Solving a diffusion equation using neural networks

Sakarias Frette, William Hirst, Mikkel Metzsch Jensen

(Dated: December 16, 2021)

CONTENTS

I. INTRODUCTION

Differential equations are a cornerstone in understanding the physical processes around us. From the Schrödinger equation, which explains low energy quantum mechanical systems, to the Navier-Stokes equations that governs flow of fluids, they explain how the system impacts itself through change with respect to time, space or both. Traditional numerical solutions such as finite difference[3] and finite element[4] have yielded good results, and they are computationally fast, but they lack in one aspect. They base themselves on discretization, and so the solution you get is also discretized (not true for finite element). Thus, you do not get a function with which can be evaluated at any given point. with the Neural ordinary differential equation paper[1] from 2018 and other papers, it was shown that neural networks also can solve both ODE's and PDE's. With these networks one can get a actual function that can be evaluated, though at a potential cost of computation time.

It was also shown in 2003[5] that neural networks can be used to solve eigenvalue problems. Eigenvalue problems occur in many physical phenomenon, such as the Schrödinger equation, where the energy of the system is the eigenvalue of the Hamilton operator, or in vibration analysis, where the eigenvalue corresponds to the eigen-frequency.

This is the motivation for this report, where we will use both finite difference and machine learning to solve the diffusion equation and compare computation time and error, as well as compute eigenvalues for a given matrix using machine learning.

The report is structured in the sections theory, implementation results and discussion, and conclusion. The theory includes the mathematics behind the diffusion equation, the finite difference method, neural networks for solving PDE's and eigenvalue problems using neural networks. The implementation includes the pseudo code algorithms for the finite difference method, the neural network PDE solver, as well as some of the choices used to solve the eigenvalue problem. The results and discussion includes results from the finite difference method, the neural network for the PDE and the eigenvalue problem, as well as discussion comparing the methods. We finally summarize our findings in the conclusion.

II. THEORY

A. The diffusion equation

The generalized diffusion equation is given as

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

where u is the solution, $\mathbf{x}$ is a 3D vector containing spacial coordinates, t is temporal coordinate, and alpha is a function comprised of the temporal and spacial derivative constants and possibly the solution u. For this project we will look at a simpler example, namely the 1D diffusion equation with $\alpha = 1$. It is given as

$$\frac{\partial}{\partial t}u(x,t) = \frac{\partial^2}{\partial x^2}u(x,t), \tag{1}$$

and in our case it is defined in the domain of

$$x \in \Omega : [0,1]$$

for $t > 0$, with the following initial condition

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

and with Zero-Dirichlet boundary conditions

$$u(0,t) = u(L,t) = 0\ ;\ t \geq 0.$$

1. *Analytical solution*

The 1D diffusion equation 1 can be solved analytically, which can serve as a test for numerical solutions. By considering the specifics of the diffusion equation one can derive a solution on the following form that will satisfy equation 1:

$$u_e(x,t) = A\sin(kx)e^{-\omega t}, \tag{2}$$

where we have introduced three degrees of freedom: A, k and ω. A is the amplitude of the sinusoidal function, k is the angular frequency of the spacial domain and ω is the rate of decay. By inserting the proposed solution proposed solution 2 into the diffusion equation 1 we find the derivatives

$$\frac{\partial^2 u_e(x,t)}{\partial x^2} = -k^2 A\sin kx e^{-\omega t},$$
$$\frac{\partial u_e(x,t)}{\partial t} = -\omega A\sin(kx)e^{-\omega t},$$

which leads to the condition

$$\omega = k^2. \tag{3}$$

By expression ω in terms of k according to 3 and implementing the zero Dirichlet boundary condition we can determine k as

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

Thus we must have $kL = n\pi$, for any integer $n = \{0, \pm1, \pm2, \ldots\}$ such that

$$k = \frac{n\pi}{L}$$

Finally we can determine A by the use of the initial condition. With the use of the previous findings of ω and k we get

$$\begin{aligned} u_e(x,0) &= \sin(\pi x) \\ A\sin(\frac{n\pi}{L}x) &= \sin(\pi x) \end{aligned}$$

By choosing $n = 1$ and $L = 1$ we find $A = 1$.
Our final, analytical solution is then

$$u_e(x,t) = \sin(\pi x)e^{-\pi^2 t}. \tag{4}$$

2. *Finite difference discretization*

One common approach to solving the diffusion equation numerically is by the use of finite difference method. By discretizing the space and time domain respectively as

$$\begin{aligned} x &= i\Delta x, & i &\in \{0,1,2,\ldots,N_x\}, & N_x &= L/\Delta x + 1, \\ t &= n\Delta t, & n &\in \{0,1,2,\ldots,N_t\}, & N_t &= T/\Delta t + 1, \end{aligned}$$

for a total time T, we can approximate the derivatives as finite differences. By convention we will use subscript indexing for spacial coordinates and superscript indexing for temporal coordinates such that the discretized solution at a given point is written $u(x_i, t_n) = u_i^n$. For the first order temporal derivative we will use the so-called explicit forward Euler scheme, while for the second order spacial derivative we use the second order central difference, that is

$$\begin{aligned} \frac{\partial u(x,t)}{\partial t} &= \frac{u_i^{n+1} - u_i^n}{\Delta t} + \mathcal{O}(\Delta t) \\ \frac{\partial^2 u(x,t)}{\partial x^2} &= \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} + \mathcal{O}(\Delta x^2). \end{aligned}$$

Inserting the approximations into the diffusion equation 1 we get the discretized equation

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

which can be rewritten on the form

$$u_i^{n+1} = u_i^n(1 - 2C) + C(u_{i+1}^n + u_{i-1}^n). \tag{5}$$

for $C = \Delta t/\Delta x^2$. Equation 5 has a stability criteria, formalised by von Neuman[2], given as

$$\alpha C \leq \frac{1}{2}, \tag{6}$$

where α is the parameter multiplied with the spacial part of the diffusion equation. In our case $\alpha = 1$, which allows for reasonably choice step sizes, while for large α this method requires an extensive fine discretization of the time domain.

B. Neural networks and PDE's

Another, but yet very different, approach to the numerical solution is by the use of a neural network.

