Solving partial differential equations & Eigenvalue Problems with neural networks

Written by:

Jens Bratten Due
Oliver Lerstøl Hebnes
Mohamed Ismail

Department of Physics UiO
December 18, 2019

Abstract

This project is divided into two parts, where the first part gives an analysis of four different methods to approximate a partial differential equation that describes the temperature gradient of a one-dimensional rod with length $L = 1$. The methods implemented are Forward Euler, Backward Euler, Crank-Nicolsen and a deep neural network. It was found that Crank-Nicolsen performed overall best for steps $\Delta x = 0.1 \wedge 0.01$ with the highest error found being $2.70 \cdot 10^{-3}$ and $2.24 \cdot 10^{-4}$, respectively.

The second part uses a deep neural network to find the eigenvalues of a given symmetric 6×6 -matrix, as proposed from the article Computers and Mathematics with Applications 47, 1155 (2004). The eigenvalues converged towards all eigenvalues, and not exclusively the minimum or maximum eigenvalue, as it was proposed from the article.

Contents

I Introduction

The world is dark and full of terrors, and to face these challenges, and the ability of modelling systems as differential/partial differential equations or eigenvalue problems are invaluable. Therefore, we are in need of new methods that can solve these kind of problems more accurate and precise, than standard solvers that exists today.

Thus, in this project we will be looking into solving the general diffusion equation and computing the eigenvalue of a symmetric matrix with the use of a neural network (NN).

In the case of the diffusion equation, we will compare our implementation of NN with standard PDE solvers, such as our own implementation of forward Euler, backward Euler and Crank-Nicolson. While in the eigenvalue scenario, we will try to follow the discussion from the article of Yi et al.[7] and attempt to find the eigenvalues of a real and symmetric 6×6 matrix, and compare the results with solution from numerical diagonalization with standard eigenvalue solvers from linear algebra, found in the numpy library in python.

II Theoretical Methods and Technicalities

The equation of state is

$$\frac{\partial^2 u}{\partial x^2} = \frac{\partial u}{\partial t}, \quad t > 0, \quad x \in [0, L] \tag{1}$$

or

$$u_{xx} = u_t,$$

with initial conditions, i.e., the conditions at $t = 0$

$$u(x, 0) = \sin(\pi x), \quad 0 < x < L \tag{2}$$

with $L = 1$ the length of the x-region of interest. The boundary conditions are

$$u(0, t) = 0, \quad t \geq 0 \tag{3}$$

and

$$u(L, t) = 0, \quad t \geq 0. \tag{4}$$

Luckily, it exists analytical solutions for this problem, and when doing scientifical research numerically, it is an invaluable resource to have analytical solutions to compare the numerical results with.

One possible analytical solution to the partial differential equation above is given as

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

where the long and exciting derivation of this can be found in Appendix A.

The partial derivatives of this is

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

As seen, these two equations are exactly the same, thus they are indeed an analytical expression for the equation in question.

i Forward Euler - Explicit Scheme

One method of solving the diffusion equation, which we will discuss is the explicit forward Euler algorithm with discretized versions of time given by a forward formula and a centered difference in space resulting in

$$\frac{\partial u}{\partial t} = u_t \approx \frac{u(x, t+\Delta t) - u(x,t)}{\Delta t} = \frac{u(x_i, t_j + \Delta t) - u(x_i, t_j)}{\Delta t}$$

and

$$\frac{\partial^2 u}{\partial x^2} = u_{xx} \approx \frac{u(x+\Delta x, t) - 2u(x,t) + u(x-\Delta x, t)}{\Delta x^2} = \frac{u(x_i + \Delta x, t_j) - 2u(x_i, t_j) + u(x_i - \Delta x, t_j)}{\Delta x^2}$$

with a local approximation error of $O(\Delta t)$ and $O(\Delta x^2)$ respectively.

These equations can then be further simplified into

$$u_t \approx \frac{u_{i,j+1} - u_{i,j}}{\Delta t}$$
$$u_{xx} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2}$$

The one-dimensional diffusion equation can then be rewritten in its discretized version as

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

Defining $\alpha = \frac{\Delta t}{\Delta x^2}$ results in the explicit scheme

$$u_{i,j+1} = \alpha u_{i-1,j} + (1-2\alpha)u_{i,j} + \alpha u_{i+1,j} \tag{5}$$

This system of equations can be rewritten in terms of a matrix-vector multiplication, by defining a matrix $\boldsymbol{A}$

$$\boldsymbol{A} = \begin{bmatrix} 1-2\alpha & \alpha & 0 & 0 & \dots & 0 \\ \alpha & 1-2\alpha & \alpha & 0 & \dots & 0 \\ 0 & \alpha & 1-2\alpha & \alpha & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \alpha & 1-2\alpha & \alpha \\ 0 & \dots & \dots & \dots & \alpha & 1-2\alpha \end{bmatrix}$$

$$U_{j+1} = \boldsymbol{A}U_j \tag{6}$$

where the vector U_j is

$$U_j = \begin{bmatrix} u_{1,j} \\ u_{2,j} \\ \dots \\ u_{n,j} \end{bmatrix}$$

For a computation of the system over j time steps, then the matrix $\boldsymbol{A}$ is applied j times, then we can rewrite the problem as

$$U_j = \boldsymbol{A}U_{j-1} = \boldsymbol{A}\big(\boldsymbol{A}U_{j-2}\big) = \cdots = \boldsymbol{A}^j U_0 \tag{7}$$

ii Backward Euler - Implicit Scheme

Another well known method of solving partial differential equations, is the the implicit Backward Euler with it's discretized version of time given by a backward formula and a centered difference in space resulting in

$$\frac{\partial u}{\partial t} = u_t \approx \frac{u(x,t) - u(x,t-\Delta t)}{\Delta t} = \frac{u(x_i,t_j) - u(x_i,t_j-\Delta t)}{\Delta t}$$

and

$$\frac{\partial^2 u}{\partial x^2} = u_{xx} \approx \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2} = \frac{u(x_i+\Delta x,t_j) - 2u(x_i,t_j) + u(x_i-\Delta x,t_j)}{\Delta x^2}$$

still with a truncation error which goes like $O(\Delta t)$ and $O(\Delta x^2)$.

