Using machine learning to study a PDE

Behnoosh Ashrafi, Derya Aricigil, Jørn Eirik Betten
(Dated: December 17, 2021)

In this project we investigate two different methods for solving a partial differential equation of two dimensions, an Euler scheme and neural network, respectively. There is also an implementation of a neural network that was supposed to be able to find the eigenvalues of a symmetric input matrix A. We conclude that the numerical approach was better in approximating the diffusion equation, and our neural network found an eigenvector of the input matrix A.

I. INTRODUCTION

Here we implement different methods for solving the diffusion equation, a partial differential equation, using the Forward Euler method and Feed Forward Neural Network(FFNN) at different time points. The diffusion equation utilized consists of two dimensions: x and t, and we do not do calculations beyond these two dimensions. Then we will proceed to analyze an eigenvalue problem using a FFNN. The report is outlined with a methods section next that briefly solves the diffusion equation analytically, then explains the numerical approach of the explicit Euler scheme, and discusses a few details about the neural network approach to solving a differential equation, and also a bit on a specific differential equation that can solve eigenvalue problems for a symmetric matrix. In the Results and Discussion section we first present results from the numerical and the neural network approximation of the diffusion equation, then we look at the results from our FFNN eigensolver. Finally, we conclude.

Our codes can be found in our GitHub repository [1].

II. METHODS

The diffusion equation

We have the partial differential equation (PDE)

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

with the initial conditions

$$\begin{aligned} u(x,0) &= sin(\pi x) \;\; 0 \leq x \leq L, \\ u(0,t) &= 0 \;\; t \geq 0, \\ u(L,t) &= 0 \;\; t \geq 0, \end{aligned} \tag{2}$$

which we will solve numerically, analytically and using machine learning methods.

Analytical solution to the one dimensional diffusion equation.

We assume a separable solution

$$u(x,t) = \phi(x)\theta(t), \tag{3}$$

and insert in the PDE,

$$\frac{\partial^2 \phi(x)}{\partial x^2}\theta(t) = \frac{\partial \theta(t)}{\partial t}\phi(x). \tag{4}$$

We divide both sides of the equation with $u(x,t)$ and get

$$\frac{d^2\phi(x)}{dx^2}\frac{1}{\phi(x)} = \frac{d\theta(t)}{dt}\frac{1}{\theta(t)}. \tag{5}$$

For all x and t the relation must hold, thus both sides must be equal to a constant, which we'll call $-\lambda^2$. We then get two ordinary differential equations of the forms

$$\begin{aligned} \frac{d^2\phi(x)}{dx^2} &= -\lambda^2\phi(x), \\ \frac{d\theta(t)}{dt} &= -\lambda^2\theta(t), \end{aligned} \tag{6}$$

which have general solutions

$$\begin{aligned} \phi(x) &= Asin(\lambda x) + Bcos(\lambda x), \\ \theta(t) &= Ce^{-\lambda^2 t}. \end{aligned} \tag{7}$$

When we now apply the boundary conditions in 2 to the solutions. Then we get a more specific solution

$$u(x,t) = A_n sin(\frac{n\pi}{L}x)e^{-\frac{n^2\pi^2}{L^2}t}, \tag{8}$$

which in reality is non-specific, as there are infinite solutions, corresponding to $n = 1, 2, \ldots, \infty$. However, due to the linearity of the PDE we know that any superposition of solutions also is a solution, and we may write

$$u(x,t) = \sum_{n=1}^{\infty} A_n sin(\frac{n\pi}{L}x)e^{-\frac{n^2\pi^2}{L^2}t}, \tag{9}$$

as a general solution. By using $u(x,0) = sin(\pi x)$ we will specify our solution, via Fourier's trick:

$$A_n = \frac{2}{L}\int_0^L sin(\pi x)sin(\frac{n\pi}{L}x)dx. \tag{10}$$

In our case, we simply scale the problem by setting $L = 1$ as the natural unit length. After doing this, we recognize the orthogonality of the basis set $\mathcal{F} = \sin(n\pi x)\}$, where $n = 1, 2, \ldots, \infty$. Our initial condition is in fact one of

