Using the Forward Time Centred Space method and Neural Network to solve differential equations

Isabel Sagen, Eir Eline Hørlyk & Dorina Rustaj
(Dated: December 18, 2024)

In recent years, neural networks have proven to be especially useful for solving partial differential equations (PDEs), providing significant advantages over traditional numerical methods. To explore this, we have investigated the Forward time centred space (FTCS) explicit scheme and neural networks to solve for the homogeneous diffusion equation. We compared the performance of a neural network with the FTCS method by observing their MSE to the solution of the diffusion equation for different step sizes. According to our results FTCS yielded a smaller MSE with an order of around 10^{-11} compared to an order of 10^{-7} for neural network. Moreover, we explored the eigenvalue problem, a neural network-based approach for solving general ordinary differential equations (ODEs) by leveraging eigenvalues and eigenvectors. The eigenvalue problem was generated and tested for a 5×5 symmetric matrix. The neural network converged to the largest and smallest eigenvalues and their eigenvectors. Starting with the eigenvectors corresponding to the largest and smallest eigenvalues ensures convergence to the correct eigenvalues. All computational work is displayed in the GitHub repository [a].

I. INTRODUCTION

Partial Differential Equations (PDEs) are a necessity when modelling physical phenomena across fields such as heat transfer, fluid dynamics, quantum mechanics, and financial modelling. Although traditional numerical methods, example-wise finite difference schemes, have been the standard approach. Recent advancements in machine learning, particularly neural networks, have introduced flexible and scalable alternatives. Techniques such as Physics-Informed Neural Networks (PINNs) and neural network parameterizations, first introduced around 2017, have proved promising in solving high-dimensional PDEs. This challenges the conventional methods, effectively mitigating the curse of dimensionality. Foundational studies by Raissi et al. [3], Sirignano and Spiliopoulos [4], and Han et al. [1] have demonstrated the ability of neural networks to approximate solutions for forward and inverse problems, showcasing their potential to greatly improve computational science.

This investigation examines advancements in numerical methods by comparing the explicit finite difference scheme, specifically the forward time centred space (FTCS) method, with neural network-based solutions for the one-dimensional diffusion equation. The comparison focuses on the mean square error (MSE) for different step sizes in space and time, providing insights into the accuracy and stability of both approaches. To further evaluate the neural network's performance, its learning history and the effects of different activation functions are analysed. Additionally, we extend the analysis to a general eigenvalue problem, a commonly used technique for solving differential equations. In this context, the neural network is trained to approximate eigenvalues and eigenvectors by optimizing their representations and minimizing residuals. Overall, this study aims to assess the strengths and limitations of machine learning methods for solving differential equations, evaluate their scalability with increasing complexity, and investigate the influence of neural network architectures and activation functions on solution quality.

The report will have the following structure: the theoretical foundations for the algorithms will be outlined in the *Methods* section II, introducing the diffusion equation, FTCS method and neural network method in addition to the eigenvalue problem. This includes the analytical basis of the models. Further we showcase our results and a detailed analysis and reflection on the obtained results in the *Results and Discussion* section III. Here we will compare the FTCS method with the neural network for the diffusion equation and evaluate different tests on the eigenvalue problem for a symmetric 5×5 matrix. Finally, we summarize our work and discuss future improvements in the *Conclusion* section IV.

II. METHODS

Partial differential equations (PDEs) are used to model complex processes and systems with multiple variables, making them suitable for describing phenomena evolving in both space and time. We will provide the necessary background for solving the diffusion equation using forward time centred space explicit scheme and then a neural network utilising TensorFlow. Lastly we will present a general method for solving differential equations using neural network, known as the eigenvalue problem.

[a] https://github.uio.no/isabesag/FYS-STK4155_P3/tree/main

1. The diffusion equation

In one spatial dimension and a homogeneous medium, the *diffusion equations* reads

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

where $f(x,t)$ is the diffusive quantity, D is the diffusion coefficient, t is the time coordinate and x is the spatial coordinate. In our case $D = -1$, and 1 will correspond to the heat equation. The aim is to find $f(x,t)$ numerically, which requires some conditions. Firstly, we will only study the spacial domain where

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

at t = 0 and

$$f(x,0) = \sin \pi x.$$

We then applying Dirichlet boundary conditions

$$f(0,t) = f(1,t) = 0, \quad \text{for} \quad t \geq 0.$$

Furthermore, the analytical solution of our problem (1) is

$$f(x,t) = \exp\left(-\pi^2 t\right) \sin \pi x. \tag{2}$$

The complete derivation is presented in Appendix **??**.

2. Mean square error

To test the accuracy of our numerical methods we observe the *mean square error* (MSE). It measures the average squared difference between the numerical solution and the exact solution at specific points within the computational domain. It is mathematically defined as

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(u_i^{\text{num}} - u_i^{\text{exact}})^2, \tag{3}$$

where N is the total number of discrete points in the domain, u_i^{num} is the numerical solution at the i-th point, u_i^{exact} is the exact solution at the i-th point. The MSE provides a single scalar value that measures the overall deviation of the numerical solution from the exact solution.

3. Forward time centred space

One of the methods utilised to solve the diffusion equation is the Forward time centred space method. This is a explicit finite-difference scheme, meaning that the solution at each time step is calculated directly from the known values at the previous time step. A natural first step of the method is discretizing the space and time coordinates as

$$x_i = i\Delta x \quad \text{and} \quad t_n = n\Delta t.$$

Here $i \in [0, i_{\max}]$ and $i_{\max} = L/\Delta x$. For our investigation we set $L = 1$. The time index takes the values $n \in [0, n_{\max}]$ where $n_{\max} = T/\Delta t$. T is the total time the for the evolution. The FTCS uses the Forward Euler method for time evolution:

$$\frac{df_i^n}{dt} \approx \frac{f_i^{n+1} - f_i^n}{\Delta t} + \mathcal{O}(\Delta t). \tag{4}$$

This approximation provides first-order accuracy in time. In the spatial dimension, the second derivative is approximated using a centred difference scheme,

$$\frac{d^2 f_i^n}{dx^2} \approx \frac{f_{i+1}^n - 2f_i^n + f_{i-1}^n}{(\Delta x)^2} + \mathcal{O}(\Delta x^2), \tag{5}$$

which provides second-order accuracy in space. When we substitute equation (4) and (5) into the diffusion equation, (1) and solve for f_i^{n+1}, we get the complete FTCS scheme:

$$f_i^{n+1} \approx f_i^n + \frac{\Delta x}{\Delta t^2}\left(f_{i+1}^n - 2f_i^n + f_{i-1}^n\right). \tag{6}$$