We now obtain

$$\begin{aligned} \frac{u(x_i,t_j) - u(x_i,t_j-\Delta t)}{\Delta t} &= \frac{u(x_i+\Delta x,t_j) - 2u(x_i,t_j) + u(x_i-\Delta x,t_j)}{\Delta x^2} \\ \frac{u_{i,j} - u_{i,j-1}}{\Delta t} &= \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} \\ u_{i,j-1} &= -\alpha u_{i-1,j} + (1+2\alpha)u_{i,j} - \alpha u_{i+1,j} \end{aligned}$$

where $u_{i,j-1}$ is the only unknown quantity. By defining a matrix $\boldsymbol{A}$

$$\boldsymbol{A} = \begin{bmatrix} 1+2\alpha & -\alpha & 0 & 0 & \dots & 0 \\ -\alpha & 1+2\alpha & -\alpha & 0 & \dots & 0 \\ 0 & -\alpha & 1+2\alpha & -\alpha & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & -\alpha & 1+2\alpha & -\alpha \\ 0 & \dots & \dots & \dots & -\alpha & 1+2\alpha \end{bmatrix}$$

we can reformulate the problem as a matrix-vector multiplication

$$\boldsymbol{A}V_j = V_{j-1} \tag{8}$$

It means that we can rewrite the problem as

$$V_j = \boldsymbol{A}^{-1}V_{j-1} = \boldsymbol{A}^{-1}\big(\boldsymbol{A}^{-1}V_{j-2}\big) = \cdots = A^{-j}V_0 \tag{9}$$

where the vector V_j is

$$V_j = \begin{bmatrix} u_{1,j} \\ u_{2,j} \\ \cdots \\ u_{n,j} \end{bmatrix}$$

and V_0 is the initial vector at time $t = 0$ defined by the initial value $u(x, 0)$.

This is an implicit scheme since it relies on determining the vector $u_{i,j-1}$ instead of $u_{i,j+1}$. If α does not depend on time t, we then only need to invert the matrix once. Alternatively we can solve this system of equations using methods from linear algebra. These are however very cumbersome ways of solving since they involve $\sim O(N^3)$ operations for a $N \times N$. It is much faster to solve these linear equations using methods for tridiagonal matrices, since these involve only $\sim O(N)$ operations.

iii Crank-Nicolson - Implicit Scheme

It is possible to combine the implicit and explicit methods in a slightly more general approach. Introducing a parameter θ (the so-called θ-rule) we can set up an equation

$$\frac{\theta}{\Delta x^2}\big(u_{i-1,j} - 2u_{i,j} + u_{i+1,j}\big) + \frac{1-\theta}{\Delta x^2}\big(u_{i+1,j-1} - 2u_{i,j-1} + u_{i-1,j-1}\big) = \frac{1}{\Delta t}\big(u_{i,j} - u_{i,j-1}\big) \tag{10}$$

which for $\theta = 0$ yields the forward formula for the first derivative and the explicit scheme, while $\theta = 1$ yields the backward formula and the implicit scheme. These two schemes are called the backward and forward Euler schemes, respectively. For $\theta = 1/2$ we obtain a new scheme after its inventors, Crank and Nicolson. This scheme yields a truncation in time which goes like $O(\Delta t^2)$ and it is stable for all possible combinations of Δt and Δx.

To derive the Crank-Nicolson equation, one starts with the forward Euler scheme and Taylor expand $u(x, t + \Delta t)$, $u(x + \Delta x, t)$ and $u(x - \Delta x, t)$. However, for the Crank-Nicolson scheme one requires an additional Taylor expansion of $u(x, t + \Delta t)$, $u(x + \Delta x, t)$ and $u(x\Delta x, t)$ around $(x, t + \Delta t/2)$.

Omitting all the long and exciting derivation, which can be found here [3], we end up with the following equations

$$\frac{\partial u}{\partial t} = u_t \approx \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} = \frac{u(x_i, t_j + \Delta t) - u(x_i, t_j)}{\Delta t}.$$

The corresponding spatial second-order derivative reads

$$\frac{\partial^2 u}{\partial x^2} = u_{xx} \approx \frac{1}{2}\left(\frac{u(x_i+\Delta x,t_j)-2u(x_i,t_j)+u(x_i-\Delta x,t_j)}{\Delta x^2}+ \frac{u(x_i+\Delta x,t_j+\Delta t)-2u(x_i,t_j+\Delta t)+u(x_i-\Delta x,t_j+\Delta t)}{\Delta x^2}\right).$$

Note well that we are using a time-centered scheme wih $t+\Delta t/2$ as center. Using our previous definition of $\alpha = \frac{\Delta t}{\Delta x}$ we can rewrite Eq. (4) as

$$-\alpha u_{i-1,j} + (2+2\alpha)u_{i,j} - \alpha u_{i+1,j} = \alpha u_{i-1,j-1} + (2-2\alpha)u_{i,j-1} + \alpha u_{i+1,j-1} \tag{11}$$

or in matrix-vector form as

$$\Big(2\boldsymbol{I} + \alpha \boldsymbol{B}\Big)\boldsymbol{V}_j = \Big(2\boldsymbol{I} - \alpha \boldsymbol{B}\Big)\boldsymbol{V}_{j-1} \tag{12}$$

where the vector $\boldsymbol{V}_j$ is the same as defined in the implicit case (backward Euler) while the matrix $\boldsymbol{B}$ is

$$\boldsymbol{B} = \begin{bmatrix} 2 & -1 & 0 & 0 & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & -1 & 2 & -1 \\ 0 & \dots & \dots & \dots & -1 & 2 \end{bmatrix}$$

And we can rewrite the Crank-Nicholson scheme as follows

$$\boldsymbol{V}_j = \Big(2\boldsymbol{I} + \alpha \boldsymbol{B}\Big)^{-1}\Big(2\boldsymbol{I} - \alpha \boldsymbol{B}\Big)\boldsymbol{V}_{j-1} \tag{13}$$

iv Truncation errors and stability properties

The truncation errors of all the different methods is given in table 1 and the derivation for them can be found here [3], where it is meticulously shown.