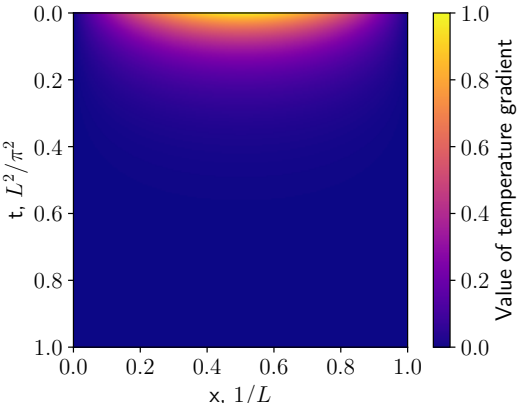


FIG. 1: A colourmap of the analytical solution of the heat equation, on the grid of values with $x \in [0, L]$ and $t \in [0, \frac{L^2}{\pi^2}]$. The dissipation of the heat gradient with time is quadratically dependent on the length of the rod.

our basis functions, namely $sin(\pi x)$, and therefore, we have that

$$A_n = \begin{cases} 1 & \text{if } n = 1, \\ 0 & \text{otherwise.} \end{cases}$$

The analytic solution of the diffusion equation is therefore

$$u(x,t) = sin(\frac{\pi}{L}x)e^{-\frac{\pi^2}{L^2}t}, \tag{11}$$

where x has units L, and the domain is $x \in [0, 1]$. A colourmap of the analytical solution is shown in figure 1, which has been made using equation 11 on a grid of values with $x \in [0, L]$ and $t \in [0, \frac{L^2}{\pi^2}]$.

Numerical solution to the PDE

To numerically solve the PDE we need to discretize the ranges of the spatial and the time dimensions. For the range of the spatial dimension we discretize the x, t and u values followingly

$$\begin{aligned} x &\to x_i = i\Delta x, \\ t &\to t_n = n\Delta t, \\ u(x,t) &\to u(x_i, t_n) = u_i^n, \end{aligned} \tag{12}$$

for all $i = 0, 1, \ldots, M$, and $n = 0, 1, \ldots, N$. We implemented an explicit scheme, the forward Euler, to numerically solve equation 1. The forward Euler method uses exclusively values from the nth time step to calculate values at the $(n + 1)$th time step. The method can be mathematically written as

$$u_i^{n+1} = u_i^n + \frac{\Delta t}{\Delta x^2}[u_{i+1}^n - 2u_i^n + u_{i-1}^n]. \tag{13}$$

The forward Euler method is explained further in V A.

A neural network as a differential equation solver.

For a general deep neural network approach, see [2]. The main approach reason why neural networks can be used as solvers for differential equations is due to the nature of the problem. We want one function, which is dependent on some kind of gradient, to become equal some other function. In the neural network approach, we want to minimize the error in the differential equation itself. The cost function is in a sense the differential equation problem (we square it so that we won't have to deal with derivatives of absolute values). However, we need to define a so-called trial function which we need to use to put into the differential equation. The trial function needs to have qualities that yield the correct boundary and initial conditions of the differential equation, and it needs to be dependent on the parameters of the neural network. Let's denote how the neural network deals with input as the function $N(\text{input}, P)$, where P is its trainable variables. The general formula for the trial function we use, is

$$g_t(x) = h_1(x) + h_2(x)N(x, P), \tag{14}$$

where we have denoted g_t as the trial function, $h_1(x)$ as a part of the function that is not dependent on the variables of the neural network and $h_2(x)N(x, P)$ as the part of the function that is influenced by the neural network. We call $h_1(x)$ our initial guess, where we utilize our knowledge about the solution of the differential equation. In the case of the diffusion equation, we had a few initial and boundary conditions 2. We have the initial condition $u(x, 0) = sin(\pi x)$, and the boundary conditions $u(0, t) = u(1, t) = 0$. We had these in mind when we came up with our trial function,

