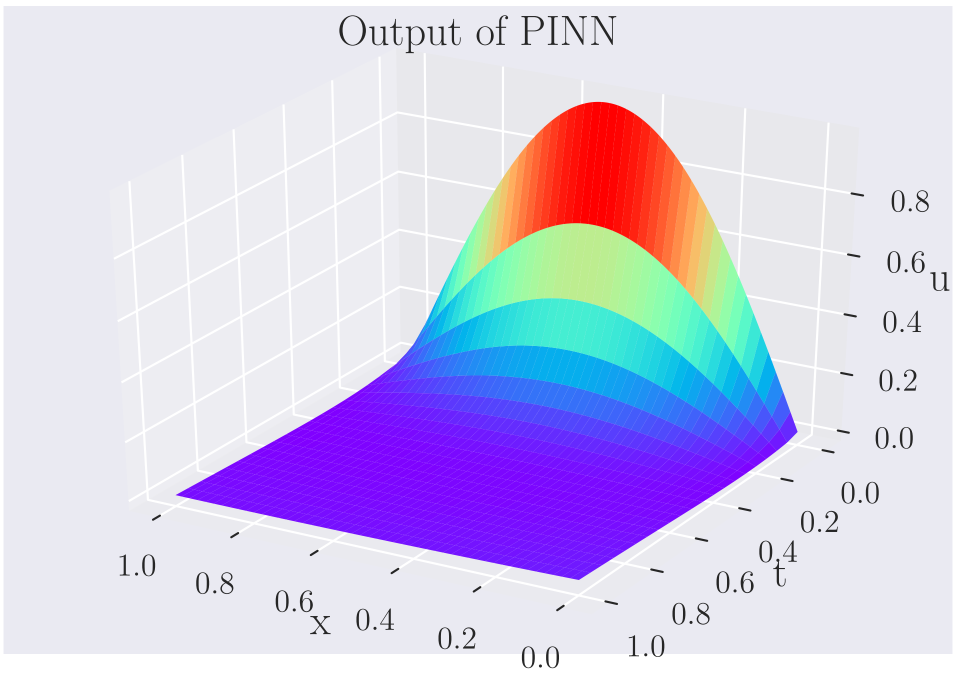


Figure 3. The solution of the diffusion equation of the final neural network, with $N_h = 4$ hidden layers.

4.2 Error of Forward Euler scheme

To assess the accuracy of our implemented numerical scheme, we need to test the how well the produced solution fits the analytical solution. Figure 4 shows the MSE as function of time for the forward euler scheme (note the scaling of the y-axis). Keeping in mind that we choose the time step to give a stability factor of $\alpha = 0.5$.

The MSE in figure 4 increases significantly for early time steps, reaching a peak at around $t_1 = 0.1$ and quickly reduces thereafter. At $t_2 = 0.5$ the MSE is significantly lower compared to t_1.

4.3 Comparison of error

Table 2 shows the resulting MSE of the neural network at t_1 and t_2 after four different number of training iterations. Except from an increase for t_1 when we go from 500 to

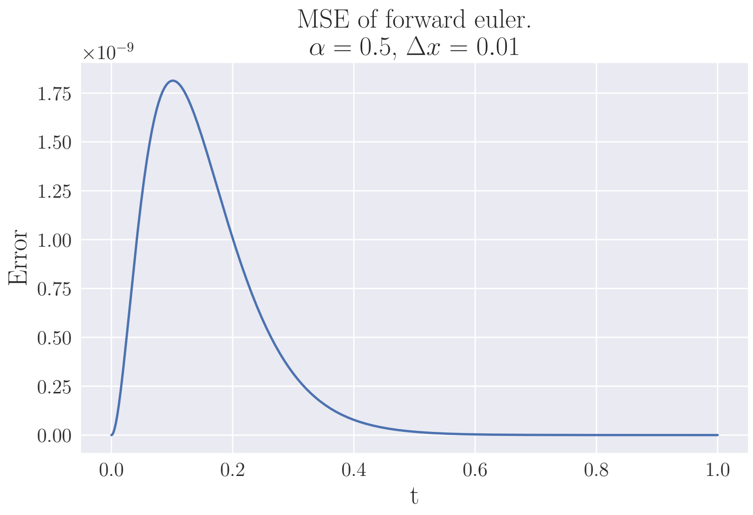


Figure 4. Mean squared error of approximated solution by forward euler, using $\Delta x = 0.01$ and time step Δt dictated by the stability criterion (2.11).