The stability criteria depends on the spectral radius, ρ_{max} of the matrix $\boldsymbol{A}$ in all the methods. The spectral radius of the matrix is defined as the largest absolute eigenvalue of the matrix

$$\rho(\boldsymbol{A}) = max\big\{|\lambda| : det\big(\boldsymbol{A} - \lambda \boldsymbol{I}\big) = 0\big\} \tag{14}$$

And the methods are stabile when they fulfil the condition of $\rho(\boldsymbol{A}) < 1$. The explicit scheme only satisfies the spectral radius for $\Delta t \leq \frac{1}{2}\Delta x^2$. However, the implicit schemes are always stable since their spectral radius always satisfies $\rho(\boldsymbol{A}) < 1$. This has been visualized in table 1. The proof for this can be found here [3].

Table 1: Truncation errors and stability

Scheme	Truncation Error	Stability Requirements
Crank-Nicolson	$O(\Delta x^2)$ & $O(\Delta t^2)$	Stable for all Δt & Δx
Backward Euler	$O(\Delta x^2)$ & $O(\Delta t)$	Stable for all Δt & Δx
Forward Euler	$O(\Delta x^2)$ & $O(\Delta t)$	$\Delta t \leq \frac{1}{2}\Delta x^2$

v Neural Networks

For a brief summary of how a neural network is constructed can be found in a previous https://github.com/ohebbi/FYS-STK4155/tree/master/project2. For a more thorough explanation this can be found here [2]

vi Partial Differential Equations

A partial differential equation (PDE) is an equation involving functions having more than one variable. In general, a PDE for a function $g(\boldsymbol{x})$, where $\boldsymbol{x} = x_1, \ldots, x_N$ with N variables is an equation of the form

$$f\left(\boldsymbol{x}, \frac{\partial g(\boldsymbol{x})}{\partial x_1}, \ldots, \frac{\partial g(\boldsymbol{x})}{\partial x_N}, \frac{\partial^2 g(\boldsymbol{x})}{\partial x_1 \partial x_2}, \ldots, \frac{\partial^n g(\boldsymbol{x})}{\partial x_N^n}\right) = 0 \tag{15}$$

where f is an expression involving all kinds of possible mixed derivatives of $g(x_1, \ldots, x_N)$ up to an order n. In order for the solution to be unique, some additional conditions must also be given.

Given a partial differential equation, as shown above, it is desirable to know how to reformulate it into an equation a neural network can solve. One method is to transform the PDE into an equation where minimization of some parameters must be done. This can be accomplished by formulating a trial solution and utilize the neural networks and it's many functionalities, such as which activation functions to deploy, the number of hidden layers to use, number of neurons within each layer, the learning rate and many more parameters to minimize the trial solution make it converge into the true solution.

i Trial solution

The next question is how to formulate the trial solution. One way of expressing the trial solution is

$$g_{trial}(\boldsymbol{x}) = h_1(\boldsymbol{x}) + h_2(\boldsymbol{x}, N(\boldsymbol{x}, P))$$

where $h_1(\boldsymbol{x})$ is a function that makes $g_{trial}(\boldsymbol{x})$ satisfy a given set of conditions, $N(\boldsymbol{x}, P)$ a neural network with weights and biases described by P and $h_2(\boldsymbol{x}, N(\boldsymbol{x}, P))$ is some expression involving the neural network. The role of the function $h_2(\boldsymbol{x}, N\boldsymbol{x}, P))$, is to ensure that the output of $N(\boldsymbol{x}, P)$ is zero when $g_{trial}(\boldsymbol{x})$ is evaluated at the values of $\boldsymbol{x}$ where the given conditions must be satisfied. The function $h_1(\boldsymbol{x})$ should alone make $g_{trial}(\boldsymbol{x})$ satisfy the conditions.

ii Cost function

The next step is then to use an optimization method to minimize the parameters of the neural network, that being its weights and biases, through backward propagation. For the minimization to be defined, we need to have a cost function at hand to minimize.
By considering equation 15 and the fact that f should be equal to zero, it is possible to use the mean squared error as the cost function. Therefore, the cost function $C(\boldsymbol{x}, P)$ can be expressed as

$$C(\boldsymbol{x}, P) = \left(f\left(\boldsymbol{x}, \frac{\partial g(\boldsymbol{x})}{\partial x_1}, \ldots, \frac{\partial g(\boldsymbol{x})}{\partial x_N}, \frac{\partial^2 g(\boldsymbol{x})}{\partial x_1 \partial x_2}, \ldots, \frac{\partial^n g(\boldsymbol{x})}{\partial x_N^n} \right) \right)^2$$

If we also have M different sets of values for $\boldsymbol{x}$, that is $\boldsymbol{x}_i = (x_1^i, \ldots, x_N^i)$ for $i = 1, \ldots, M$ being the rows in matrix $\boldsymbol{X}$, the cost function can then be generalized into

$$C(\boldsymbol{X}, P) = \sum_{i=1}^{M} \left(f\left(\boldsymbol{x}, \frac{\partial g(\boldsymbol{x})}{\partial x_1}, \ldots, \frac{\partial g(\boldsymbol{x})}{\partial x_N}, \frac{\partial^2 g(\boldsymbol{x})}{\partial x_1 \partial x_2}, \ldots, \frac{\partial^n g(\boldsymbol{x})}{\partial x_N^n} \right) \right)^2 \tag{16}$$