1. *Trial function*

We begin by constructing a trial function $g_t(x,t)$ that will serve as the final prediction from our model. That is, for a well trained model the trail function will resemble the exact solution. We define the generic trail function as

$$g_t(x,t) = h_1(x,t) + h_2(x,t)N(x,t,P), \tag{7}$$

where $N(x,t,P)$ is the output from the neural network as a function of space coordinate x, time coordinate t and weights and bias P, while $h_1(x,t)$ and $h_2(x,t)$ are functions that ensures that initial conditions and boundary conditions are satisfied. We have zero-Dirichlet conditions on the boundaries, so we might choose a function like $x(1-x)$ which will ensure that we get zero on the boundaries. In addition we want to satisfy the initial condition such that $h_1(x,0) = u(x,0)$ and $h_2(x,0) = 0$. One way of achieving this is by the functions

$$\begin{aligned} h_1 &= (1-t)u(x,0) = (1-t)\sin(\pi x) \\ h_2 &= x(1-x)t \end{aligned}$$

A side effect of this is a linear decrease of h_1 and a linear increase of h_2. However this is no problem since $N(x,t,P)$ should simple adjust accordingly. By combining the expressions of h_1 and h_2 the trial function reads

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

2. Cost function

When using the neural network for supervised learning we essentially wants an output $N(x,t,P)$ for which the trial functions satisfy the PDE as good as possible. Thus we must define a cost functions that assures this. Since it is not usual to know the exact solution we will instead use the residual. The residual is defined as putting all the terms on one side of the equation, and in the case of the diffusion equation, we have the residual

$$R = \frac{\partial u(x,t)}{\partial t} - \frac{\partial^2 u(x,t)}{\partial x^2}, \tag{8}$$

which should be equal to zero in the whole domain for a perfect solution of the PDE. Thus we choose the cost function the be the mean squared residual

$$C(x,t) = \langle R^2 \rangle = \left\langle \left(\frac{\partial u(x,t)}{\partial t} - \frac{\partial^2 u(x,t)}{\partial x^2} \right)^2 \right\rangle. \tag{9}$$

What we have here is a minimization problem which can be used to optimize the weights and biases of the network. The practical aspect of evaluating the cost function is further explained in the implementation section III.

C. Finding eigenvalues through a neural network

The work of Yi et al. [5] proposed a method to compute the eigenvectors of a symmetric matrix A using a neural network. The idea is to define the eigenvectors of A as solutions of a simple ODE. In this section we will briefly describe the essentials needed to implement such a method.

For a given matrix A, an eigenvector v with eigenvalue λ must satisfy the eigenvalue equation

$$Av = \lambda v. \tag{10}$$

For a vector x it follows from 10 that

$$x^T x A x - x^T A x x = 0, \tag{11}$$

if and only if $x = v$ or $x = \mathbf{0}$. By choosing equation 11 as the cost function, one could alternatively solve the eigenvalue problem with respect to x and subsequently solving for the eigenvalue using equation 10 and 11 as

$$\begin{aligned} x^T x \lambda x - x^T A x x &= 0 \\ x^T x \lambda - x^T A x &= 0 \\ x^T x \lambda &= x^T A x \end{aligned}$$

$$\iff$$

$$\lambda = \frac{x^T A x}{x^T x}, \tag{12}$$

However, in order to avoid that the solution gravitates towards the trivial solution $x = \mathbf{0}$, we instead use the following ODE (suggested by Yi et al. [5]).

$$\frac{dx(t)}{dt} = f(x) - x(t), \tag{13}$$

where

$$f(x) = [x^T x A + (1 - x^T A x) I] x, \tag{14}$$

A is a symmetric matrix, I is the identity matrix and t is just a variable.

We can study the derivative of of the length

$$\begin{aligned} \frac{d(x^T x)}{dt} &= 2x(t)\frac{dx(t)}{dt} \\ &= 2x(f(x) - x) \\ &= 2x(x^T x A x - x^T A x x) = 0 \end{aligned} \tag{15}$$

where we used equation 11 and 13 for the last transition. From 15 we see that the derivative of the length must be constant, and by substituting $x(t)^T x(t) = x(0)^T x(0)$ into the ODE (13) we arrive at

$$\frac{dx(t)}{dt} = x(0)^T x(0) A x - x^T A x^T x, \tag{16}$$

which is the final formulation of the ODE that we are going to use for the solving of the eigenvalue problem. At this point we state the following theorem

Theorem 1. *Given an initial state, x(0) every solution of 16 will converge towards an eigenvector of A corresponding to the largest eigenvalue, v_{max} when $t \to \infty$ given that the following is true*

- *x(0) is non-zero,*
- *x(0) is not orthogonal to the v_{max}.*

For the proof of this theorem we refer again to the text of Yi [5], specifically theorems 3 and 4 on pages 1159 and 1160.

III. IMPLEMENTATION

The code can be found on Github.

A. Solving a PDE using finite difference

Before we solve our PDE with the neural network, we first want to solve it using an explicit solver. The solver sets up a recursive algorithm to step forward in time using equation 5. First, we initialize all arrays and constants, and insert the initial condition from II A. Then, for each time step we calculate the inner points of the solution, as the boundary points are fixed. After the boundary points are inserted, we update the arrays u and u_1, and insert u into $u_{complete}$. The algorithm is outlined in algorithm 1, while the class can be found in ExplicitSolver.py (see github) .

Algorithm 1: Finite difference solution

```
Initialize arrays for complete solution array, u_complete ;
Initialize x and t domain;
Initialize array for solution at t, u ;
Initialize array for solution at t − Δt, u_1;
Insert initial condition ;
for 1 → N_t do
    Calculate the inner spacial points for u;
    Insert boundary conditions;
    Update u, u_1;
    Insert solution to u_complete;
end
Return u_complete
```

Note here the use of three arrays, namely the final solution $u_{complete}$, u and u_1. $u_{complete}$ stores the solution for every combinations of x and t. u_1 stores the values of the previously loop and is set equal to the initial condition, II A. u stores the values calculated in the current iteration.

(GO THROUGH THIS AGAIN).

B. Solving a PDE with an neural network

For the setup of the neural network we decide to use a TensorFlow infrastructure. The solver is implemented as a class in ML_PDE_solver.py (see github). The main structure of the code is divided into three parts; creating the neural network model, training of the model and tracking the process.

1. Model

The model is build accordingly to a fixed network architecture as shown in table I.