1000 training iterations, the MSE decreases as we increase the number of iterations. The increase is likely due to the fluctuations we saw in figure 2. We will therefore use the output of the neural network after $5 \cdot 10^4$ training iterations for the remaining analysis on the diffusion equation, as this yielded the lowest MSE.

	Mean Square error	
Number of training iterations	$t_1 = 0.1$	$t_2 = 0.5$
500	0.0017	0.0070
1000	0.015	0.00055
10^4	$1.4 \cdot 10^{-4}$	$9.5 \cdot 10^{-5}$
$5 \cdot 10^4$	$1.0 \cdot 10^{-6}$	$3.0 \cdot 10^{-6}$

Table 2. MSE of the neural network at two different times, using four different training iterations.

Figure 5 shows the plot of equation (3.2), i.e. the numerical solution of the diffusion equation from forward Euler, subtracted by the analytical solution in equation (2.7). The corresponding plot for the neural network is shown in figure 6. Using $\Delta x = 0.1$ for forward Euler yielded an identically shaped plot as figure 5, but with increased values on the y-axis, and we have therefore chosen to omit it.

In figure 5 we see the expected feature of forward euler, lying below the analytical solutions throughout the simulation. Comparing with figure 6, we note that at $t_1 = 0.1$ the difference is positive for $x \lesssim 0.3$, while it is negative for larger x values. Similarly, we get a positive difference at $t_2 = 0.5$ for the lower values of x. Although there is a slight bias in the difference from the neural network, neither time points yielded a completely negative solution.

Table 3 compares the MSE obtained for forward euler and neural network at two selected time steps t. With forward Euler we have included both spatial steps Δx, while we only consider $\Delta x = 0.01$ for the neural network.

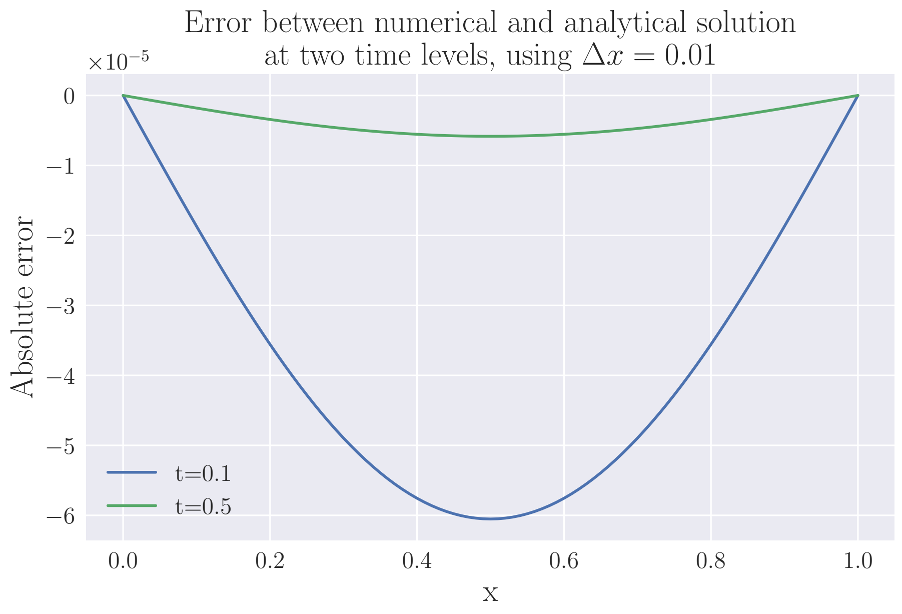


Figure 5. The difference between the analytical solution of the diffusion equation and the solution obtained by forward euler at two different points in time.