The neural network should then find some parameters P that minimizes the cost function in equation 16 for a set of M training samples $\boldsymbol{x}_i$.

vii Estimating eigenvalues with Neural Network

Following the discussion from Yi *et al.* in the article from Computers and Mathematics with Applications 47, 1155 (2004) it is possible to use a neural network to find the eigenvectors of a symmetric matrix by setting up the problem as a differential equation. Their model goes as follows:

Let $\boldsymbol{A}$ be a $n \times n$ symmetric and real matrix, where the dynamics of the model is described as

$$\frac{d\boldsymbol{x}(t)}{dt} = -\boldsymbol{x}(t) + f(\boldsymbol{x}(t)) \tag{17}$$

for $t \geq 0$, and where

$$f(\boldsymbol{x}) = [\boldsymbol{x}^T \boldsymbol{x} \boldsymbol{A} + (1 - \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x}) \boldsymbol{I}] \boldsymbol{x}$$

and $\boldsymbol{x} = (x_1, ...x_n)^T \in R^n$ represents the state of the network and $\boldsymbol{I}$ is the identity matrix [7].

By using the model 17 and setting the derivative to zero, one finds the maximum/minimum eigenvectors of the matrix $\boldsymbol{A}$. After calculating the extreme eigenvectors of the matrix $\boldsymbol{A}$, it is possible to compute the corresponding eigenvalue to the eigenvectors by this relation

$$R(\boldsymbol{A}, \boldsymbol{x}) = \frac{\boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x}}{\boldsymbol{x}^T \boldsymbol{x}} = \lambda_{max} \vee \lambda_{min} \tag{18}$$

which is known as the Rayleigh quotient. It can be shown that, for a given matrix, the Rayleigh quotient reaches its minimum or maximum value, $\lambda_{min}/\lambda_{max}$, when $\boldsymbol{x}$ is $\boldsymbol{x}_{min}/\boldsymbol{x}_{max}$ (the corresponding eigenvector). This means that if one starts with a random initial vector $\boldsymbol{x}$, it should converge to the true maximum/minimum eigenvector of the matrix and then one can compute the true corresponding eigenvalue of that eigenvector.

III Implementation

Programs used in this project can be found on our github repository [5]. These programs are inspired by the codes found in the course's github repository [2].
The implementation of the neural network and automatic differentiation was done by using Tensorflow.

i Neural Network

i PDE solver

The problem we are studying is the equation of state 1. A trial solution that satisfies the boundary and initial conditions mentioned earlier is on the form

$$g_{trial}(x,t) = (1-t)\sin(\pi x) + x(1-x)tN(x,t,P). \tag{19}$$

Then we simply compute the partial derivatives of the trial solution,

$$\frac{\partial g_{trial}}{\partial t} \tag{20}$$

$$\frac{\partial^2 g_{trial}}{\partial x^2} \tag{21}$$

and to calculate the derivatives we use Tensorflows automatic differentation.

Thereafter we minimize the two expressions using the MSE as the cost function,

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{\partial g_{trial}}{\partial t} - \frac{\partial^2 g_{trial}}{\partial x^2}\right) \tag{22}$$

where we used the Adam as the gradient descent method for optimizing the MSE.

ii Eigenvalue solver

The way we implemented the neural network is at a similar pattern as the PDE case.

By utilizing equation 17 and setting the time derivative to zero, as mentioned before, we end up with the following expression

$$\boldsymbol{x}(t) = f(\boldsymbol{x}(t))$$

where $\boldsymbol{x}(t)$ is the output from the neural network.

By defining our trial solution $\boldsymbol{x}_{trial}$ as the output from the neural network and initializing it with $\boldsymbol{x}(0) = \boldsymbol{x}_0$, where it $\boldsymbol{x}_0$ is a random vector, we could compute $f(\boldsymbol{x}_{trial})$.

Thereafter, we minimized $\boldsymbol{x}_{trial}$ and $f(\boldsymbol{x}_{trial})$ using the MSE as the cost function

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\Big(\boldsymbol{x}_{trial} - f(\boldsymbol{x}_{trial})\Big) \tag{23}$$

where we used the Adam as the gradient descent method for optimizing the MSE.

IV Results

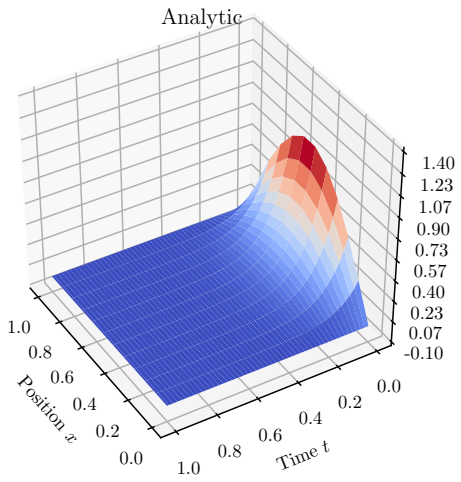


Figure 1: The three dimensional analytic solution to the PDE.

i $\Delta x = 0.1$

Table 2: Max absolute differences between numerical and analytical solutions for $\Delta x = 0.1$, evaluated at three different times. Crank-Nicolson is shown to produce the lowest errors.