Note that this method introduces two sources of truncation errors; from space and time. These errors arise due to the neglect of the higher-order terms of the Taylor expansions. We can approximate the global error to be proportional to

$$\mathcal{O}(\epsilon) \approx \mathcal{O}(\Delta x^2 \Delta t). \tag{7}$$

Naturally, smaller Δx and Δt values yields a smaller truncation error. However, we underline that FTCS is only conditionally stable. This means that the stability will depend on the Δx and Δt values. To take this into account, we use the following stability criterium:

$$\frac{\Delta t}{\Delta x^2} \leq \frac{1}{2}. \tag{8}$$

This ensures that the numerical solution remains bounded and does not behave unphysical by oscillating or diverging during time evolution. For the numerical implementation, boundary conditions are also required to fully define the problem. In this report, we apply Dirichlet boundary conditions, where the solution is fixed at the boundaries. Specifically, we set:

$$f(0, t = f(L,t) = 0.$$

These boundary conditions represent a system where the endpoints are fixed at a constant value, such as zero temperature. At every time step, the boundary values are explicitly set to assert this constraint. A full display of how this is implemented is shown in Algorithm 1.

Algorithm 1 FTCS

Input Δx value
$\Delta t = \Delta x^2/2$
Set up initial conditions
for i **do** ▷ up until $i = T/\Delta t$
$f_i^{n+1} \leftarrow f_i^n + \frac{\Delta x}{\Delta t^2}\left(f_{i+1}^n - 2f_i^n + f_{i-1}^n\right)$

FIG. 1. How we implemented the FTCS method numerically.

4. Neural network

The goal is to solve for two variables in equation (1): time (t) and position (x). These variables are presented by a one-dimensional array, and the network evaluates the solution at each possible pair (x, t), given arrays of desired x-values and t-values. The mean square error is used as the cost function defined as

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

Letting $x = (x_1, ..., x_N)$ represent an array of value for $x_1, .., x_N$, the the cost function simplifies to

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

Given M sets of values for $x_1, .., x_N$, where $x_i = (x_1^{(i)}, ..., x_N^{(i)})$ for $i = 1, .., M$ represents the $i-$ th row of the matrix X, the cost function generalizes to

$$C(x, P) = \tag{9}$$

$$\sum_{i=1}^{M} f\left(\left(x, \frac{\partial g(x)}{\partial x_1}, .., \frac{\partial g(x)}{\partial x_N}, \frac{\partial g(x)}{\partial x_1 \partial x_2}, .., \frac{\partial^n g(x)}{\partial x_N^n}\right)\right)^2. \tag{10}$$

For each pair of (x, t), the network evaluates the trial solution and calculates its Jacobian, which is in turn used to find derivatives of the trial solution with respect to x and t. The Jacobian is then differentiated to compute the second derivatives, resulting in a *Hessian* matrix containing all possible second-order mixed derivatives of $f(x, t)$. By these fact the cost function is simple the mean square error

$$\langle r^2 \rangle = \left\langle \left(\frac{\partial f_i^n}{\partial t} - \frac{\partial^2 f_i^n}{\partial x^2}\right)^2 \right\rangle.$$

Ideally, $r = 0$. This cost function frames the problem as a minimization task for the neural network, allowing it to learn without requiring explicit expected outputs or exact solutions. First- and second-order derivatives are computed using automatic differentiation.

The neural network optimizes its weights and biases, represented by a matrix P, by minimizing the cost function. The trial solution must meet specific system conditions and is expressed as

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

where, $h_1(x, t)$ and $h_2(x, t)$ enforce the initial and boundary conditions, and $N(x, t, P)$ is the neural network. The initial condition is enforced by

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

and the Dirichlet boundary conditions are enforced by

$$h_2(x, t) = x(1 - x)t.$$

Thus, the trial function becomes

$$f(x, t) = (1 - t)\sin \pi x + x(1 - x)tN(x, t, P).$$

5. The eigenvalue problem

When solving the diffusion equation with the method of separation of variables, an *eigenvalue problem* is bound to appear. More generally, one can find the eigenvectors corresponding to the largest or smallest eigenvalues. We will provide a method for computing eigenvectors of any real symmetric matrix, however, we will adhere to a 5×5 real matrix. Assume A is an $n \times n$ symmetric matrix, and the dynamics of the proposed neural network model is described by

$$\frac{dx(t)}{dt} = -x(t) + f(x(t)), \quad t \geq 0 \tag{11}$$

where

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

Here, $x = (x_1, ..., x_n)^T \in R^n$ represents the state of the network, I is the $n \times n$ identity matrix. We can derive eigenvectors and eigenvalues based on equilibrium points. The vector x is said to be an equilibrium point if $-x + f(x) = 0$, and

$$x^T x A x - x^T A x x = 0.$$

Once an eigenvector, v is obtained this could be if $lim_{t\to\infty} x(t) = v$ or x is the null vector, the corresponding eigenvalue can be computed from

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

Based on [5] there are a couple of theorems that are useful for finding the eigenvalues.

Theorem 1 *The following properties hold for the solutions of (11):*

1. *Each solution starting from any nonzero point in $\mathbb{R}^n$ will converge to an eigenvector of A.*
2. *Given any nonzero vector $x(0) \in \mathbb{R}^n$, if $x(0)$ is not orthogonal to V, the solution will converge to an eigenvector corresponding to the largest eigenvalue of A.*
3. *Replacing A in (11) with $-A$, and assuming $x(0)$ is a nonzero vector in $\mathbb{R}^n$ which is not orthogonal to V, the solution starting from $x(0)$ will converge to an eigenvector corresponding to the smallest eigenvalue of A.*

V is an eigenspace. Point 2 and 3 in the theorem gives the necessary condition for the network to converge to eigenvectors corresponding to the largest and smallest eigenvalue. To incorporate this into a neural network, one must first generate a symmetric matrix based on

$$A = \frac{Q^T + Q}{2},$$

where Q is any randomly generated real matrix. We then need to define a trial function, $g(t, P)$ which satisfy the boundary conditions; $g(0, p) = x(0)$ and $\lim_{t\to\infty} g(t, P) = \lim_{t\to\infty} N(t, P)$. The first ensures that trial function is a solution of the differential equation at $t = 0$ and the second part ensures that the trial function converges to the solution of the equation. Choosing the trial function as

$$g(t, p) = \exp\{-t\}x(0) + (1 - \exp\{-t\})N(t, P),$$

and the cost function,

$$C(t, P) = \left\langle \left(\frac{dg(t_i, P)}{\partial t} + g(t_i, P) - f(g(t_i, P)) \right)^2 \right\rangle.$$

Furthermore, we use ADAM optimisation based on the TensorFlow package [2].