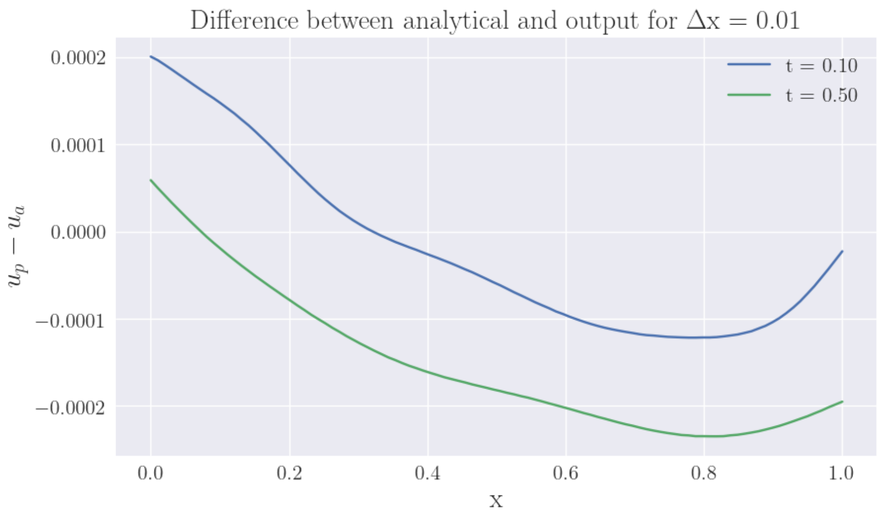


Figure 6. The difference between the analytical solution of the diffusion equation and the solution obtained by the neural network at two different points in time, after $5 \cdot 10^4$ training iterations. The neural network has $N_h = 4$ hidden layers.

	$\Delta x = 0.1$		$\Delta x = 0.01$	
	FE	**NN**	**FE**	**NN**
$t_1 = 0.1$	$1.72 \cdot 10^{-5}$	-	$1.81 \cdot 10^{-9}$	$1.0 \cdot 10^{-6}$
$t_2 = 0.5$	$1.50 \cdot 10^{-7}$	-	$1.69 \cdot 10^{-11}$	$3.0 \cdot 10^{-6}$

Table 3. 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.

In table 3 we see that forward Euler yields much lower MSE values for $\Delta x = 0.01$ than the neural network. For $t_1 = 0.1$ and $t_2 = 0.5$ the MSE of the neural network exceeds the MSE of forward Euler by factors of $\sim 10^3$ and $\sim 10^5$, respectively. At $t_2 = 0.5$ the MSE of the neural network with $\Delta x = 0.01$ is approximately 20 times greater than the MSE of forward Euler with $\Delta x = 0.1$, i.e. 10 times fewer data points.

4.4 Eigenvalue problem

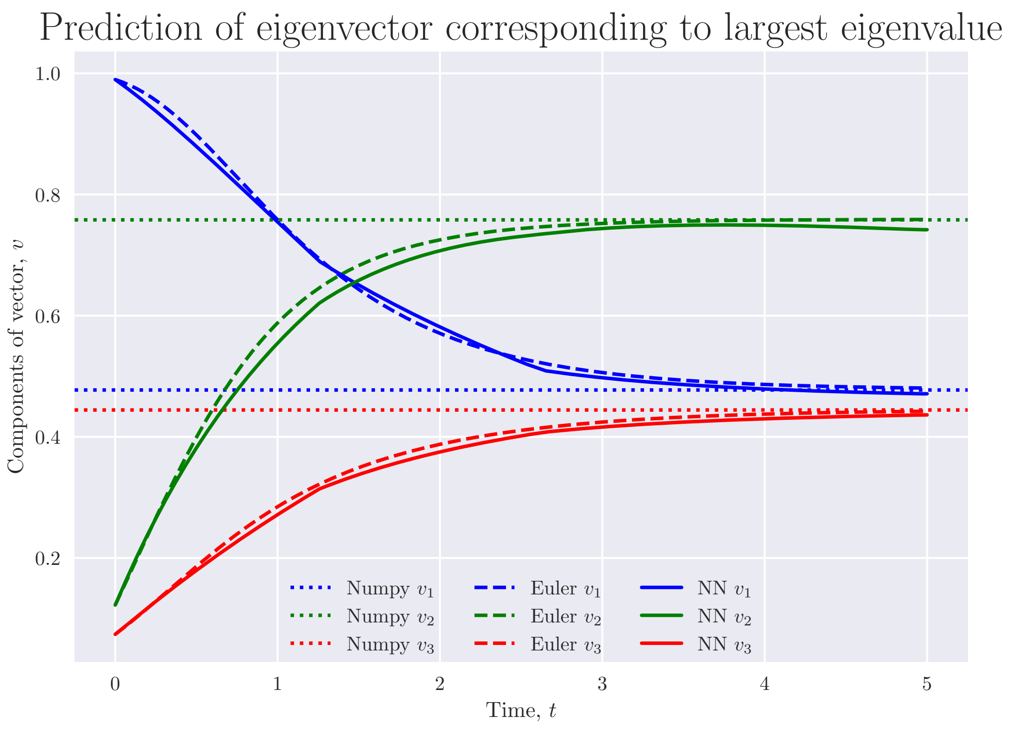


Figure 7. Predictions by forward euler (FE) and neural network (NN) of the eigenvector corresponding to the largest eigenvalue of a real, symmetric 3×3 matrix. The horizontal dotted lines are the components of the eigenvector as found by NumPy.