Table I: Network architecture, for each hidden layer showing the number of number of hidden nodes, the number of tunable parameters given by the weights and biases, and the activation function.

Layer	Nodes	Params	Activation
1	20	60	Sigmoid
2	20	420	Sigmoid
3	20	420	Sigmoid
Output	1	21	None

This specific architecture was decided upon after an extensive testing of the network performance on the PDE. The data, which is feed into the first hidden layer of the model, has two features, namely position, x, and time, t. Since we want the network to predict the solution in the whole spacial and temporal domain we need to train on every combination of the discretized spacial and temporal points. To achieve this, we create a design matrix given as

$$X = \left.\begin{bmatrix} t_0 & x_0 \\ t_0 & x_1 \\ \vdots & \vdots \\ t_1 & x_0 \\ t_1 & x_1 \\ \vdots & \vdots \\ t_{N_t} & x_{N_x} \end{bmatrix}\right\} (N_x \cdot N_t). \tag{17}$$

2. Training

The training of the network is executed as a custom training, since TensorFlow's own training functionality can not easily adapt to the cost function of this problem. Hence, for each epoch we calculate the loss and gradient of the loss with respect to model parameters using TensorFlow GradientTape. The parameters are then updated using the ADAM optimizer, which uses an adaptive momentum. The ADAM optimizer is ideal in situations where the model gets stuck at a local minimum. Given such a situation the ADAM optimizer applies a small increase in the learning rate in an attempt to "jump" out of the minimum. The algorithm is outlined in algorithm 2.

Algorithm 2: Training of the network.

```
Initialize model ;
for epoch in epochs do
    Calculate the loss and the gradient of the loss with respect to model parameters ;
    Update model parameters using ADAM optimizer ;
    Store loss;
end
```

C. Eigenvalue and eigenvectors

In order to calculate eigenvalues and eigenvectors we first need to specify the matrix A of interest. In this report we will limit ourselves to a symmetric 6×6-matrix. We generate first a 6×6 matrix Q by drawing random points from a standard normal distribution. We insure that the final matrix A is symmetric by defining it as

$$A = \frac{Q^T + Q}{2}. \tag{18}$$

From this point on we can define the ODE according to equation 13 and solve for the eigenvectors similar to what we have done for the diffusion equation III B.

The main distinction from III B is the sizes of the input and output layer, and the cost function. For the PDE case the input was two-dimensional (for x and t) and the output was a single number. In our ODE case we only have one variable, t and our output is a 6-dimensional vector. Therefore the input layer had one node and the output layer has nodes 6.

As for the cost function we will again use the residual of our equation 16 given as

$$C(x) = x(0)^T x(0) Ax - x^T A x^T x - \frac{dx(t)}{dt}$$

Finally we must also satisfy the criteria in theorem 1. Firstly, as we have no pre-knowledge on v_{max}, there is no way of guarantying that our choice of $X(0)$ will not be orthogonal to v_{max}. Therefore we instead randomly generate the values of $X(0)$, again sampling from a "standard normal" distribution, in order to get some values that fit the criteria. Secondly, we must also only approximate the convergence for $t \to \infty$. We do so by defining a large enough t-domain as $t \in [0, 10^4]$.

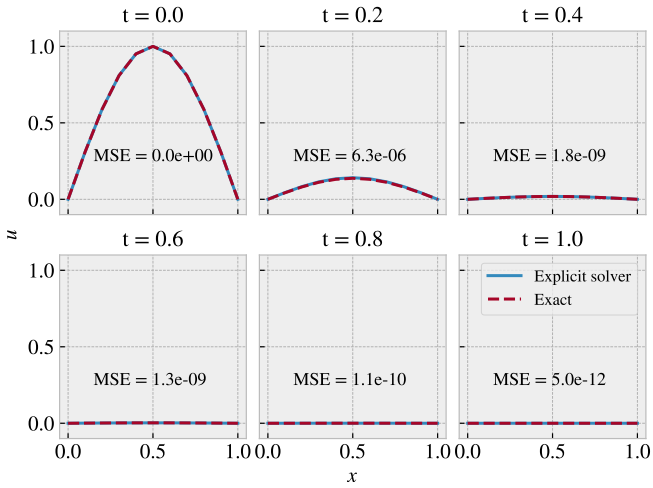
IV. RESULTS AND DISCUSSION

A. Diffusion Equation

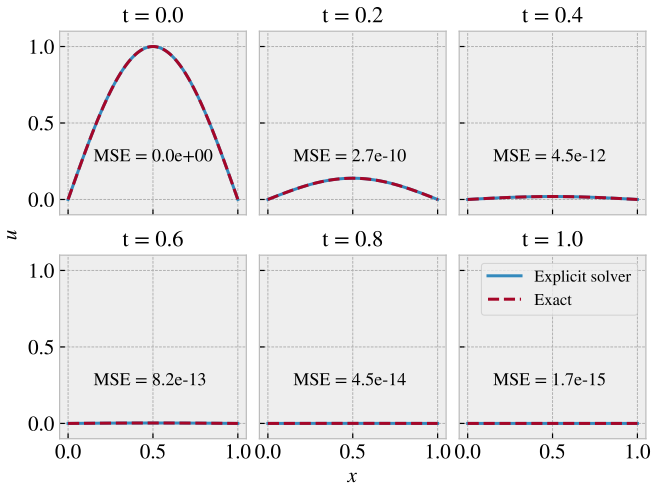
In this section we address the numerical solutions of the diffusion equation 1 discussed in section II A using finite difference and the neural network respectively.

1. *Finite Difference*

We are going to investigate the explicit solution for two spatial step sizes: $\Delta x = 0.1$ and $\Delta x = 0.001$. For the time step we choose $\Delta t = \Delta x^2/2$ according to the von Neuman condition stability criteria in equation 6. The resulting numerical solution is shown against the exact solution for six selected time points in figure 1.



(a) $\Delta x = 0.1$, $\Delta t = 5 \times 10^{-3}$.



(b) $\Delta x = 0.01$, $\Delta t = 5 \times 10^{-5}$.