III. RESULTS AND DISCUSSION

A. The diffusion equation

We begin the analysis by examining the differential method used to solve the diffusion equation (1).

1. FTCS

Using the FTCS method we solve the diffusion equation. We want to test the method for $\Delta x = 0.1$ and $\Delta x = 0.01$. From equation (8), $\Delta x = 0.1$ and $\Delta x = 0.01$ will correspond to the time step values of $\Delta t = 5 \cdot 10^{-3}$ and $\Delta t = 5 \cdot 10^{-5}$, respectively. We want to study the curve at four different points in time, $t_{1,2,3,4}$, that display a range of curves, from almost linear curve to a prominent curve. Note here that $t_{1,2,3,4}$ is not the time point for the index n, but arbitrary point that display the desired behaviour. After trail and error, we found that $t = 0.02, 0.2, 0.4, 0.7$ where our points of interest.

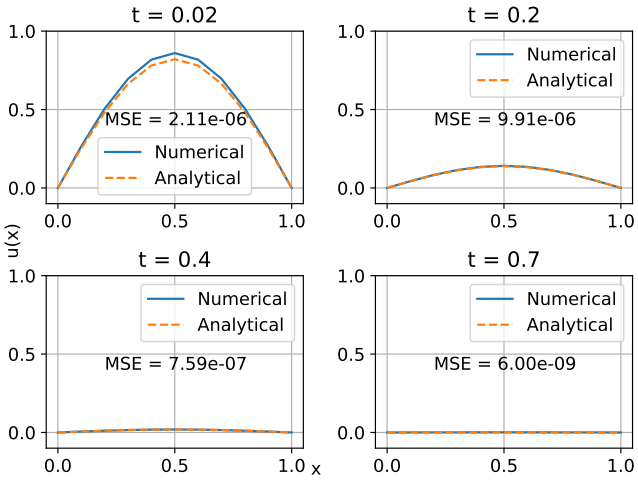

FIG. 2. Comparing the numerical and analytical solution to the diffusion equation for $t = 0.02, 0.2, 0.4, 0.7$ for chosen step size $\Delta x = 0.1$ and $\Delta t = 5 \cdot 10^{-3}$. Here we also see the MSE values for the different time points

The solutions for $\Delta x = 0.1$ and $\Delta x = 0.01$ are displayed in Figure 2 and Figure 3, respectively. Each figure compares the numerical solution with the analytical solution and includes the MSE for the selected t values. Notably the general MSE is very small taking into account the small resolution in space. For both figures we see that the MSE value is largest for $t = 0.2$, and from there, decreases with increasing t values. This can be a result of the diffusion equation being more dynamic in the beginning, and stabilizing as the system evolves. Further, comparing Figure 2 and 3, as expected, the MSE values are significantly smaller for the latter. Roughly, the MSE are smaller by a factor of 10^{-4}, which corresponds with the approximated order of the error (7). To look deeper into this we illustrate the MSE as a function of time for $\Delta x = 0.1$ and $\Delta x = 0.01$ in Figure 4. The results confirm the earlier observations: the MSE peaks right after the beginning of the simulation and decreases evenly as time passes. This may a result of the more pronounced dynamics in the beginning, calming down with time. Moreover, the smaller Δx value provides a MSE smaller by a factor of 10^{-4}, again, as expected from equation (7). This stays consistent during the whole time evolution.

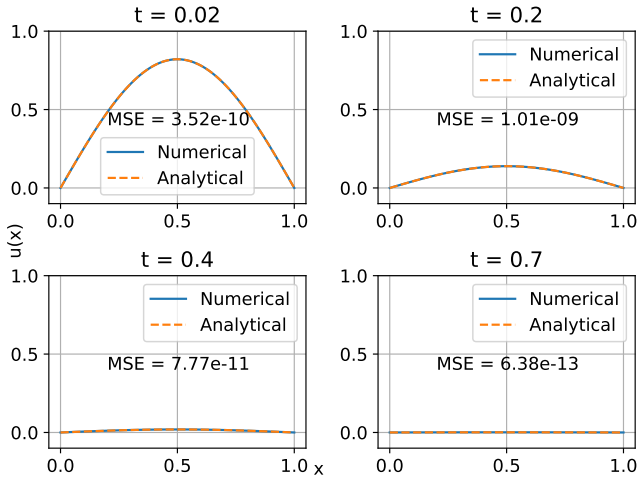

FIG. 3. The numerical and analytical solution to the diffusion equation for t = 0.02, 0.2, 0.4, 0.7 for chosen step size $\Delta x = 0.01$ and and $\Delta t = 5 \cdot 10^{-5}$. Comparing these numerical and analytical we get the MSE value for the different time points.

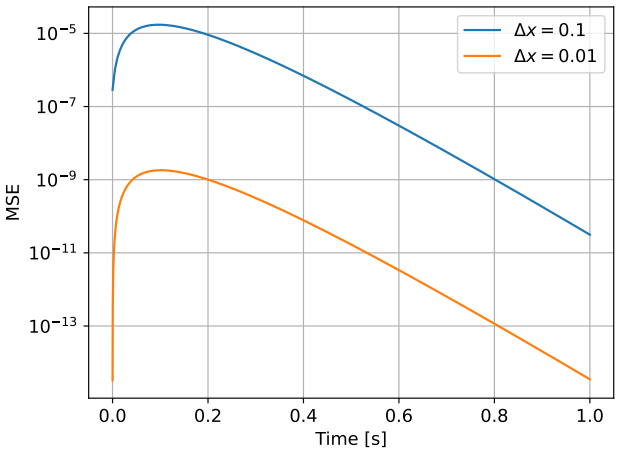

FIG. 4. The MSE as a function of the time t for chosen step sizes $\Delta x = 0.1$ and $\Delta x = 0.01$. Comparing these we observe that $\Delta x = 0.01$ curve has a smaller MSE of order 10^{-4} throughout the whole time evolution.