Figure 7 shows predictions of the eigenvector corresponding to the largest eigenvalue of a symmetric (3×3) matrix A of randomly assigned values between zero and one. 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 obtained from NumPy. The largest eigenvalue is shown in Figure 8 together with the rayleigh quotients estimated by forward euler and the neural network.

Note that some of the figures that follow are horizontally aligned for readability purposes. This means that the axes labels are small, but we inform that the labels are identical to those of figure 7 and figure 8. The plots are only meant to illustrate trends as we alter simulation parameters.

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

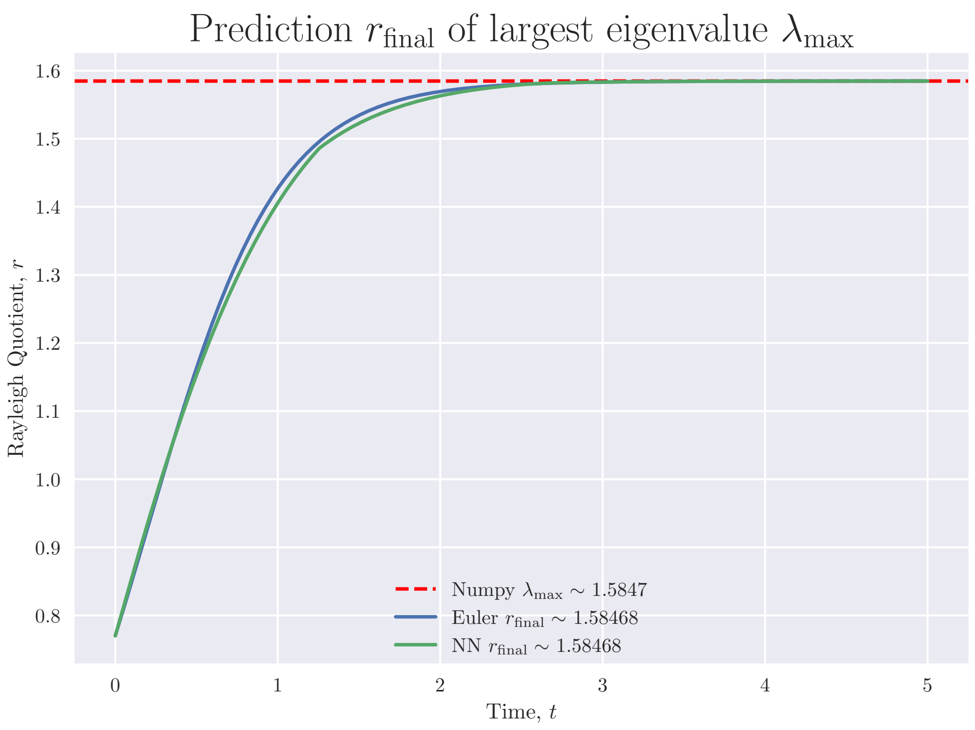
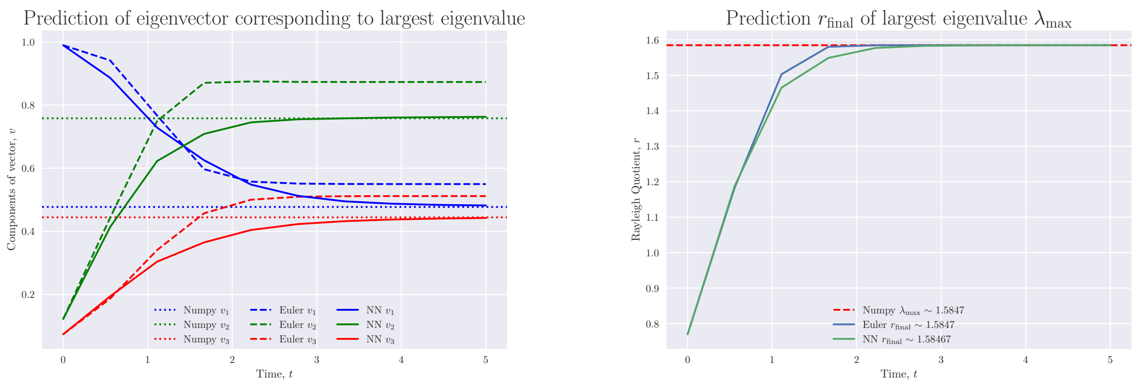