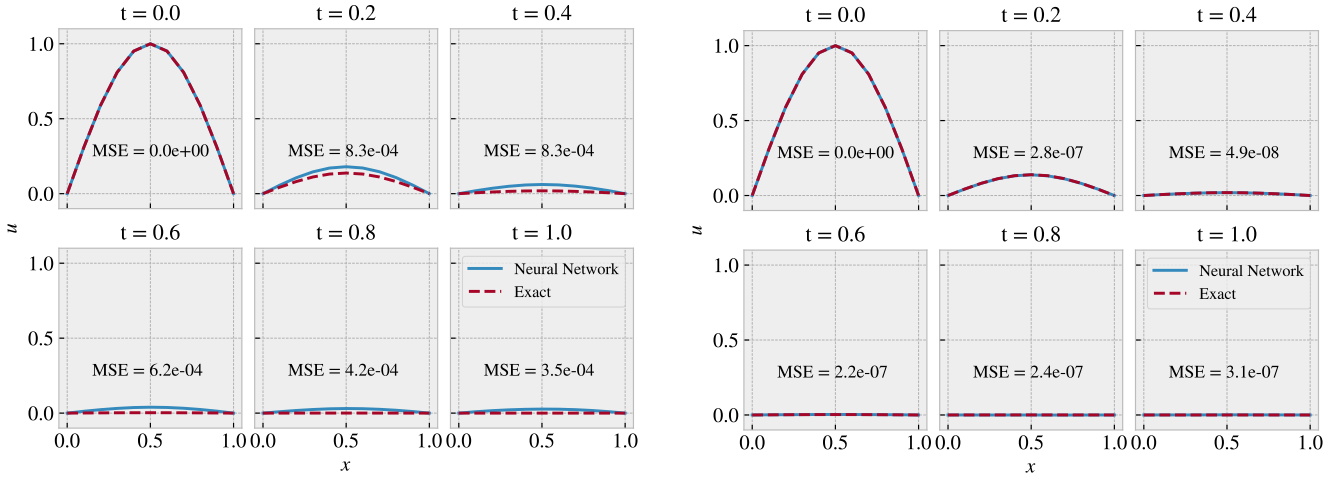
Figure 1: Explicit solution vs. exact solution for the diffusion equation 1 with $L = 1$ and $T = 1$ for different combinations of Δx and Δt. MSE denotes the mean squared error between the numerical and analytical solution.

From the results in figure 1 we observe immediately a relatively strong fit between the numerical and analytical solution. As expected, we see that the numerical solution is improved when decreasing Δx from $\Delta x = 0.1$ (1a) to $\Delta x = 0.01$ (1b). When inspecting each time frame we first observe in both cases that the solution is exact for $t = 0$ which is also expected since the numerical solution takes a starting point from the initial condition. Secondly, we observe that, among the six time points, the error peaks at $t = 0.2$. We know that the analytical solution decays from a sine curve into a horizontal line, for which the change of the solution is more radical at the beginning of the solution. Thus it is not so

strange that the numerical solution will miss the most in the first part of the time domain. As time goes on and the solution approach the horizontal line we see that the MSE becomes extremely small, which supports the idea of the error being linked to the rate of change of the analytical solution.

2. *Neural Network*

Next we investigate the numerical solution using the neural network approach. We use the network architecture as described in the implementation section III B, and a learning rate of 0.05 which proved to be a sturdy choice. Naturally we want to start off by using the same discretization as used in the finite difference analysis, namely $\Delta x = 0.1$ and $\Delta x = 0.01$ with $\Delta t = \Delta x^2/2$ corresponding to the von Neumann stability criteria. However, by running the computation with both the Neumann stability criteria and equal step size $\Delta x = \Delta t$ we saw that the equal step size performed significantly better. This is shown in figure 2 for $\Delta x = 0.1$ and 2000 epochs.



(a) Neumann stability criteria: $\Delta x = 0.1$, $\Delta t = 5 \times 10^{-3}$.

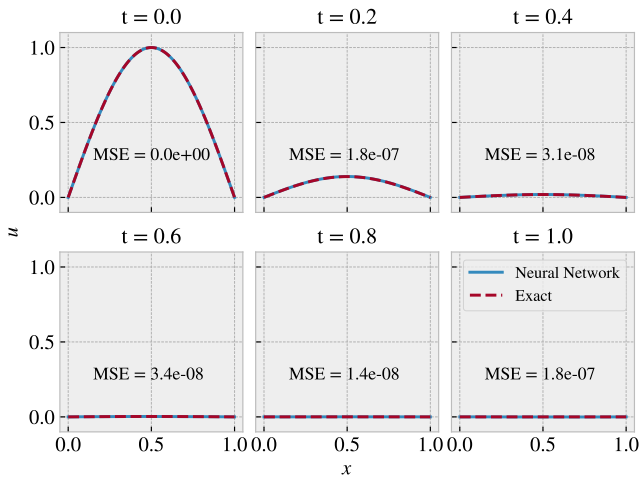
(b) Equal step size : $\Delta x = \Delta t = 0.1$.

Figure 2: Neural network solution vs. exact solution for the diffusion equation 1 with $L = 1$ and $T = 1$. The discretization is done with $\Delta x = 0.1$ combined with $\Delta t = 5 \times 10^{-3}$ and $\Delta t = 5 \times 10^{-3}$ respectively. We used 2000 epochs with a learning rate of 0.05. MSE denotes the mean squared error between the numerical and analytical solution at a given time point.

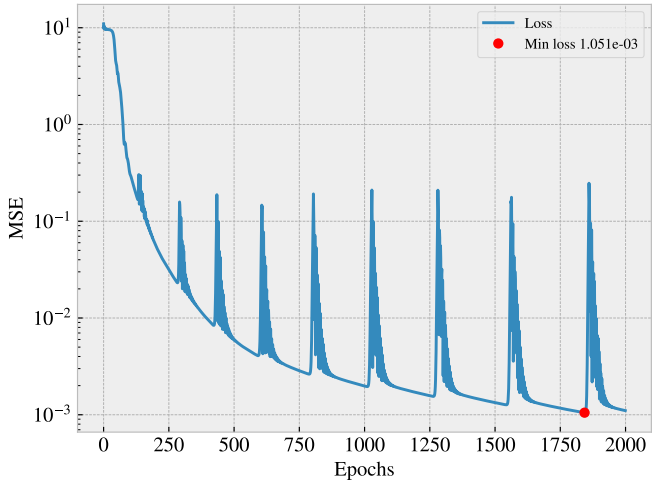
It is somewhat surprising that the equal step sizes (2b) produced a better result than the one with the Neumann criteria (figure 2a), given that the latter case has the finer discretization. A possible explanation is that the training of the network becomes somehow biased when we have an uneven number of spacial and temporal points. We did not look further into this, but it is an interesting topic for future investigation. Given this result we will use equal step sizes for the future neural network solutions.