Time	Forward Euler	Backward Euler	Crank-Nicolson	Neural network
0.5	$9.39 \cdot 10^{-3}$	$5.17 \cdot 10^{-3}$	$2.70 \cdot 10^{-3}$	$3.09 \cdot 10^{-3}$
1.0	$2.24 \cdot 10^{-4}$	$9.60 \cdot 10^{-5}$	$4.16 \cdot 10^{-5}$	$6.96 \cdot 10^{-5}$

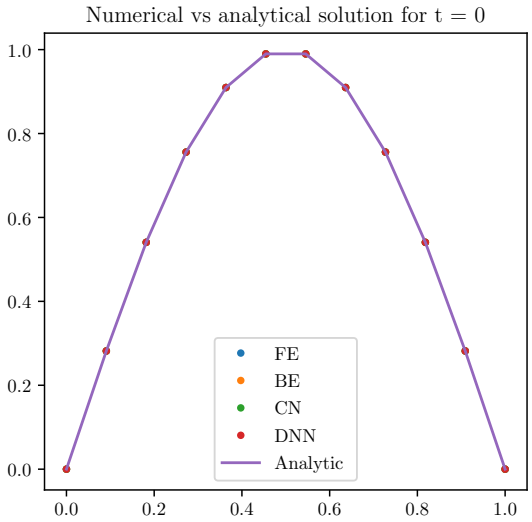


Figure 2: Comparison of the 4 PDE solvers with the analytical solution at time $t = 0$ and step size $\Delta x = 0.1$. The neural network is run with 10^5 iterations, no discernible difference is observed between the performance of any of the solvers.

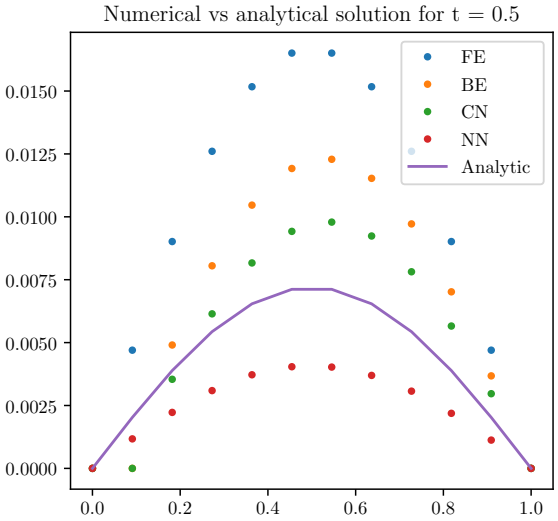


Figure 3: Comparison of the 4 PDE solvers with the analytical solution at time $t = 0.5$. Although the race is close between the solvers, Crank-Nicolson seems to barely win out compared with the others. NN iterations are $= 10^5$ and $\Delta x = 0.1$.

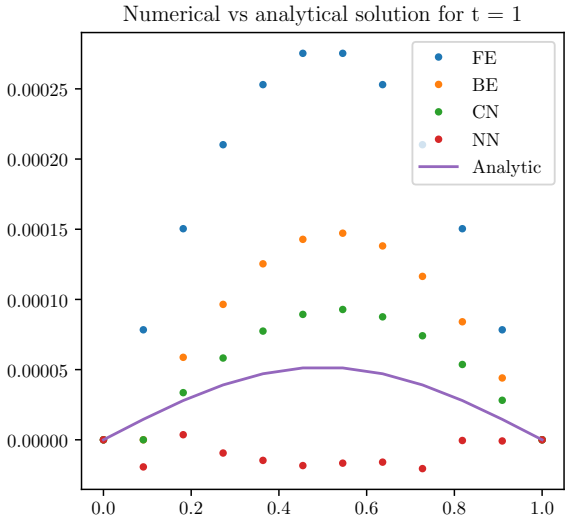


Figure 4: Comparison of the 4 PDE solvers with the analytical solution at time $t = 1$. We see a repeat of the standings from 3. Crank-Nicolson is yet again the best performer, with the other methods following close behind. NN iterations are $= 10^5$ and $\Delta x = 0.1$.

ii $\Delta x = 0.01$

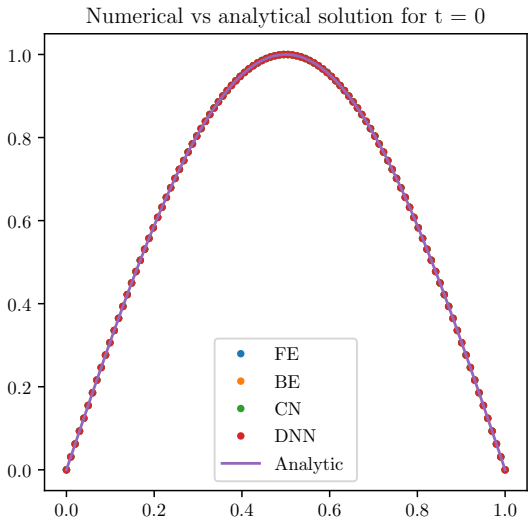


Figure 5: Comparison of the 4 PDE solvers with the analytical solution at time $t = 0.0$. NN iterations are $= 10^5$ and $\Delta x = 0.1$.

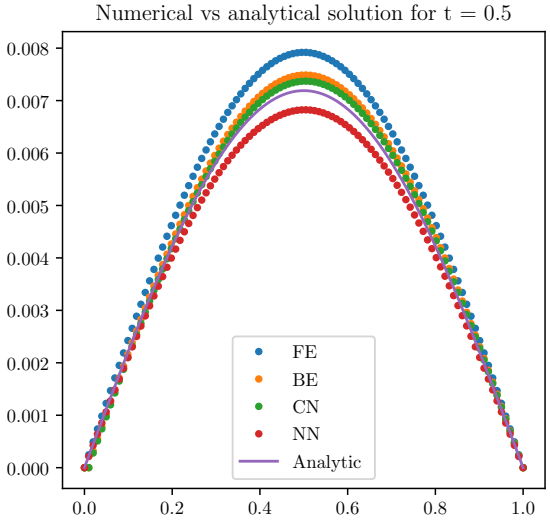


Figure 6: Comparison of the 4 PDE solvers with the analytical solution at time $t = 0.5$. Crank-Nicolsen shows the best fit, with Backward Euler on second. NN iterations are $= 10^5$ and $\Delta x = 0.1$.