Figure 8. Rayleigh quotients of forward euler (FE) and neural network (NN) representing predictions of the largest eigenvalue of a real, symmetric 3×3 matrix. The horizontal dotted line is the largest eigenvalue as found by NumPy.

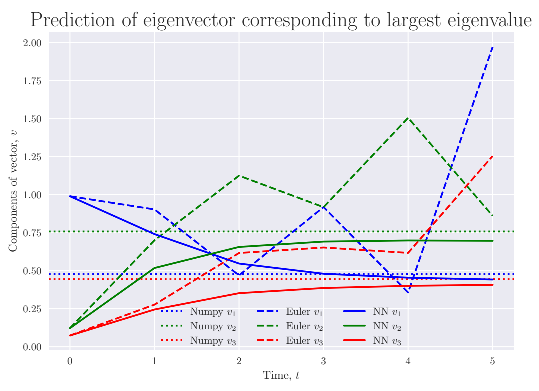


(a) Eigenvectors (b) Rayleigh quotients

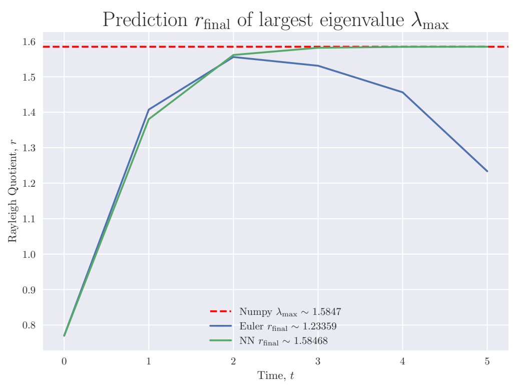
Figure 9. Predictions by forward euler (FE) and neural network (NN) of largest eigenvalue and corresponding eigenvector of a real, symmetric 3×3 matrix, using a time step $\Delta t = 0.5$. The x and y axes are identical to those in figure 7 and 8

Results of a further increase in the time step is shown in Figure 10, using $\Delta t = 0.83$ corresponding to $N = 6$ estimations, where the left panel shows the eigenvector components and the right panel shows the associated Rayleigh quotients. Note that diverging eigenvector components in figure 10a have altered the y-axis, but is otherwise similar to that of previous figures.