When considering the result from the case of equal step sizes (2b), we see that the MSE is quite equally distributed across the chosen time frames ($t = 0.2, 0.4, 0.6, 0.8, 1$) with an MSE at roughly 10^{-7}. For the finite difference case using $\Delta x = 0.1$, $\Delta t = 5 \times 10^{-3}$ we saw a much higher difference in the error across the various time frames. This leads to the fact that the neural network performs slightly better for the first part of the time domain (where the solution changes more drastically) shown at $t = 0.2$, but worse for the remaining part of the time domain.

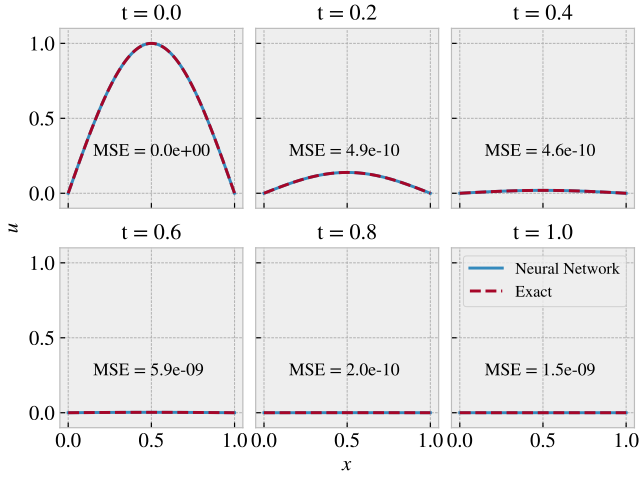
We also computed the numerical solution using $\Delta x = \Delta t = 0.01$. By continuing to use 2,000 epochs we did not see a great difference in the precision. Thus we ran the computation with 20,000 epochs in addition. Both results is shown in figure 3 together with the learning history, which show the mean squared residual as a function of epochs.



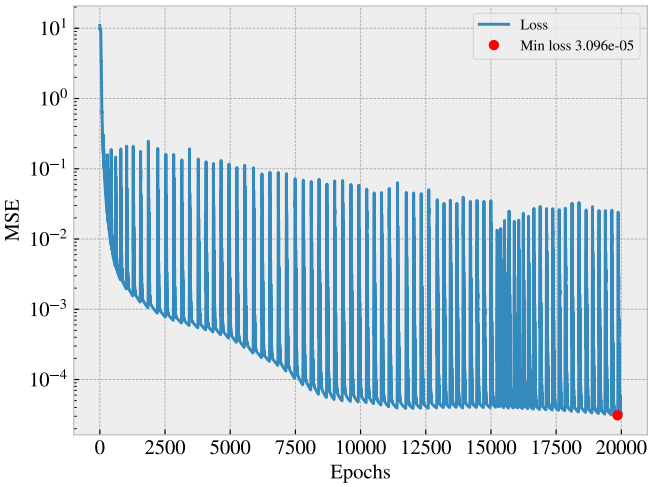
(a) Numerical solution, 2000 epochs



(b) Learning history, 2000 epochs



(c) Numerical solution, 20,000 epochs



(d) Learning history, 20,000 epochs

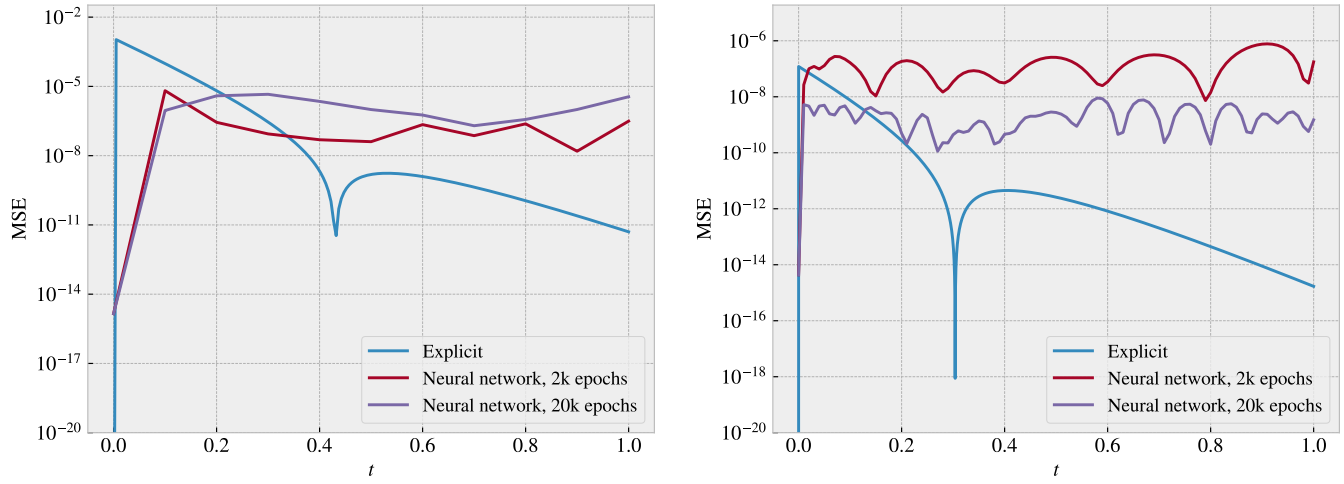
Figure 3: Neural network solution vs. exact solution for the diffusion equation 1 with $L = 1$ and $T = 1$ using $\Delta x = \Delta t = 0.01$. We used 2,000 and 20,000 respectively epochs with a learning rate of 5×10^{-2}. The left column shows the solution at different timestamps and the right column shows the learning history (loss) given as the mean squared error (residual) for each epoch.