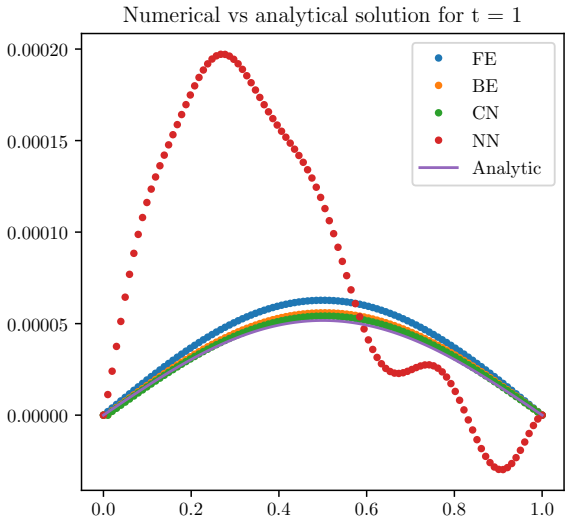


Figure 7: Comparison of the 4 PDE solvers with the analytical solution at time $t = 1.0$. Here we see some fluctuations for the neural network, but not of high amplitude as the errors are generally low. Crank-Nicolsen gives the smallest error here. NN iterations are $= 10^5$ and $\Delta x = 0.1$.

Table 3: Max absolute differences between numerical and analytical solutions for $\Delta x = 0.01$, evaluated at three different times. Crank-Nicolson is shown to produce the lowest errors.

Time	Forward Euler	Backward Euler	Crank-Nicolson	Neural network
0.5	$7.29 \cdot 10^{-4}$	$3.05 \cdot 10^{-4}$	$2.24 \cdot 10^{-4}$	$3.68 \cdot 10^{-4}$
1.0	$1.10 \cdot 10^{-5}$	$4.23 \cdot 10^{-6}$	$2.49 \cdot 10^{-6}$	$1.59 \cdot 10^{-4}$

i Eigenvalues

The known matrix is given as

$$\boldsymbol{A} = \begin{bmatrix} 2 & -1 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & -1 & 2 \end{bmatrix}$$

with the computed eigenvalues from Numpy shown in the legend of figure 9. The random matrix is given as

$$\boldsymbol{B} = \begin{bmatrix} 0.753 & 0.500 & 0.283 & 0.532 & 0.298 & 0.422 \\ 0.500 & 0.461 & 0.290 & 0.504 & 0.553 & 0.545 \\ 0.283 & 0.290 & 0.492 & 0.779 & 0.202 & 0.754 \\ 0.532 & 0.504 & 0.779 & 0.582 & 0.552 & 0.123 \\ 0.298 & 0.553 & 0.202 & 0.552 & 0.326 & 0.439 \\ 0.422 & 0.545 & 0.754 & 0.123 & 0.439 & 0.664 \end{bmatrix}$$

with the computed eigenvalues from Numpy shown in the legend of figure 10.

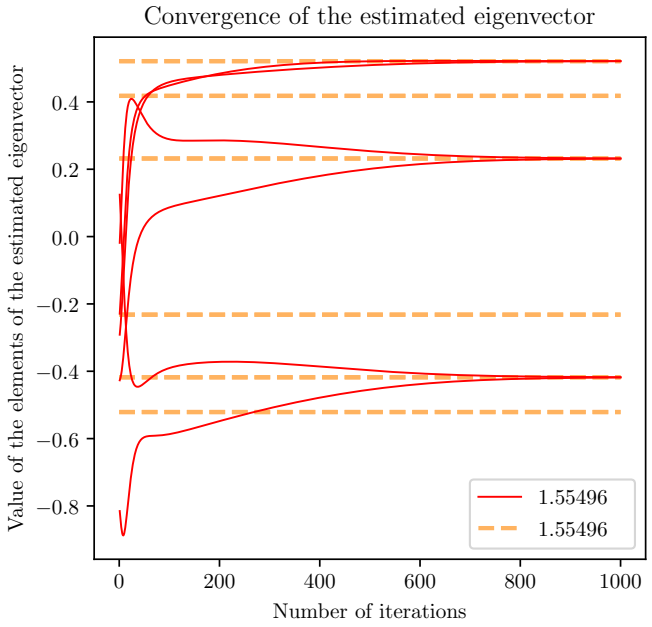
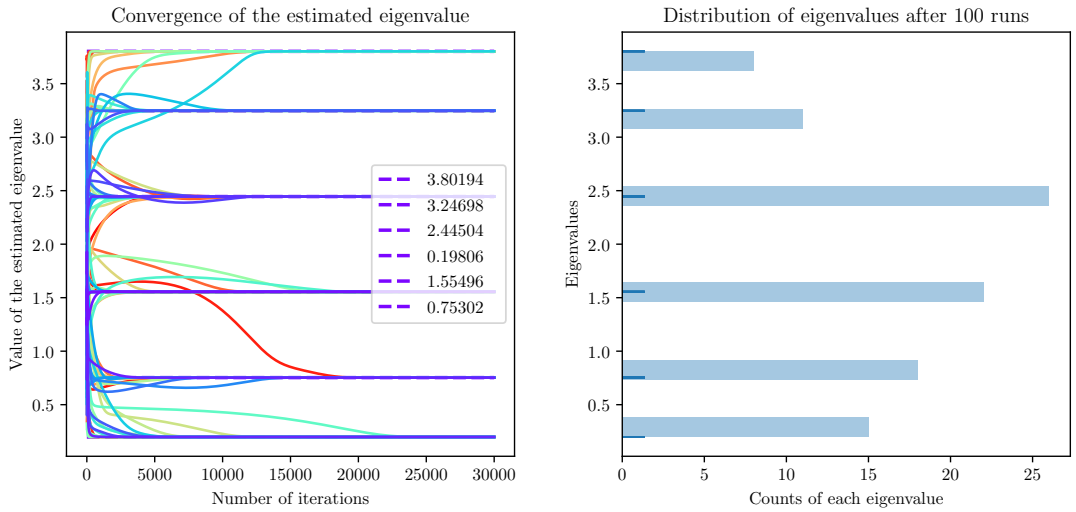
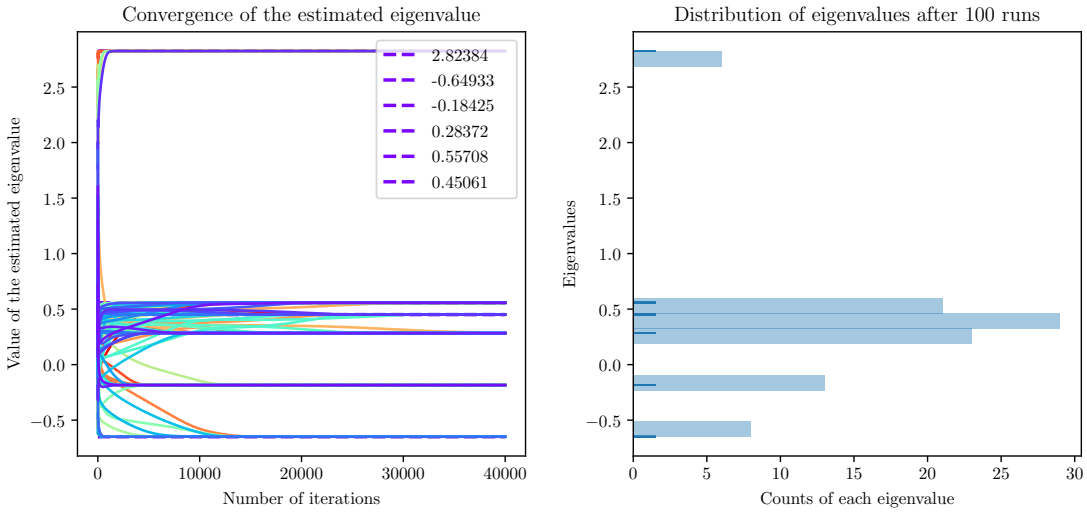