2. *Neural network*

We now want to look at how a neural network solved the diffusion equation (1). A trial-and-error approach was used to determine the appropriate learning rate and number of epochs. The optimal numbers were $\eta = 0.001$ and epochs = 5000. The full architecture can be found in Table I.

The exact solution for $\Delta x = 0.01$ and $\Delta t = 0.005$ shown in Figure 6, while the exact solution for $\Delta x = \Delta t = 0.02$ is shown in Figure 5. Both chosen time steps align well with the exact solution, but Figure 6 yields a

TABLE I. The parameters chosen for the architecture of the Neural network to solve the diffusion equation.

Quantity	Value
Learning rate	0.001
Epochs	5000
Activation	Tanh
Outer layer activation	Sigmoid

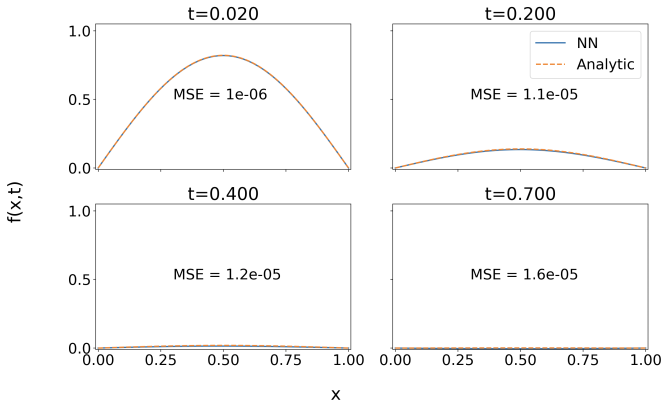

FIG. 5. Exact solution to the diffusion equation for t = 0.02, 0.2, 0.4, 0.7 for chosen stepsize $\Delta x = \Delta t = 0.2$ from the neural network. The MSE for each time is also showcased.

slightly smaller overall MSE value, with the smallest error occurring at $t = 0.02$ and an MSE value of 2.2×10^{-8}. The learning histories, represented by the MSE values as a function of epochs are plotted in Figure 7 for $\Delta x = 0.01$ and $\Delta t = 0.0005$, and in Figure 8 for $\Delta x = \Delta t = 0.02$. Both figures exhibit an exponentially decaying trend with occasional spikes, and the MSE values across epochs are comparable between the two cases. Figure 9 provides a clearer picture of the MSE as a function of time. For $\Delta x = 0.01$ and $\Delta t = 0.005$, the MSE remains relatively

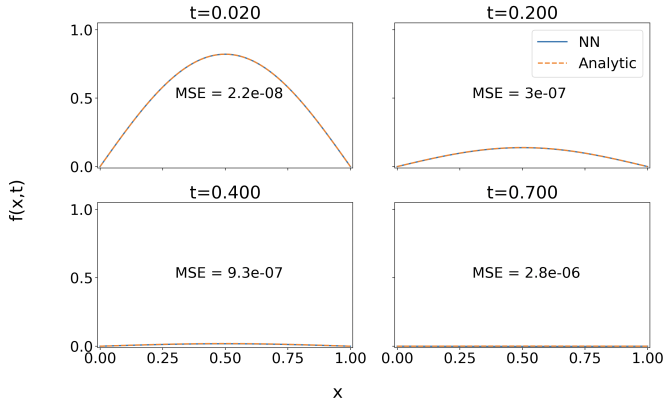

FIG. 6. Exact solution to the diffusion equation for t = 0.02, 0.2, 0.4, 0.7 for chosen stepsize $\Delta x = 0.01$ and $\Delta t = 0.005$ from the neural network. The MSE for each time is also showcased

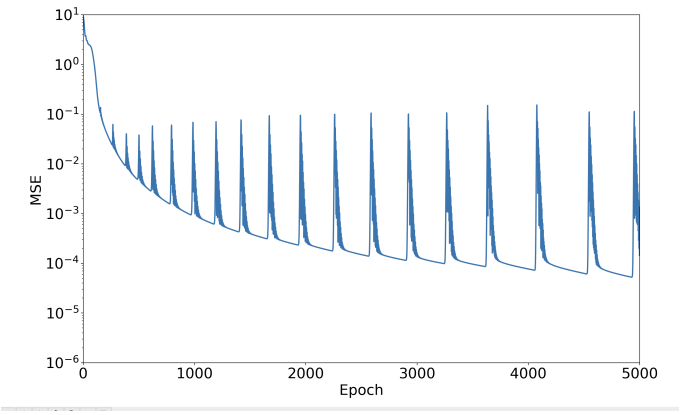

FIG. 7. MSE against number of epochs for $\Delta x = 0.01\Delta t = 5\times10^{-3}$. Showcase the learning history of the neural network.

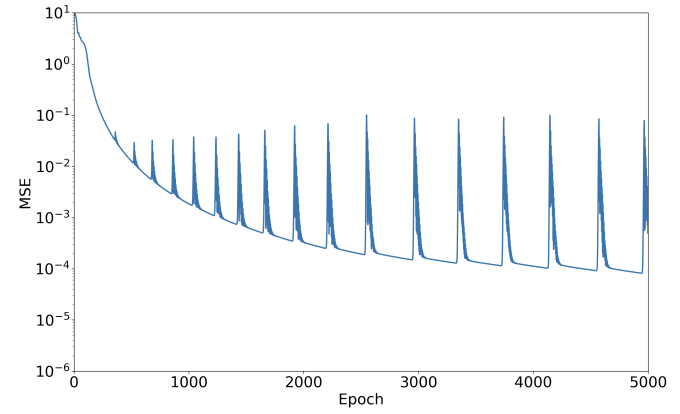

FIG. 8. MSE against number of epochs for $\Delta x = \Delta t = 0.2$. Showcase the learning rate of the neural network

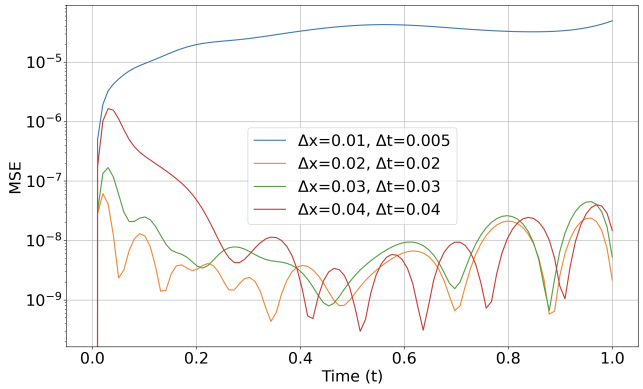

FIG. 9. MSE against t for different Δx and Δt values from the trained neural network

stable, whereas for $\Delta x = \Delta t = 0.02$, the MSE shows a slightly lower trend over time for the trained data. Overall, while $\Delta x = \Delta t$ may offer marginal better stability in the MSE trend, the differences are minor. This suggest that choosing $\Delta x = \Delta t$ does not significantly affect the accuracy of the exact solution, as the configurations exhibit similar MSE trends.