From figure 3 we again observe that the error is quite evenly distributed across the various time frames. By increasing from 2000 to 20,000 epochs we get a solution that is competitive with the finite difference result at $t = 0.2$ but lacks behind quite severely at the remaining time points. Note that some runs gave an unfortunate stopping point right after the perturbation from the ADAM optimizer. In that case we simply rerun as we did not bother to implement any automatic response to avoid this. We could possible improve the neural network performance by running even more epochs, but since the computation time already far over exceeds the time used for the finite difference, we did not want to pursue this. Instead we will investigate the trade-off between error and computation time in more detail in section IV A 3. When considering the learning history for 2000 epochs (3b) we notice a descending trend, similar to an exponential decay, with a seemingly regular pattern of saw tooth spiked perturbations. These spikes match the expected effect from the ADAM optimizer which apply a disturbance to the model parameters in order to escape any eventual local minimum in the training process. When looking at the learning history for 20,000 epochs (3d) we see that initial exponential decay trend is interrupted (first time around 5000 epochs) as the MSE drops considerable more.

3. *Extended comparison of explicit and neural network*

To compare the numerical methods in more detail, we compute the MSE as a function of time for the two

methods explicit and neural network respectively. We compute the results for both 2,000 and 20,000 epochs as shown in figure 4.



(a) $\Delta x = 0.1$.

(b) $\Delta x = 0.01$.

Figure 4: Comparison of MSE for explicit and neural network vs exact solution as a function of time. We use common settings of $\Delta x = 0.01$, $L = 1$ and $T = 1$, and use $dt = dx$ for the neural network and $dt = dx^2/2$ for the finite difference solution. For the neural network we used a learning rate of $\eta = 0.05$ and both 2×10^3 and 2×10^4 epochs. We fixed a lower limit on the y-axis at 10^{-20} to improve the readability of the plot.

From figure 4 we see that the finite difference solution is dominant in the majority of the time domain. The only exception is the first interval $t \approx [0, 0.1]$. For the $dx = 0.1$ case (4a) we also notice that increasing the number of epochs actually turned out to give a mainly worse result. This could easily be due to an unfortunate stopping point combined with the effect from the ADAM optimizer, but we did not spend more time to unravel this.

As we computed the result for figure 4 it was apparent that the finite difference solution was significantly faster computational wise. Hence one could speculate whether this lead in computation time can be utilized in the lowering of the step sizes, such that the finite difference might be able to produce a better solution with respect to used computation time. In other words, we want to investigate which method yields the lowest error with respect to computation time used. Since we already know that the finite difference computation is faster than the neural network in the calculations for figure 4 the only possible domain for which the neural network might be better is at the early time points. Hence we pick $t = 0.05$ as our point of interest, and run the two solvers for different choices of Δx, still using $\Delta x = \Delta t$ for the neural network and $dt = dx^2/2$ for the finite difference. In addition we try various number of epochs for the neural network. We record the computation time for the various combinations and plot the MSE as a function of computation time. The result is shown in figure 5.

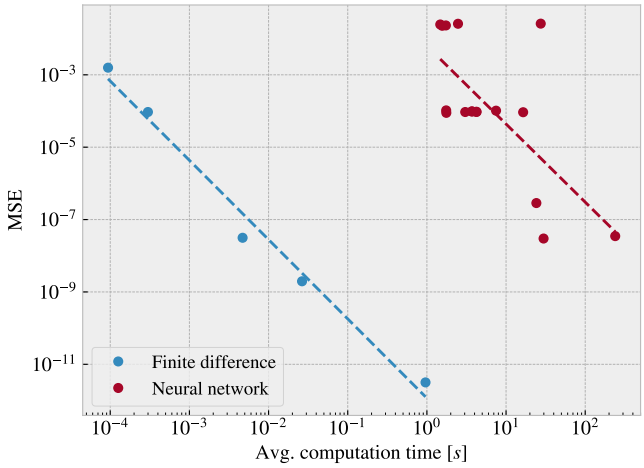


Figure 5: Mean square error (MSE) between numerical and analytical function for $t = 0.05$ for the diffusion equation 1 with $L = 1$. To give a rough estimate of the trends we included a linear regression on the logarithmic scale.

From figure 5 we see that the data points corresponding to the two solvers gathers around two distinct areas. The closer the points are to origo (0,0), the better the performance, and since there is quite a severe gap between these two areas, the finite difference method appears to be the most efficient solver. By performing linear fits on the logarithmic plot we attempt to describe the relation between MSE and computation as a power law, on the

form $f(x) \propto x^a$. For the finite difference case we get a fairly good linear fit, in contrast to the neural network case where the data points seem to be more scattered. However, by considering these fits anyway it appears that the MSE decay with a similar rate for both methods. Thus it does not show any signs of the neural network efficiency being better for if we considered solutions for even higher computation time. One might find a steeper fit for the neural network area by excluding certain outliers, corresponding to bad choices of number of epochs. However it still seems unreasonable to expect that the neural network could achieve the better efficiency in practice, since the point of overtaking the finite difference would be at an extremely low error and high computation time. Thus it is clear that the finite difference is superior in terms of optimizing the solution for the specific diffusion equation 1. But this might not be the case for a more complex problem involving more dimensions, such as many-body problems. This is yet another topic of interest for future studies, for which ine could repeat a similar analysis for more complex PDE's.

Another fact that should be mentioned is that the performance of the neural network regarding MSE vs computation time could be affected by optimizing the code. However, since we use TensorFlow, it is unrealistic to speculate in a code speed up that could close the gap between the finite difference solver and the neural network as seen in figure 5.

B. Eigen value problem

In this section we have studied the the eigenvalue problem discussed in section II C. The optimizer, number of hidden layers and nodes used in this section are the same as in the previous sections.
We generate a random symmetric 6×6-matrix and run the network over 200 epochs with an initial learning rate of 5×10^{-4}. In figure 6 we plot the evolution of the eigenvalue found by the network over each epoch. The evolution is visualized together with the eigenvalues found by the standard eigenvalue solver.

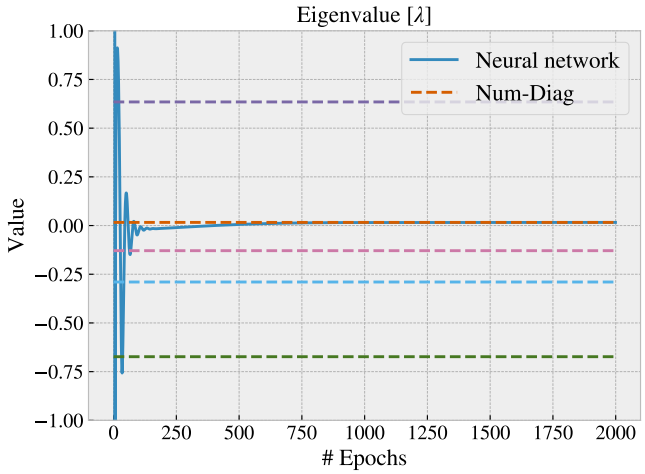


Figure 6: Dashed lines represent all 6 eigenvalues of the randomly generated matrix as calculated by numpy's eigen-solver. Solid line shows the neural networks calculated eigenvalue over each epoch.