Figure 8: The convergence of the values in an eigenvector given a random initial vector. The dotted line is numpy's calculated eigenvalue, while the normal line is the neural network's estimated eigenvalue. In this case, the neural network found a convergence within 1000 iterations.



(a) (b)

Figure 9: (a) The convergence of 100 test-runs using one random initial vector and a known symmetric matrix. The dotted line is numpy's calculated eigenvalues, while the normal line is the neural network's estimated eigenvalues, resulting in a tight fit. (b) Showing the distribution of the 100 converged eigenvalues.



(a) (b)

Figure 10: (a) The convergence of 100 test-runs using one random initial vector with a random symmetric matrix. The dotted line is numpy's calculated eigenvalues, while the normal lines is the neural network's estimated eigenvalues, giving a tight fit. (b) Showing the distribution of the 100 converged eigenvalues.

V Discussion

i PDE solvers

For $\Delta x = 0.1$ in figure 2, we see that all models have a perfect fit. This is also expected, since the models reflects the initial condition. For the next time step in figure 3, $\Delta t = 0.5$, we can see that Crank-Nicolson performs best with the neural networks' solution breathing in its neck. This internal competition between the two solutions intensifies for the last time step as well, where the winning algorithm changes after every run.

For the first time step and with $\Delta x = 0.01$ at figure 5, the models once again have a perfect fit. When the time evolves, we see a bigger discrepancy. In figure 6, we can see that Crank-Nicolson has the best fit, closely followed by Backward Euler. This discrepancy gets reduced for the last time step in figure 7, as we see the lowest error for Crank-Nicolson. For this case, we can see that the neural network is giving the biggest error. However, the neural network has room for improvement and further optimization, as it could be training for longer a longer time. Where the normal algorithms does not change dramatically for every run, the neural network can be tuned to run for hours and hours with optimalization. Unfortunately, the available infrastructure for this project is limited, but it is indeed an interesting and future aspect to consider.

ii Eigenvalues

As seen in figure 8, the neural network does make the values of the elements in the estimated eigenvector converge towards to the values that are corresponding to the eigenvalue. However, it was given in the [7] that it should converge towards either a minimum or a maximum eigenvalue, but our neural network often converge to a local eigenvalue, as seen in figure 9. The distribution of the converged eigenvalues are equal, and the neural network does not favor either the minimum or maximum eigenvalues. This is both the case for a known symmetric matrix (9) and a random symmetric matrix (10).

One possible explanation to this might be that the neural network finds a local minimum, which an eigenvalue is, and does not further improve its approximation. Another reason might be that the optimization that is used is Adam, which changes the learning rate dynamically and after its best ability. However, Adam might choose a small learning rate when the eigenvalue have converged to a local minimum, and will get stuck because of a low learning rate. If we would tune this learning parameter when using gradient descent as optimizer instead of Adam, it might lead to different results. This is something one might indulge in on during further improvement of the model. A third option can also be the choice of neural network. The eigenvalues in [7] are found using a recurrent neural network, and not a normal deep neural network. As this is an entirely own type of neural network, it raises questions that are left unanswered in this report if this is what could have helped our neural network to converge towards the global minimum or maximum eigenvalues.

VI Conclusion

In this project, we have implemented three different algorithms named Forward Euler, Backward Euler and Crank-Nicolson to compete against a neural network's solution to test which algorithm can model the temperature gradient in a one-dimensional rod with the lowest error. Observing the first problem with $\Delta x = 0.1$ it was found that Crank-Nicolsen and the neural network solutions were the closest fit to the model. However, this changed when we implementing $\Delta x = 0.01$, where Crank-Nicolsen performed overall best. In addition, the neural network required a lot of memory and running time for novice computers that students possess.

For the second part of the project we wanted to use a neural network to find the minimum and maximum eigenvalues of a given matrix, as it was proposed it would in [7]. The neural network did find the eigenvalues, but did not converge solely towards the minimum or maximum eigenvalues. Some reasons for this might be either the use of optimizer, the type of neural network or a simple reason like not running for a long enough time.