$$g_t(x,t) = (1-t)sin(\pi x) + x(1-x)tN(X,P), \tag{15}$$

where X is our meshgrid of values we feed the network. The idea is that the neural network will optimize its parameters in a way such that this trial function will replicate a solution to the differential equation. In the case of our eigenvalue problem, we based our cost function on the nonlinear differential equation

$$\dot{x} = -x(t) + f(x(t)), \tag{16}$$

where $f(x) = [x^T x A + (1 - x^T A x)I]x$, where A is a square and symmetric matrix, say dimensions $n \times n$, and x is a vector of length equal n. I is the identity matrix of dimensions $n \times n$. We chose a trial function that looks like

$$trial(x,t) = e^{-t}x + (1 - e^{-t})N(t, P), \tag{17}$$

as it can be shown that the solution to this differential equation is when the trial function is an eigenvector of A. The inital vector x will vanish as the neural network takes over at larger values of t. We want the neural network N to yield an eigenvector, as this would minimize the cost

function (the right hand side equals the left hand side of equation 16). Also here we square the difference. More about the mathematics behind the eigenvalue problem, see [3].

ADAM optimizer

For the neural network methods, we always use the ADAM optimizer as our optimizer. When using stochastic gradient descent (SGD), we need to tune the learning rate (η) as a function of time. η determines the speed of which we move in the slops of gradient descent. If it is too large we may jump out of the min zone and if it is too small it will take a long time to reach the min. Choosing the right η can save us a lot of time and resources. But deciding on a constant η is not practical as we want our learning rate to adapt itself based on the steepness or flatness of the curvature. A lot of methods have been introduced to accomplish this. Adagrad, AdaDelta, RMSprob, ADAM, etc. These methods try to take into account more than just the gradient, but also the second moment of the gradient.

Here we use ADAM. ADAM is very similar to RMSprob but here we only explain ADAM. This algorithms estimates based on the 1th order moment (the gradient mean), $\mathbb{E}[\mathbf{g}_t]$, and 2nd order moment (element-wise squared gradient), $\mathbb{E}[\mathbf{g}_t^2]$.

final weight updates $\propto \eta \frac{\text{1st moment}}{\sqrt{\text{2nd moment}^2}}$

This means that η will be large when in flat surfaces and can move faster and will be small when in steep. This means moving fast and taking large steps when in the safe zone and taking small and careful steps when one step in the wrong direction can take ruin the results and lead the model in the wrong direction.

We can show each single parameter θ_t by:

$$\Delta\theta_{t+1} = -\eta_t \frac{\boldsymbol{m}_t}{\sqrt{\sigma_t^2 + m_t^2 + \epsilon}}.$$

in which σ_t^2 is the variance given by $\boldsymbol{\sigma}_t^2 = \mathbf{s}_t - (\mathbf{m}_t)^2$, and $\mathbf{s}_t = \mathbb{E}[\mathbf{g}_t^2]$. $\epsilon \sim 10^{-8}$is a small regularization parameter to prevent divergence.

The algorithm

III. RESULTS & DISCUSSION

Forward Euler Scheme

Comparing results for $t = 0$, $t = 1.0$ using $\Delta x = 1/10$ and $\Delta x = 1/100$ in fig. 9 and fig. 10 shows that dividing the x interval into 100 subintervals, $\Delta x = 1/100$, provides a significant improvement of estimating the analytical solution in comparison to dividing it in 10 subintervals, $\Delta x = 1/10$. This is accompanied by the use of a larger dt at 0.05 instead of 0.005, which is desirable for a more efficient computation.

Furthermore, as t increases from $t = 0$ to $t = 1.0$ there appears a visible error in fig. 9b. This is because fig. 9a is unable to show smaller range of values, which means that the deviation seen in 9b from the analytical solution appears more clearly. The solution obtained with Euler is found below the curve of the analytical solution, closer to 0, indicating that the heat dissipates faster from the rod according to the Euler scheme.

$$\epsilon_{absolute} = |u(x,t)_{approx.} - u(x,t)_{analytical}| \tag{18}$$

In the absolute error plot shown in fig. 2, which is calculated using eq. 18, the error appears largely in the beginning portion of the time interval. In the yellow or orange portions of the figures is where the difference is the largest, presumably due to the Euler scheme estimating faster heat dissipation during this time period, and then slowing down.