Figure 6 displays the eigenvalue oscilating drastically in the first 100 epochs. After 100 epochs the network shows a clear convergence towards one of the eigenvalues of A. After 2000 epochs the network produces a $\lambda = 0.01618$. Compared to the the standard eigenvalue solver, $\lambda = 0.01611$ the network produces an error of $7 \cdot 10^{-5}$, which is $\approx 0.05\%$.
As discussed in section II C, we calculated the eigenvector in order to find the eigenvalue. Figure 7 shows the evolution of the eigenvector to the corresponding eigenvalue presented above, over all epochs.

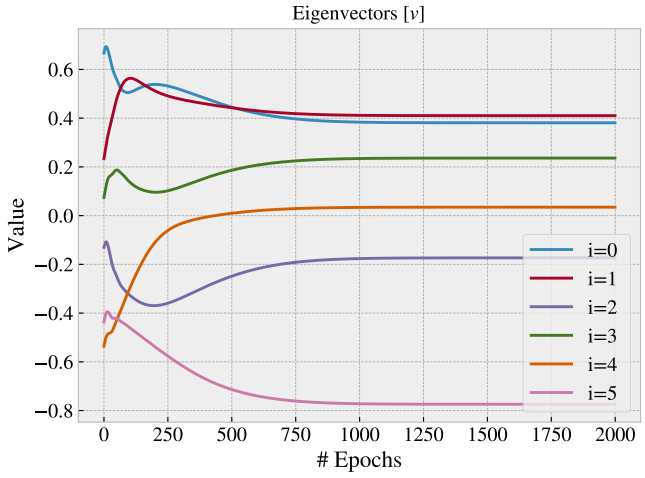


Figure 7: The lines in the figure show the evolution over 200 epochs of the values of each index in the eigenvector produced in the network.

In figure 7 we observe that the network converges towards a stable solution. Contrary to the eigenvalues the elements of the eigenvector do not exhibit the same os-

cillations, but instead converges towards a solution in a much more stable fashion. The final eigenvector produced after 200 epochs is

$$\lambda^{NN} = \begin{bmatrix} -0.381 \\ -0.410 \\ 0.173 \\ -0.236 \\ -0.035 \\ 0.774 \end{bmatrix} \tag{19}$$

, and the vector produced by numpy's eigenvalue solver is

$$\lambda^{ND} = \begin{bmatrix} -0.381 \\ -0.410 \\ 0.173 \\ -0.236 \\ -0.034 \\ 0.774 \end{bmatrix} \tag{20}$$

The two vectors are identical up to an order of 10^{-3} for all but one of the vectors. This shows that the network was able to accurately calculate a eigenvector of the matrix A.

The comparison between the neural network and numerical diagonalization is displayed in figure 8.

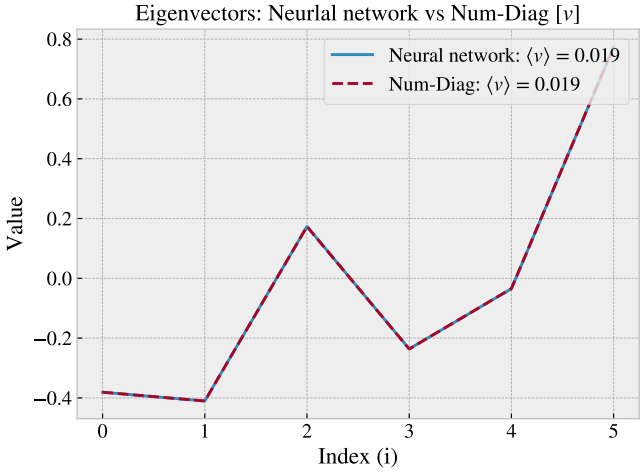


Figure 8: A comparison between the eigenvector produced by the network nad the eigenvector produced by numerical diagonalization.

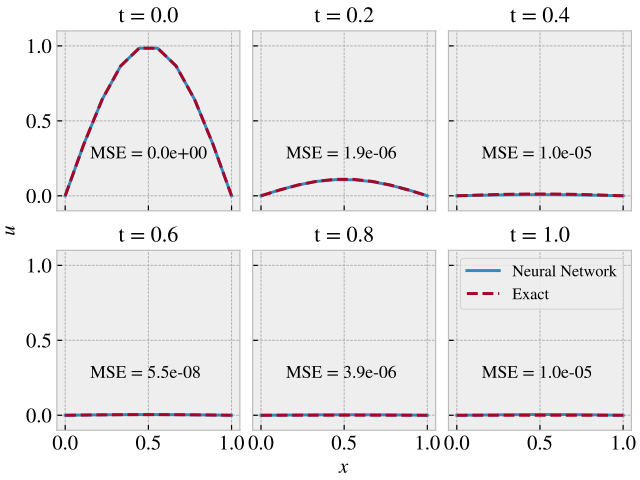
From figure 8 we observe that the vector produced from our network and the one produced by numpy are indistinguishable. The mean squared error of of the eigenvector is $2.08 \times 10^{-}13$.
Alternatively we could have studied other eigenvalues by using different distributions and random seeds, but by trial and error we found that the following combination produced the best results.