The architecture shown in Table I consists of two hidden layers with 100 nodes, where the activation function in the first layer is tanh, while the second layer uses sigmoid. This decision was based on the fact that having two hidden layers provides sufficient expressive power to approximate the complexity of the diffusion equation. According to the universal approximation theorem, a neural network with at least one hidden layer and a non-linear activation function can approximate any continuous function. The larger number of nodes enhances the networks ability to represent fine details and subtle variations in the solution, this is important because the diffusion equation often has a smooth but non-trivial solutions that require a detailed representation. The tanh function outputs values in the range $[-1, 1]$, which helps normalize the input and ensure smoother gradients during backpropagation, it captures both positive and negative variations in the data, making it suitable for approximating function with oscillatory behaviours. The sigmoid function outputs values in the range $[0, 1]$, which aligns well with the boundary conditions. Moreover, it introduces non-linearity, allowing the network to approximate intricate mappings between inputs and outputs. The combination with tanh handling broader variations and sigmoid refining the final output suited best for this purpose. However, one could have tried with ReLU or Leaky ReLU for faster convergence and better gradient flow, but for the simple one dimension diffusion equation our choice of activation function was sufficient. Lastly, one could have experienced with a deeper network that offers the potential for better approximations and the ability to handle more complex scenarios, but it comes with increased computational costs and challenges in training. Based on the MSE values on Figures 9 our problem does not demand added complexity, and a simpler architecture is more than efficient.

The neural network undergoes a training process aimed at minimizing the loss function and fitting the given data as closely as possible. Unlike direct analytical methods, the network does not explicitly solve for an exact theoretical solution. As a result, the shape of the numerical solution may exhibit slight deviations from the exact solution due to the inherent approximations involved in the optimization process. As shown in Figures 6 and 5 for$\Delta x = 0.01$ and $\Delta t = 0.005$ and $\Delta x = \Delta t = 0.02$ respectively, the analytical and numerical solutions align well overall, but minor variations can be observed. These variations may arise from factors such as the neural networks architecture, the choice of hyperparameters, or the stochastic nature of the training process. Despite these small deviations, the networks solution remains a close approximation and aligns effectively with the data. This is evident when comparing the networks output to the solution provides by the FTCS method, as shown in Figure 3. While the FTCS solution serves as the benchmark, the neural network demonstrates its capability to

approximate the solution with high accuracy, validating its potential as an effective numerical solver.

Figures 7 and 8 illustrate the learning history of the neural network, showcasing a descending trend in the MSE as a function of epochs. The overall behaviour follows an exponential decay with multiple spikes observed along the curve. Around 5000 epochs, one can notice a convergence trend, which aligns with the original choice for the optimal number of epochs. An interesting phenomenon is the presence of spikes in the MSE during training. These spikes may result from periodic resets or adjustments within the optimization process. Optimizers with larger learning rates can induce oscillations or instability, manifesting as spikes in the training error. Additionally, the ADAM optimizer, which utilise per-parameter adaptive learning rates based on moving averages of gradients and squared gradients, could also contribute to this behaviour. Noise in gradient estimates or overly aggressive adjustments of the moving averages may lead to temporary oscillations, producing the observed spikes.

To address this issue, one potential solution involves tuning the hyperprarameters, such as reducing the learning rate. Experiments were conducted with learning rates $\eta = 0.1$ and $\eta = 0.005$, but the results did not differ significantly. Another observation is that increasing the number of epochs does not necessarily improve the MSE further, suggesting diminishing returns beyond certain training duration. Moreover, the choice of $\Delta x = \Delta t$ does not seem to provide any noticeable advantage for the neural networks performance, as evidenced by figure 9. While these variations do not significantly affect the solutions in Figures 6 and 5, careful tuning of the learning rate and other hyperparameters remains crucial for ensuring stable and efficient training.

The FCTS method and neural network represent two fundamentally different approaches for solving the diffusion equation 1. The FTCS evolves with time and relies on an explicit update formula to propagate the solution from one time step to the next. In contrast, the neural network learns a solution function directly by minimizing the loss function derived from the PDE and the boundary conditions. Instead of relying on explicit discretization, the network uses optimization techniques to approximate the solution as a continuous function. The approach provides the flexibility to evaluate $u(x,t)$ at arbitrary points without a predefined grid, making it suitable for irregular domains or higher-dimensional problems. However, training the neural network is computationally expensive and required careful tuning of hyperprarameters to achieve optimal performance.

This difference in the way the two methods operate is shown clearly when comparing the MSE history in Figure 4 and Figure 9. For FTCS that evolves with time, the curve is very smooth and almost linear. On the other hand, the neural network's MSE shows a more irregular evolution, reflecting the complexity of the training process.

As shown in Figures 3 and 2, the FTCS predicts the solution more accurate than the neural network in this case based on the MSE values. For $\Delta x = 0.01$ the FTCS provides a MSE smaller than for the neural network by a factor of approximately 10^{-4}. This outcome reflects the efficiency of FTCS for simple 1D or 2D problems where a grid-based methods suffice. By contrast, the neural networks ability to handle high-dimensional problems and irregular domains underscores its potential for more complex scenarios, albeit at a higher computational cost.