Solving the diffusion equation using a neural network.

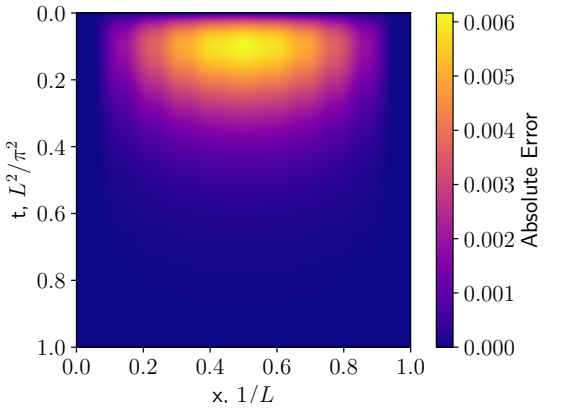
Training

When solving the diffusion equation on the domain $t, x \in [0, 1]$ using a neural network approach as the solver, we set the the training grid of values to be a 20×20 of x and t, and set the learning rate to $\eta = 0.01$. We chose four hidden layers with a 100 neurons in each of them. The evolution of the cost function as a function of training iterations are shown in figure 3. The figure shows a quite rapid decline of the cost function. When the training was done, we ran a different meshgrid of 100×100 values of x and t through the optimized network, and the results can be viewed in figure 4, which is a 3D-projection of the temperature gradient. For comparison, figure 5 displays the analytical solution. Figure 6 presents the absolute error of the neural network approximation as a colourmap. The absolute error has its maximum where the analytical solution changes most rapidly, with a value of order $\sim 10^{-2}$. If we were to use a more fine-grained training set, this error would probably have been possible to reduce. Using a relatively "small" training set of only 20×20 values might not be enough to cover the nuances in this region. However, this would increase the computational cost by quite a lot, as the neural network performs multiple operations on each value.

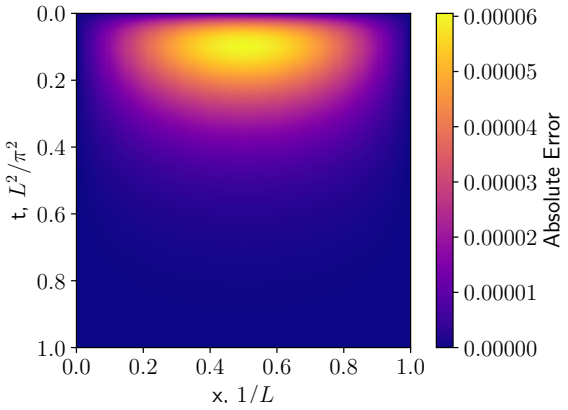
Comparing Euler with neural network

Fig. 4 gives the approximated solution using neural networks and fig. 5 is the analytical solution.

The absolute difference found between neural networks and analytical solution is found in fig. 6. It shows that



(a) $\Delta x = 1/10$



(b) $\Delta x = 1/100$

FIG. 2: Absolute difference between the solved diffusion equation using the Forward Euler Scheme with $dt = 0.05$ and $N = 101$, compared to the analytical solution.

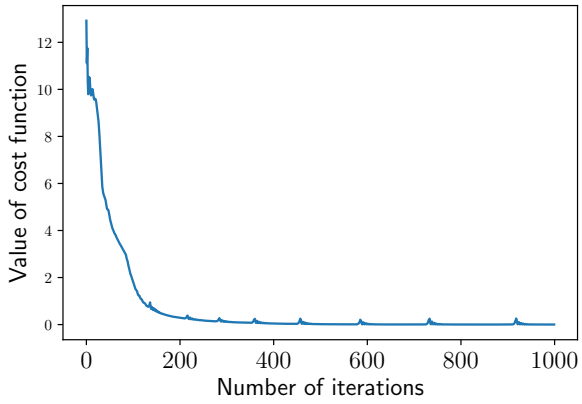


FIG. 3: The evolution of the value of the cost function as a function of number of training iterations in the neural network. The network was trained on a 20×20 meshgrid of x- and t-values.