V. CONCLUSION

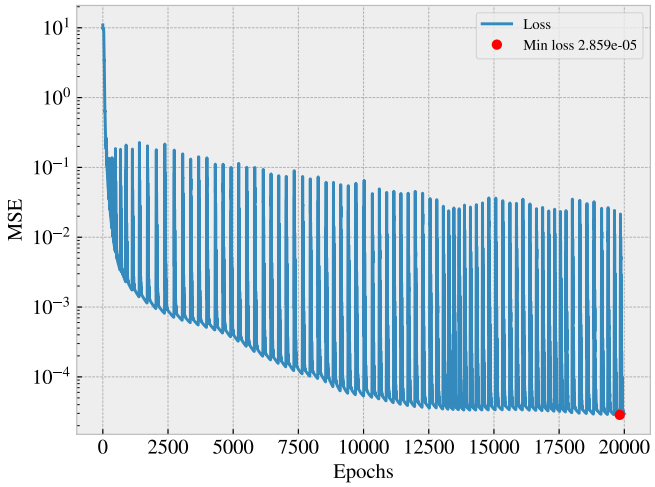
We successfully managed to produce numerical solutions for the 1D diffusion equation 1 using both the finite difference and neural network approach. We mainly focused on the numerical solutions using a spacial step size of $\Delta x = 0.1$ and $\Delta x = 0.01$. For the finite difference method we where limited by the von Neumann criteria given as $\Delta t = \Delta x^2/2$ where the Neural network has no predefined limits. However, it turned out that the network performance was better for an equal number of spacial and temporal points, i.e. $\Delta x = \Delta t$ when $L = T$. We simply accepted this sub-result, but it would be interesting to investigate the details of this behaviour. When comparing the numerical solutions against the analytical solution, mean squared error (MSE), for equal choices of Δx we found the finite difference method to be generally more accurate. The only exception was for small $t \lesssim 0.1$. However this was only achieved for a sufficient amount of epochs, 2000 epochs for $\Delta x = 0.1$ and 20,000 epochs $\Delta x = 0.01$, which required significantly more computational effort than the finite difference solutions. When computing the MSE as a function of computation time it was evident that the finite difference method is superior in terms of the error vs computation time ratio. It should be noted that this result might only be applicable to this specific type of problem given by the diffusion 1D equation. For more complex multidimensional problem this might not be the case, which serve as an interesting topic for future investigations. A benefit for the neural network though is the ability to evaluate the numerical solution in any given point in contrast to the finite difference solution which only gives you discretized point. However we did not investigate the quality of the neural network solution for points in-between and outside the training grid points, but this is yet another interesting topic for further studies.

ideas for future projects: neural network uneven dx and dt investigation.
In our study of the eigenvalue problem we found that we successfully calculated the maximum eigenvalue of a randomly generated symmetric matrix A, with an accuracy of up to 0.05% of . The value of the eigenvalue was found by our network

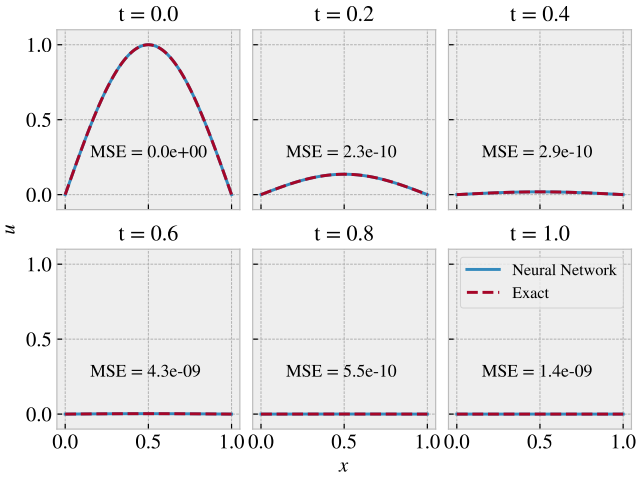
Appendix A: Neural network plots



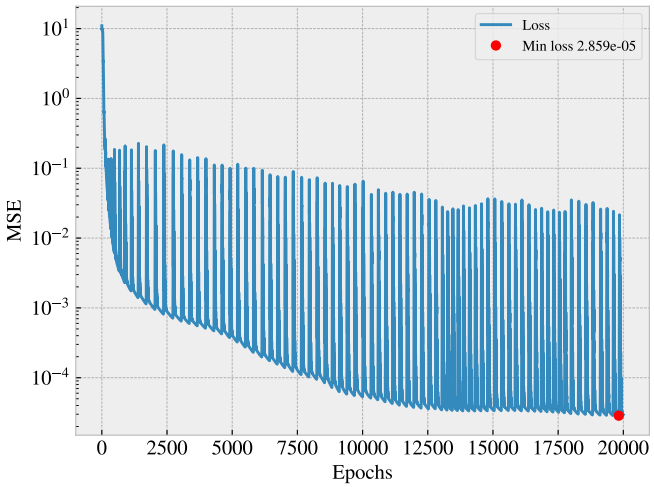
(a) Numerical solution, $\Delta x = \Delta t = 0.1$.



(b) Learning history, $\Delta x = \Delta t = 0.1$.



(c) Numerical solution, $\Delta x = \Delta t = 0.01$.



(d) Learning history, $\Delta x = \Delta t = 0.01$.

Figure 9: Neural network solution vs. exact solution for the diffusion equation 1 with $L = 1$ and $T = 1$ for different choice of $\Delta x = \Delta t$. We used 20000 epochs with a learning rate of 5×10^{-2}. MSE denotes the mean squared error between the numerical and analytical solution.

REFERENCES

[1] David Deuvenad et al. “Neural Ordinary Differential Equations.” In: (2018). DOI: https://arxiv.org/pdf/1806.07366.pdf.

[2] J. G. Charney, R. FjÖrtoft, and J. Von Neumann. “Numerical Integration of the Barotropic Vorticity Equation”. In: *Tellus* 2.4 (1950), pp. 237–254. DOI: 10.3402/tellusa.v2i4.8607. eprint: https://doi.org/10.3402/tellusa.v2i4.8607. URL: https://doi.org/10.3402/tellusa.v2i4.8607.

[3] Hans Petter Langtangen. *Finite Difference Computing with PDEs*. Texts in Computational Science and Engineering. Springer, 2016. ISBN: 978-3-319-55455-6.

[4] Hans Petter Langtangen. *Introduction to Numerical Methods for Variational Problems*. Texts in Computational Science and Engineering. Springer, 2016. ISBN: 978-3-319-55455-6.

[5] Zhang Yi, Yan Fu, and Hua Jin Tang. “Neural Networks Based Approach Computing Eigenvectors and Eigenvalues of Symmetric Matrix .” In: (2003). URL: https://reader.elsevier.com/reader/sd/pii/S0898122104901101?token=285C446EECA78BB06330CC19351564FA763701EC1AED7F900F0B771C78DDCC414E17F80A24E583AB289A684A01EEDC0E&originRegion=eu-west-1&originCreation=20211216101049.