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

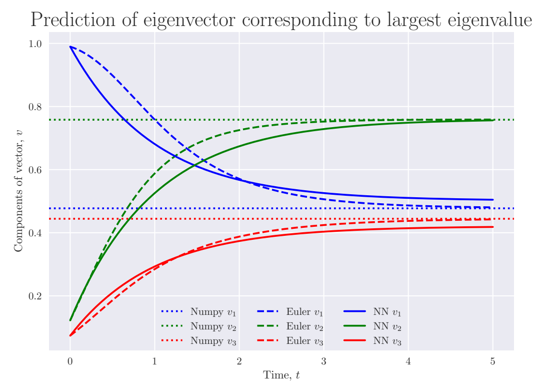


(a) Eigenvectors

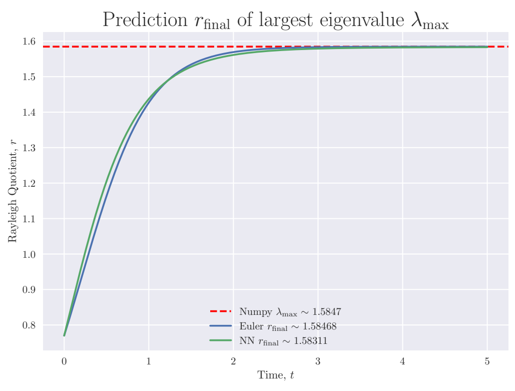


(b) Rayleigh quotients

Figure 10. 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) Eigenvectors



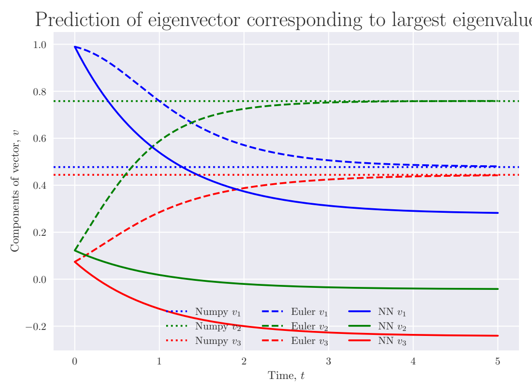
(b) Rayleigh quotients

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

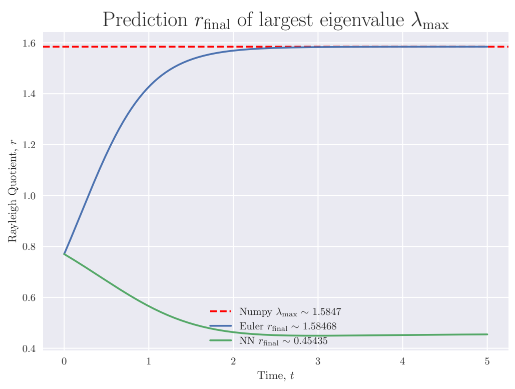
0.1. The associated rayleigh quotients is shown in Figure 11b. Predictions of the same eigenvector is shown in Figure 12a, with the learning rate retained at 0.1 but the number of epochs reduced from 2000 to 50. The associated rayleigh quotients are shown in Figure 12b.

An additional simulation using a 6×6 matrix has also been conducted. Figure 13 and Figure 14 shows the predicted eigenvector for forward euler and neural network, repsectively, and Figure 15 the rayleigh coefficient corresponding to the largest eigenvalue of a 6×6 symmetric matrix. A time step of $\Delta t = 0.005$ have been used and a total time of $T = 2$. The reason for the latter choice is based on experiments with a (6×6) matrix indicating convergence long before $T = 5$. Simulating for a shorter time period better illustrates the convergence of the eigenvector.

The reason for the latter choice is based on the previous simulations with $\Delta t = 0.005$, as both forward euler and neural network converge long before $T = 5$. Reducing the



(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

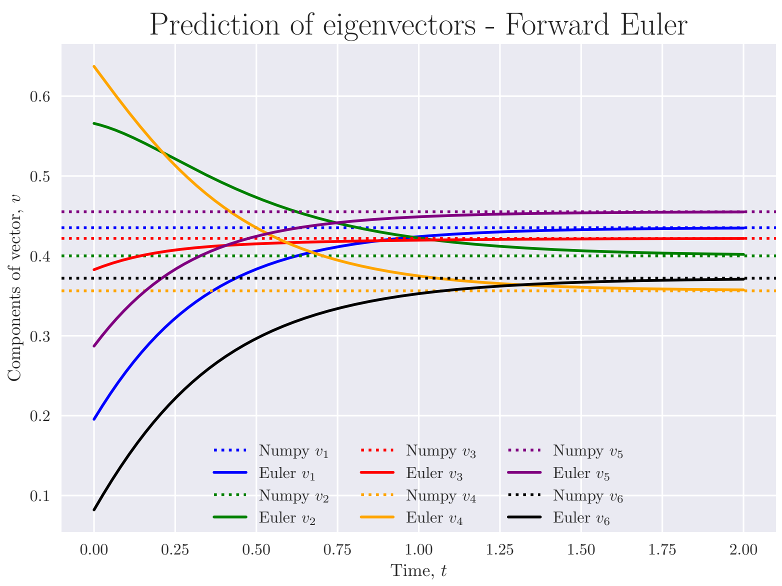


Figure 13. Predictions of eigenvector corresponding to the largest eigenvalue for a real, symmetric 6×6 matrix using Forward Euler. A time step of $\Delta t = 0.005$ and a total time $T = 2$ is used.

simulation time makes it easier to illustrate the convergence property of the six eigenvector components.