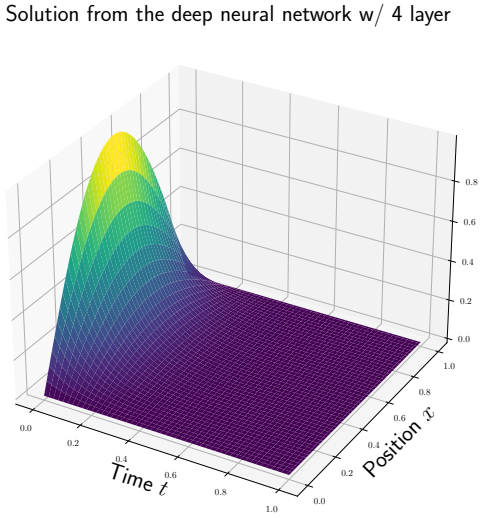


FIG. 4: Calculated solution of the diffusion equation using a neural network with 4 hidden layers with 100 neurons each, plotted using a 3D-projection. The calculated solution is the prediction on a 100×100 meshgrid of x- and t-values.

the error is minimal in the beginning, i.e. the heat dissipation is approximately the same. Then the proceeding yellow region is, according to our assumption, due to the neural networks' inability to train on too many grid points, as this would be computationally very expensive. The maximal absolute error is of the same order as when we used only 100 grid points to calculate a numerical approximation using the forward Euler method. Using the forward Euler method with only 100 mesh grid points was far quicker to compute. The timing of the forward Euler with $\Delta x = 0.01$ gives a far better approximation, compared to the neural network. It was also quicker to compute.

Solving eigenvalue problem with neural network.

Table I lists the calculated coordinates of the eigenvector, one using the **eig** functionality in the **np.linalg** library, and the other using the neural network approach. The neural network converges to an eigenvector solution, but as we see in figure 8, it's not the one that was predicted in [3], the eigenvector corresponding to the maximal eigenvalue, but the third smallest. Figure 8 shows how the eigenvalue converges as a function of the number of iterations of the neural network, whilst figure 7 shows how each individual coordinate of the trial function approaches the coordinates of the eigenvector corre-

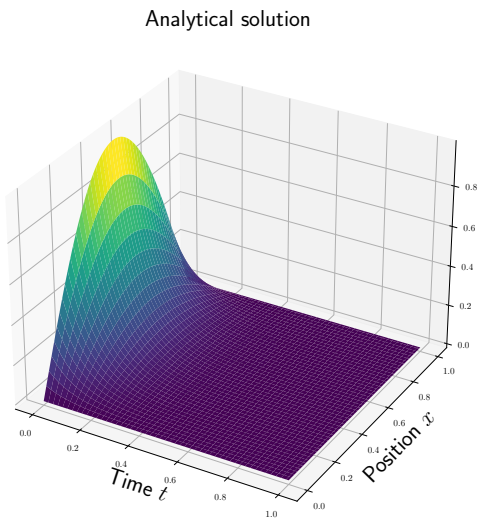


FIG. 5: Analytical solution of the diffusion equation plotted in a 3D-projection, using the same grid of x- and t-values as in figure 4.

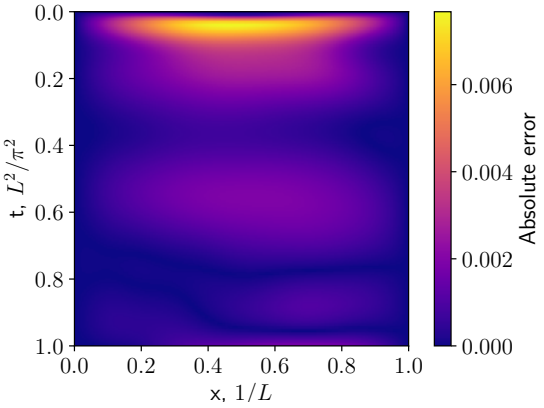


FIG. 6: The absolute difference between the analytical solution of the diffusion equation and the calculated solution using a the neural network approach.

sponding to this eigenvalue. It's not what was expected at all, as they proved mathematically that if we chose a starting vector that is not orthogonal to the eigenvector corresponding to the maximal eigenvalue, the method would yield that eigenvector. Regardless, the neural network yielded an eigenvalue, and an eigenvector, as this minimized the cost function.