B. Eigenvalue problem

We will now look into solving the eigenvalue problem using neural network. A symmetric 5×5 matrix A was generated from a random uniform distribution. The eigenvalues and the eigenvectors were computed using a neural network with a single hidden layer and 1000 nodes. The rest of the architecture can be found in Table II. Based on the previous neural network to solve the diffusion equation, a learning rate $\eta = 0.0001$ and ADAM optimization were chosen. The predictions were made with total time $T = 10^3$ with $N = 100$ points. The largest eigenvalue $\lambda_{\max}$ was computed using A, while the smallest eigenvalue $\lambda_{\min}$ was computed with $-A$. A comparison with numpy's eigenvalue solver are found in Table III in addition the MSE value. Figures 10 and 11 showcase the largest and smallest eigenvalues as a function of epochs. We can notice from Figure 10 that the eigenvalue converges to $\lambda_{\max}$ after around 100 epochs, while $\lambda_{\min}$ converges after 100 epochs in figure 11. Figures 12 and 13 shows the eigenvectors that converges after approximately 200 epochs for both the largest and smallest eigenvalue, respectively.

TABLE II. The parameters chosen for the architecture of the Neural network to solve the eigenvalue problem.

Quantity	numerical value
Learning rate	0.0001
Epochs	5000
Activation	ReLU
Outer layer activation	None

TABLE III. Comparison of the largest and smallest eigenvalue compared to the solution provided by Numpy's eigenvalue solver

Eigenvalue	numpy solution	NN	MSE
$\lambda_{\max}$	2.92	2.92	2.4×10^{-4}
$\lambda_{\min}$	-0.48	-0.48	9.3×10^{-6}

Lastly, the same symmetric matrix A was utilised to

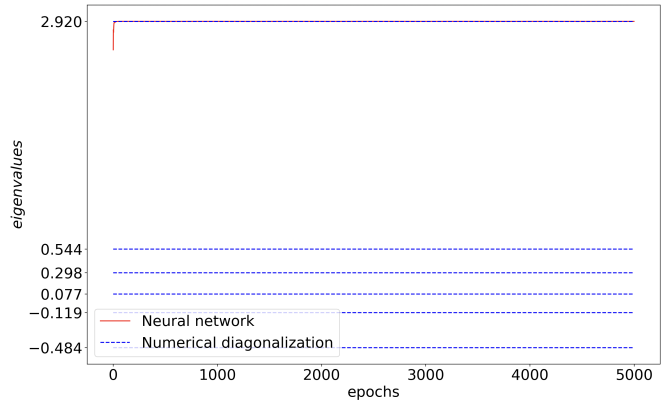

FIG. 10. The largest eigenvalue as a function of epochs for a neural network with a single layer containing 1000 nodes. Eigenvalues computed using NumPys eigenvalue solver are shown as dashed lines.

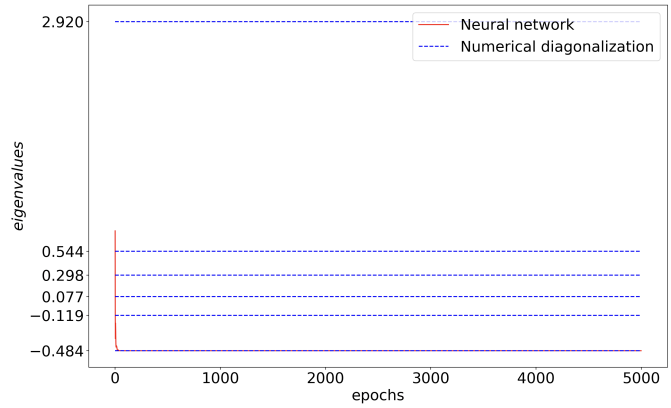

FIG. 11. The smallets eigenvalue as a function of epochs for a neural network with a single layer containing 1000 nodes. Eigenvalues computed using NumPys eigenvalue solver are shown as dashed lines.

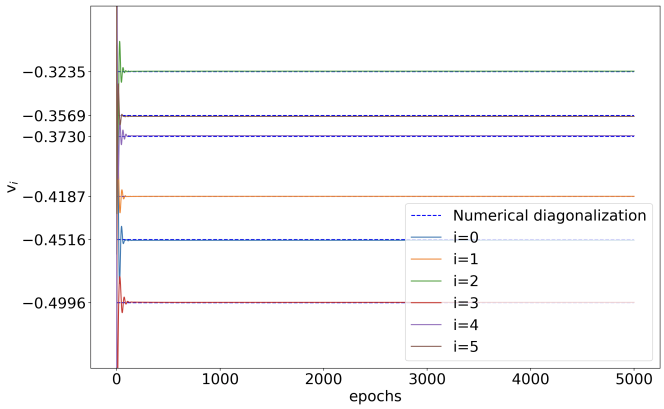

FIG. 12. Eigenvector components as a function of epochs for the largest eigenvalue in a neural network with a single layer containing 1000 nodes. Eigenvectors computed using NumPys eigenvalue solver are shown as dashed lines.

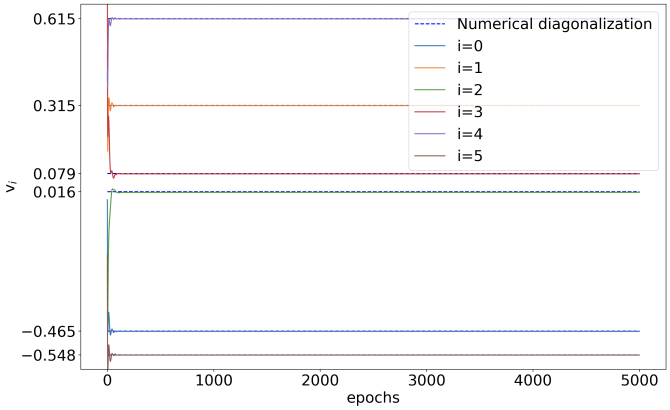

FIG. 13. Eigenvector components as a function of epochs for the smallest eigenvalue in a neural network with a single layer containing 1000 nodes. Eigenvectors computed using NumPys eigenvalue solver are shown as dashed lines.

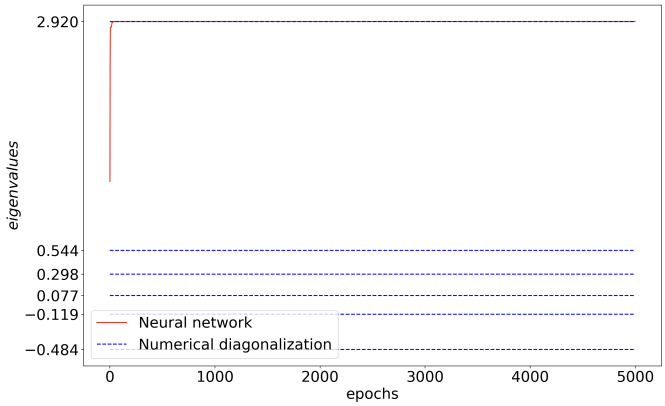

FIG. 14. Evaluation of the eigenvalue as a function of epochs when initialized to the eigenvector corresponding to the largest eigenvalue in a neural network with a single hidden layer containing 1000 nodes. Eigenvalues computed using NumPy's eigenvalue solver are shown as dashed lines.