For future improvements it would be interesting to have a proper infrastructure to investigate if it possible to tune the neural network beyond Crank-Nicolsen. It is of great importance to have an accurate approximation to partial differential equations. Often there is no analytical solution, and people's lives and projects might depend on accurate calculation. It is exceedingly motivating to explore the possibilities of new applications, such as a neural network.

Appendices

A Analytical solution (1 dim)

We start of with the PDE

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

with initial conditions, i.e., the conditions at $t = 0$

$$u(x,0) = \sin(\pi x), \quad 0 < x < L \tag{24}$$

with $L = 1$ the length of the x-region of interest. The boundary conditions are

$$u(0,t) = 0, \quad t \geq 0 \tag{25}$$

and

$$u(L,t) = 0, \quad t \geq 0. \tag{26}$$

Then we make an ansatz, where we assume that the function $u(x,t)$ can be seperated in terms of it's variables.

$$u(x,t) = F(x)G(t)$$

If we plug this into the equation above, we get

$$F(x)\frac{\partial G(t)}{\partial t} = G(t)\frac{\partial^2 F(x)}{\partial x^2}$$

Then we define the derivatives of the functions $F(x)$ and $G(t)$ as

$$\frac{\partial G(t)}{\partial t} = G'(t) \quad \& \quad \frac{\partial^2 F(x)}{\partial x^2} = F''(x)$$

Then, the equations can be rewritten as

$$F(x)G'(t) = G(t)F''(x)$$
$$\frac{G'(t)}{G(t)} = \frac{F''(x)}{F(x)}$$

where the derivative is with respect to x on the left hand side (lhs) and with respect to t on the right hand side (rhs). This equation should hold for all x and t . We must then, require the rhs and lhs to be equal to a constant. We call this constant $-\omega^2$. This gives us the two differential equations,

$$\frac{G'(t)}{G(t)} = -\omega^2 \quad \& \quad \frac{F''(x)}{F(x)} = -\omega^2$$
$$G'(t) = -\omega^2 G(t) \quad \& \quad F''(x) + \omega^2 F(x) = 0$$

The general solutions to these differential equations are trivial and are equal to

$$G(t) = Ce^{-\omega^2 t} \quad \& \quad F(x) = A\sin(\omega x) + B\cos(\omega x)$$

By applying the boundary conditions of $u(x,t)$, we get

$$\begin{aligned}
&u(x,t) = F(x)G(t) \\
&u(0,t) = F(0)G(t) = 0 \quad \rightarrow \quad \big(A\sin(\omega \times 0) + B\cos(\omega \times 0)\big)G(t) = 0 \quad \rightarrow \quad B = 0 \\
&u(L,t) = F(L)G(t) = 0 \quad \rightarrow \quad A\sin(\omega \times L)G(t) = 0 \quad \rightarrow \quad \sin(\omega \times L) = 0 \quad \rightarrow \quad \omega = \frac{n\pi}{L}
\end{aligned}$$

To satisfy the boundary conditions we require $B = 0$ and $\omega = \frac{n\pi}{L}$. One solution is therefore found to be

$$u(x,t) = F(x)G(t) = A\sin(wx)Ce^{-\omega^2 t} = A\sin\big(\frac{n\pi}{L}x\big)Ce^{-\frac{n^2\pi^2}{L^2}t} = A_n\sin\big(\frac{n\pi}{L}x\big)e^{-\frac{n^2\pi^2}{L^2}t}$$

where we define the constant $A_n = \frac{A}{C}$.

But there are infinitely many possible n values (infinite number of solutions). Moreover, the diffusion equation is linear and because of this we know that a superposition of solutions will also be a solution of the equation. We may therefore write

$$u(x,t) = \sum_{n=1}^{\infty} A_n\sin\big(\frac{n\pi}{L}x\big)e^{-\frac{n^2\pi^2}{L^2}t}$$

The coefficient A_n is in turn determined from the initial condition, which in this case is $u(x,0) = \sin(\pi x)$.

$$u(x,0) = \sin(\pi x) = \sum_{n=1}^{\infty} A_n\sin\big(\frac{n\pi}{L}x\big)e^{-\frac{n^2\pi^2}{L^2}\times 0} = \sum_{n=1}^{\infty} A_n\sin\big(\frac{n\pi}{L}x\big)$$

To satisfy the initial condition, where $L = 1$, we see that n must be $n = 1$ and $A_n = A_1 = 1$. Thus, our analytical solution looks like

$$\begin{aligned}
u(x,t) &= \sum_{n=1}^{\infty} A_n\sin\big(\frac{n\pi}{L}x\big)e^{-\frac{n^2\pi^2}{L^2}t} \\
u(x,t) &= \sin(\pi x)\cdot e^{-\pi^2 t}
\end{aligned}$$

References

[1] GitHub repository for project 3 https://github.com/ohebbi/FYS-STK4155/tree/master/project3

[2] Morten Hjorth-Jensen, Github Repository
https://github.com/CompPhysics/MachineLearning/tree/master/doc
http://compphysics.github.io/ComputationalPhysics/doc/web/course

[3] Morten Hjorth-Jensen, Github Repository
http://compphysics.github.io/ComputationalPhysics/doc/web/course

[4] Reiichiro S. Nakano. Metrics Module of scikit-plot. 2017. https://scikit-plot.readthedocs.io/en/stable/metrics.html

[5] GitHub repository containing all code and plots, as well as a Jupyter notebook for easy reproducibiliy https://github.com/ohebbi/FYS-STK4155/tree/master/project3

[6] Trevor Hastie, Robert Tibshirani, Jerome H. Friedman, The Elements of Statistical Learning, Springer https://www.springer.com/gp/book/9780387848570.

[7] Yi *et al.* Computers and Mathematics with Applications 47, 1155 (2004) https://www.sciencedirect.com/science/article/pii/S0898122104901101