Calculated coordinates	True coordinated
0.41128969	0.411289687
-0.19969813	-0.199698133
0.70877585	0.708775848
0.18903505	0.189035051
0.50256167	0.502561675
-0.01719763	-0.0171976269

TABLE I: Neural network prediction of the eigenvector versus the true eigenvector

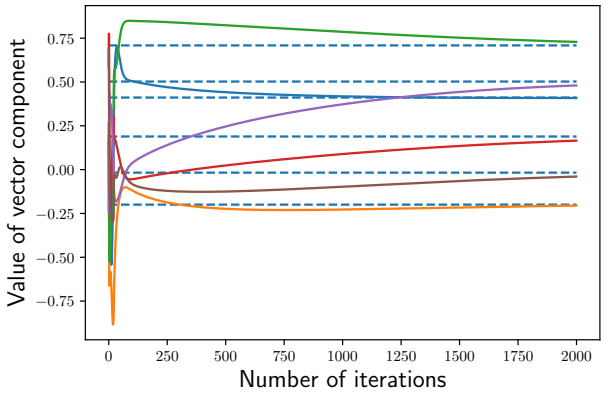


FIG. 7: The dashed lines are the coordinates of the eigenvector corresponding to the eigenvalue of our input matrix A calculated via the **eig** method in the **numpy.linalg** library. The other non-dashed lines correspond to the development of the trial function as a function of iterations. The trial function approaches an eigenvector of the input matrix A.

IV. CONCLUSION

In this project we approached the diffusion equation with three different methods for solving it. First we try a numerical approach, the explicit Euler scheme. The stability criteria for the explicit scheme is limiting our computational capacity for solving the diffusion equation exactly. However, we solve it numerically with a small absolute error, of order $\sim 10^{-2}$ using a step size of $\Delta x = 0.1$. The absolute error is decreased significantly, to an order of $\sim 10^{-4}$, which is expected. Next we solved the diffusion equation using a neural network. The computational cost of increasing the number of training points and iterations, gave us a best approximation with a maximal absolute difference of order $\sim 10^{-2}$. We therefore see that the explicit Euler scheme outperforms the neural network approach in two dimensions. Lastly, we tried to implement a neural network that yields an eigenvector of a square, symmetric matrix A. We succeeded in that the neural network approach in fact yielded an eigenvector of the input matrix, however, it did not yield

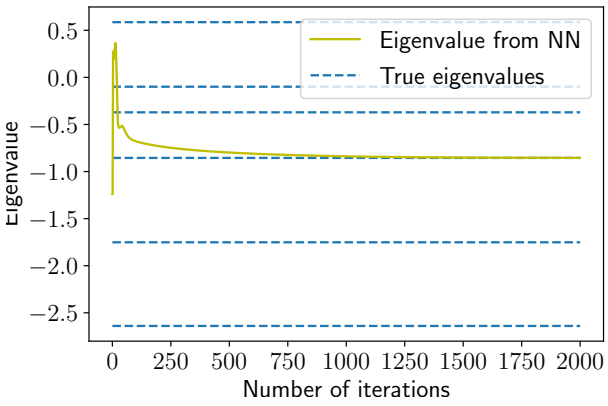


FIG. 8: The dashed lines are the true eigenvalues of the matrix A, calculated via the **eig** method in the **numpy.linalg** library. The yellow line shows the development of the calculated eigenvalue of the yielded vector from the trial function. It approaches an eigenvalue of the matrix A.

the eigenvector corresponding to the maximal eigenvalue, as was predicted in [3].