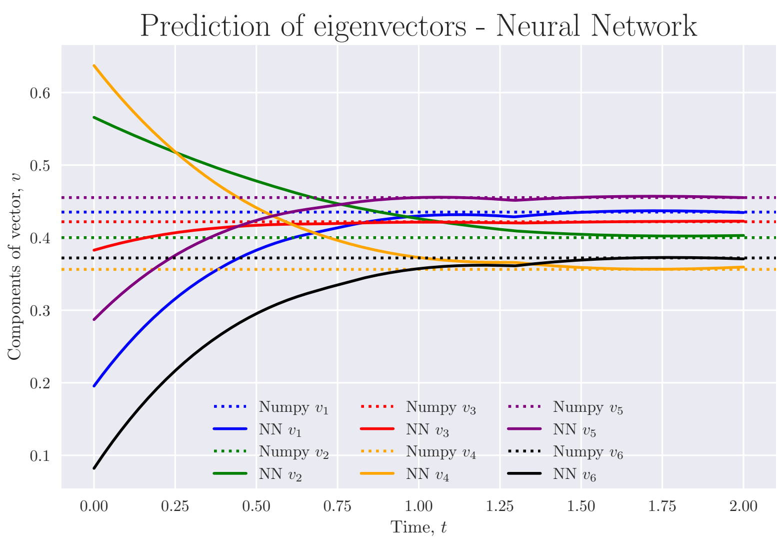


Figure 14. Predictions of eigenvector corresponding to the largest eigenvalue for a real, symmetric 6×6 matrix using Neural network. A time step of $\Delta t = 0.005$ and a total time $T = 2$ is used.

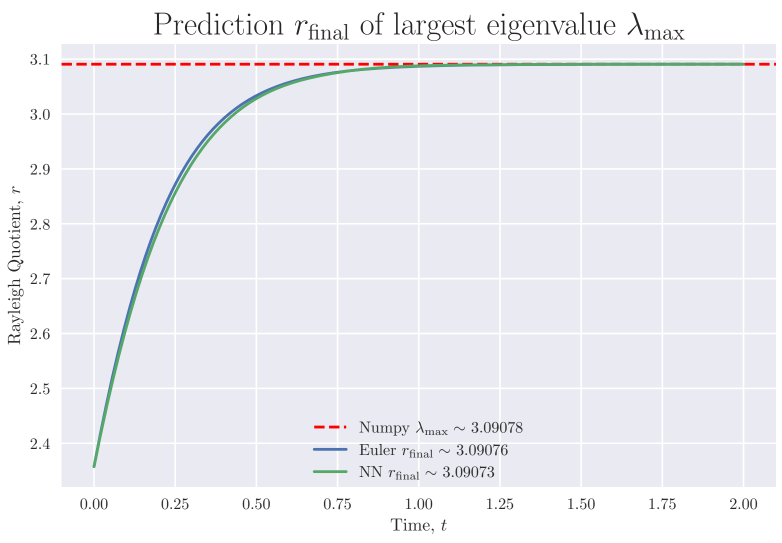


Figure 15. Rayleigh quotient corresponding to the largest eigenvalue of a real, symmetric 6×6 matrix.

5 Discussion

5.1 Solving the diffusion equation with Forward Euler

Sigurd: Skriv om under og kort ned 1: Konsekvent negativ. 2: StÃÿrst feil pÃĕ mest kritiske punkter. 3: Minimere dt. 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 [4].

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.

5.2 Solving the diffusion equation with a neural network

One reason that the loss in figure 2 flucuates instead of decreasing monotonically could be that we use the Adam optimizer [3]. As mentioned, this is due to a type of stochastic optimization method, providing more favorable features than traditional stochastic methods. Sigurd. 1: Ikke optimalisert nettverket, ikke grundig arkitekturanalyse, f.eks. aktivering, learning rate, design etc. 2: Ikke perfekt boundaries. 3: skjeve differanser(bias). 4. ulempe at boundaries ikke er satt

5.3 Comparing

Vikenes: Kommentere at t_1 gir lavere MSE enn t_2 for NN. Kan sees fra differansegrafen.

One noteworthy aspect of the neural network results, was that the MSE at $t_1 = 0.1$ was three times lower than it was at $t_2 = 0.5$. This is easily seen from figure 6, where the difference between the numerical and analytical solution is distributed more evenly between positive and negative values for t_1. The reason for this however is not apparent. One possible explanation might be that as the network trains and it converges to the analytical solution, the gradients at larger t values are smaller, adjusting the corresponding

1: Ikke samme bias for NN som for FE. 2: Lavere MSE og diff for FE(kommentar - diffusjon er jÃ¦vla snill). 3: Potential. 4: Konvergenstid

5.4 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 (2.3) 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 (2.11). 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 (2.13) 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.