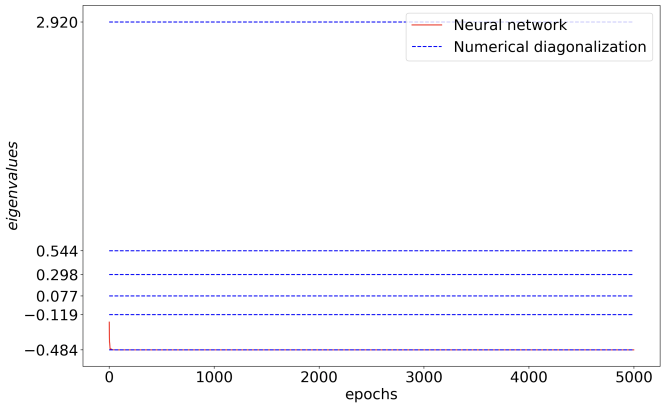

FIG. 15. Evaluation of the eigenvalue as a function of epochs when initialized to the eigenvector corresponding to the smallest eigenvalue in a neural network with a single hidden layer containing 1000 nodes. Eigenvalues computed using NumPy's eigenvalue solver are shown as dashed lines.

compute the eigenvalues starting from eigenvectors corresponding to $\lambda_{\max}$ and $\lambda_{\min}$ shown in Figures 14 and 15.

The neural network successfully demonstrated its ability to converge to both the largest and smallest eigenvalues of A, along with their corresponding eigenvectors when the initial vector $x(0)$ was drawn from a normal distribution. This behaviour aligns with the theoretical basis described in theorem 1, which states that any non-zero starting point will eventually converge to an eigenvector of A.

When analysing the convergence behaviour under specific initialisation, particular to eigenvectors corresponding to the largest and smallest eigenvalues, the results in Figures 14 and 15 demonstrate that the method reliably converges to the correct eigenvalue for the given eigenvector. The eigenvalue corresponding to the largest eigenvector in Figure 14 and the smallest eigenvector in Figure 15 both converged after approximately 100 epochs. This matches the convergence trends seen in Figures 10 and 11.

This behaviour can be attributed to the power iteration method; if an initial vector $x(0)$ is already aligned with an eigenvector, the linear combination contains only that eigenvector, and no other components are amplified, consequently the power iteration preserves the alignment, and the process quickly converges to the associated eigenvalue. Any initial vector $x(0)$ can be expressed as a linear combination of the eigenvectors of A due to the spectral theorem. Over successive iterations, the dominant eigenvalue grows exponentially compared to the smaller eigenvalues, and the corresponding eigenvector component dominates the vector. Thus, when initializing to the largest eigenvector, the method naturally converges to the largest eigenvalue. Considering the smallest eigenvalue, the matrix A is transformed to $-A$, and the eigenvalues are inverted, and the smallest eigenvalue becomes the largest in magnitude in the transformed system. In this case, the power iteration method behaves as expected, amplifying the component of the vector aligned with the smallest eigenvector, ensuring convergence to the corresponding eigenvalue.

Figures 10 and 11 reveal similar convergence trends for the largest and smallest eigenvalues, indicating consistent learning behaviour by the neural network. From Table III, it is evident that the MSE for the smallest eigenvalue is two orders of magnitude smaller than that for the largest eigenvalue. These differences can be attributed to several factors related to the neural networks learning process and the matrix properties. Firstly, the smallest eigenvalue is computed using the transformation $-A$, where it becomes the largest eigenvalue of $-A$. Neural networks tend to optimise more effectively for dominate features, and the largest eigenvalue of $-A$ naturally receives such focus, leading to a lower MSE. Secondly, eigenvalue magnitude plays a role: $\lambda_{\min}$ has a smaller absolute value than $\lambda_{\max}$, contributing less to the loss functions overall error and resulting in better convergence. Lastly, the gradients associated $-A$ might be more stable and consistent, facilitating improved optimization for $\lambda_{\min}$. These factors collectively explain the faster convergence and lower MSE for the smallest eigenvalue. Nevertheless, both MSE values are low, highlighting the effectiveness of the neural network in approximating eigenvalues and eigenvectors. This performance reflects the suitability of the chosen hyperparameters, as detailed in Table II.

While the eigenvalues converge relatively quickly, eigenvectors exhibit a slightly delayed convergence. Figure 13 shows that the eigenvector corresponding to the smallest eigenvalue converges faster than that for the largest eigenvalue Figure 12. This discrepancy likely arises from the differing nature of these quantities: eigenvalues are scalar and represent single values per eigenvector, making their optimization simpler. In contrast, eigenvectors are multidimensional, requiring the network to capture complex relationships between their components. Additionally, gradients for scalar quantities tend to be more stable and stronger than those for multi-dimensional outputs, further contributing to the faster convergence of eigenvalues compared to eigenvectors.

The architecture shown in Table II consists of a single layer with 1000 nodes and ReLU as activation function which proved effective due to several inherent characteristics of the problem. First, a single layer with 1000 nodes has a high capacity to represent complex functions. Even without multiple layers, this wide layer can approximate the eigenvalue problems solution due to the sheer number of parameters. For the eigenvalue problem, where solutions often involve smooth functions and linear components (eigenvectors are linear combinations of basis vectors), the networks wide architecture ensures it can approximate the required mappings accurately. The ReLU activation introduces non-linearity allowing the network to model non-linear relationships that arises in the eigenvalue problems. Furthermore, it outputs zero for negative inputs, which introduces sparsity in the networks activations, which can improve efficiency and reduce overfitting in certain cases. Unlike sigmoid or tanh, ReLU does not suffer from vanishing gradients for positive inputs. This ensures that the training process remains stable and gradients propagate effectively, even in a network with many nodes. Additionally, the eigenvalue problem itself lends itself well to this configuration. The smoothness of eigenvectors and the well-defined mathematical structure of eigenvalue make the solution space relatively low-dimensional and conducive to approximation by a wide, shallow network. This property ensures that the network can converge effectively to solutions by minimizing residuals. While this architecture has clear strengths, it also comes with limitations. The reliance on a single layer means that the network might struggle to scale efficiently for problems or greater complexity or dimensionality. In such cases, deeper networks or alternative architectures such as PINNs could be more appropriate. Furthermore, the computational cost of training such a wide network

increase with the problems scale, which may become prohibitive for very high-dimensional systems.

IV. CONCLUSION

In this report we have solved the diffusion equation at the time-points $t = 0.02, 0.2, 0.4, 0.7$ for different Δx and Δt values using both the FTCS method and a neural network. For the FTCS method we found that the MSE decreased significantly, by approximately 10^{-4}, when transitioning from $\Delta x = 0.1$ to $\Delta x = 0.01$. For each of the cases, the smallest MSE was for $t = 0.7$, likely because the system having stabilized during the time evolution. When comparing these result to those obtained by using the neural network we observed the lowest MSE was found at $t = 0.02$ for $\Delta x = 0.01$ and $\Delta t = 0.005$ with the MSE value of 2.2×10^{-8}. In contrast to using FTCS, the smallest MSE for neural network did not correspond to the largest t value. This could be a result of FTCS evolving with time while with neutral network the results comes from training. For the diffusion equation this may favour FTCS as it generally gave smaller MSE values for $\Delta x = 0.01$ by order of approximately 10^{-4}.

Plotting the learning history for the neural network showcased an decreasing exponential trend with spikes most likely due to the chosen ADAM optimiser, but the MSE were the lowest around 10^{-4} for 5000 epochs. The MSE as a function of time were also showcased and the results told that choosing $\Delta x = \Delta t$ gave the lowest MSE values around $10^{-8} - 10^{-9}$ for a trained neural network. Moreover, we created a single layer neural network with a learning rate of $\eta = 0.001$, 5000 epochs and ADAM optimisation method to solve the eigenvalue problem. The neural network converged to both the largest and smallest eigenvalue in addition to their eigenvectors of a randomly generated symmetric 5×5 matrix after around 100-200 epochs. The errors were found to be 2.4×10^{-4} for the largest eigenvalue and 9.3×10^{-6} for the smallest eigenvalue. When initialising from the eigenvector corresponding to the smallest and largest eigenvalue the neural network successfully converged to the correct eigenvalue after around 100 epochs, showcasing its application to solve differential equations.

The FTCS method and neural network represent two distinct paradigms for solving the heat equation, each with advantages and limitations. The trade-off between stability and computational cost is a well-documented limitation of explicit finite difference schemes.

A. Future work

The findings highlight the complementary nature of these methods. While FTCS is well-suited for traditional, grid-based numerical analysis, the neural network offer a modern alternative for tackling complex problems, where FTCS may fail. Future work could include hybrid approaches, combing the computational efficiency of finite difference methods with the flexibility of the neural network. For example, neural networks could be used to approximate corrections to finite difference solutions, or finite difference schemes could provide initial guesses for training the neural network.

Moreover, extending the neural network approach to adaptive learning techniques could reduce training time and computational costs, enhancing its practical utility. Additional investigations into the interpretability of neural network solutions and robustness under varying conditions would further solidify their role in scientific computing.

REFERENCES

[1] Jiequn Han, Arnulf Jentzen, and Weinan E. "Solving high-dimensional partial differential equations using deep learning". In: *Proceedings of the National Academy of Sciences*. Vol. 115. 34. 2018, pp. 8505–8510.

[2] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[3] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational Physics* 378 (2019), pp. 686–707.

[4] Justin Sirignano and Konstantinos Spiliopoulos. "DGM: A deep learning algorithm for solving partial differential equations". In: *Journal of Computational Physics* 375 (2018), pp. 1339–1364.

[5] Zhang Yi, Yan Fu, and Hua Jin Tang. "Neural networks based approach for computing eigenvectors and eigenvalues of symmetric matrix". In: *Computers Mathematics with Applications* 47.8 (2004), pp. 1155–1164. ISSN: 0898-1221. DOI: https://doi.org/10.1016/S0898-1221(04)90110-1. URL: https://www.sciencedirect.com/science/article/pii/S0898122104901101.

We used ChatGPT to help with fine tuning some parts of the writing of the report.

APPENDIX A

The analytical solution for the diffusion equation can be found by using the method of separation of variables, which look for a solution of the form

$$u(x,t) = X(x)T(t), \tag{12}$$

for functions X, T to be determined. We can plug in $u = XT$ into the diffusion equation to arrive at

$$XT' - kX''T = 0,$$

divided by kXT, to get

$$\frac{T'}{kT} = \frac{X''}{X} = -\lambda,$$

for some constant λ. Therefore, if there exists a solution of the form (**??**) of the diffusion equation, then T and X must satisfy the equations

$$\frac{T'}{kT} = -\lambda,$$
$$\frac{X''}{X} = -\lambda.$$

In addition, in order for u to satisfy our boundary conditions, we need our function X to satisfy our boundary conditions, hence we need to find functions X and scalars λ such that

$$\begin{cases} -X''(x) = \lambda X(x), & x \in \mathcal{I} \\ X \text{ satisfies our BCs.} \end{cases}$$

This is known as an eigenvalue problem. We then use the Dirichøet Boundary Conditions to find all solutions to the eigenvalue problem

$$\begin{cases} -X''(x) = \lambda X(x), & 0 < x < 1 \\ X(0) = 0 = X(1). \end{cases}$$

We first look for positive eigenvalues; $\lambda = \beta^2 > 0$

$$\begin{cases} -X'' + \beta^2 X = 0, & 0 < x < 1 \\ X(0) = 0 = X(1). \end{cases}$$

And find the solutions

$$X(x) = C\cos\beta x + D\sin\beta x,$$

with boundary conditions $X(0) = 0 \to C = 0$ and $X(1) = 0 \to \sin\beta = 0 \to \beta = \pi$. We therefore get a sequence of positive eigenvalues $\lambda = \pi^2$, with the corresponding eigenfunction $X(x) = D\sin(\pi x)$. Furthermore, our solutions for T are given by

$$T(t) = A\exp\{-k\lambda t\} = A\exp\{-\pi^2 t\}.$$

By letting $A = D = 1$ we get the full solution

$$f(x,t) = \exp\{-\pi^2 t\}\sin\pi x.$$