5.5 Using Forward Euler and Neural Network to find eigenvalues

Forward euler and neural networks are both capable of finding the eigenvalues of a real, symmetric matrix A by solving the ODE in equation (2.17). The result of forward euler is sensitive to the time step Δt. Comparing Figure 7 and Figure 9a for time steps $\Delta t = 0.005$ and $\Delta t = 0.5$, respectively, we observe that the accuracy of forward Euler 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 and figure 9b, converge precisely to the largest eigenvalue calculated by NumPy. 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 is a constant representing the shift, the rayleigh quotient becomes

$$r = \frac{(sv)^T A (sv)}{(sv)^T (sv)} = \frac{v^T A v}{v^T v}.$$

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. 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 $\Delta t = 0.005$. 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 a previous project [6] 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. Figure 12b shows that the rayleigh quotients of the neural network apparently converges to a different value than the largest eigenvalue.

Figure 10a shows that the eigenvector components do not converge but rather oscillate in an inconsistent pattern when we use Forward Euler with a time step $\Delta t = 0.83$. The corresponding rayleigh quotients, shown in Figure 10b, verify that forward euler is not able to converge to the true solution of equation (2.17) 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%.

5.6 Eigenvalue problems

The potential of solving differential equations spans a large domain of mathematical problems. Particularly, the model (2.17) 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 vÃęre bedre/raskere for mer komplekse modeller, og at NN er best dersom man skal simulere over lange perioder.

6 Conclusion

Ting Ãě ha med:

- FE er bedre enn NN for diffusion equation - hvorfor (MSE-konkret, enklere, raskere)
- NN - ikke samme bias som for FE
- ÃĽn setning - diffusjon er easy peazy, NN kan ha stÃÿrre timesteps
- ÃĽn setning - NN mer fleksibel
- Eig: FE-timestep, NN-eta-epochs
- eig: Euler lettere

Figure 4 shows 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 $t = 0.1$. For time levels beyond this, the MSE decreases exponentially and eventually converges towards zero.

Figure 4 shows 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.

Gammel 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 (2.11). The potential for finding eigenvalues by solving the first order ODE (2.17) 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 applicablility.

Figure 13, Figure 14 and Figure 15 simulates the same ODE as before, but with a 6×6 symmetric matrix, using $\Delta t = 0.005$ once again. The plot shows that neither forward Euler nor the 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.

7 Appendix

7.1 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 (2.3) 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.

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

$$\begin{aligned}
\frac{1}{X}\frac{\partial^2 X}{\partial x^2} &= -k^2 && (7.1) \\
\frac{d^2 X}{dx^2} &= -k^2 X && (7.2) \\
X &= X_0 \sin(kx) + X_1 \cos(kx) && (7.3)
\end{aligned}$$

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

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

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 (2.3). 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

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

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 a 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 (2.3)

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

7.2 Forward Euler unit test results

The results of the unit test for the Forward Euler scheme is given in table 4.

	$j = 1$		$j = 2$	
	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 4. 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 $j = 2$ ($t = 2\Delta t$).

References

[1] Jan Blechschmidt. Using physics informed neural networks (pinns) to solve parabolic pdes. 2021. https://colab.research.google.com/github/janblechschmidt/PDEsByNNs/blob/main/PINN_Solver.ipynb.

[2] Morten Hjorth-Jensen. Computational physics, lecture notes fall 2015. *Department of Physics, University of Oslo*, August 2015. https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Lectures/lectures2015.pdf.

[3] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[4] Svein Linge and Hans Petter Langtangen. *Diffusion Equations*, pages 207–322. Springer International Publishing, Cham, 2017.

[5] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations, 2017.

[6] HÃĕkon Olav Torvik, Vetle Vikenes, and Sigurd SÃÿrlie Rustad. Machine learning: Using regression and neural networks to fit continuous functions and classify data. *University of Oslo*, October 2021.

[7] Zhang Yi, Yan Fu, and Hua Jin Tang. Neural networks based approach for computing eigenvectors and eigenvalues of symmetric matrix. *Computers & Mathematics with Applications*, 47(8-9):1155–1164, 2004.