V. APPENDIX

A. Discretization and numerical solver

We numerically solved the heat equation 1 using the forward Euler method. Discretizing the x-values with M equidistant points in the interval $[0, L = 1]$, giving a step size of

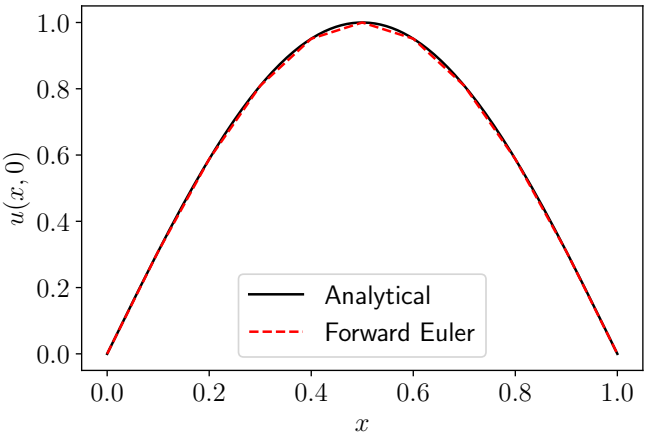
$$\Delta x = \frac{1}{M-1}, \tag{19}$$

and for the values in the time dimension we discretize with N equidistant points from $t_0 = 0$ to $t_N = T$, and the step size, Δt, becomes

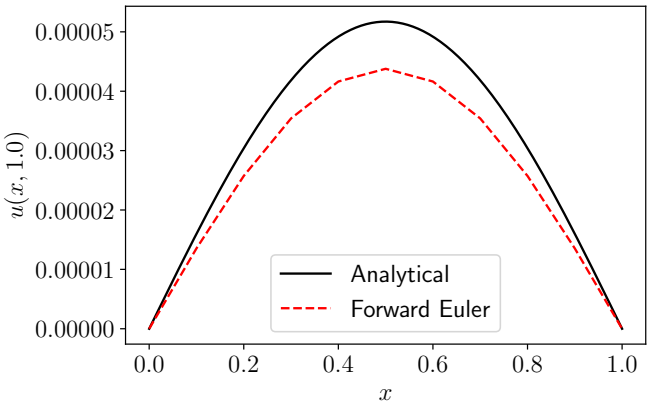
$$\Delta t = \frac{T}{N-1}. \tag{20}$$

We then define $x_i = i\Delta x$ and $t_n = n\Delta t$ for all $i = 0, 1, \ldots, M$ and $n = 0, 1, \ldots, N$. Then we can define $u(x_i, t_n) = u_i^n$, for all values of i and n, and we have discretized our target. Now, we have

[1] J. E. Betten, D. Aricigil, and B. Ashrafi, "Project 3 github repository," (2021).
[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning* (MIT Press, 2016) http://www.deeplearningbook.org.
[3] Z. Yi, Y. Fu, and H. J. Tang, Computers Mathematics with Applications **47**, 1155 (2004).

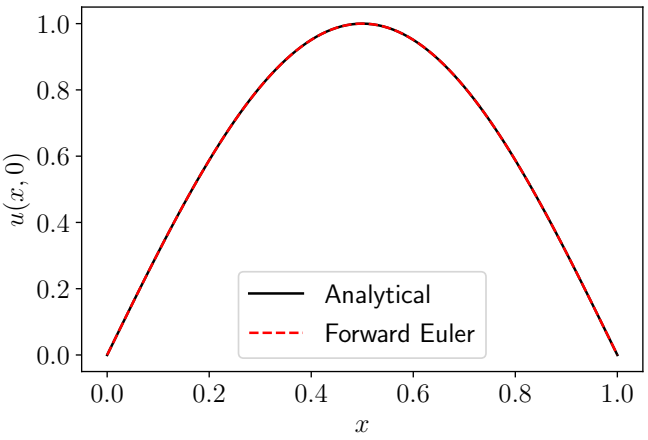


(a) Euler t=0.0.

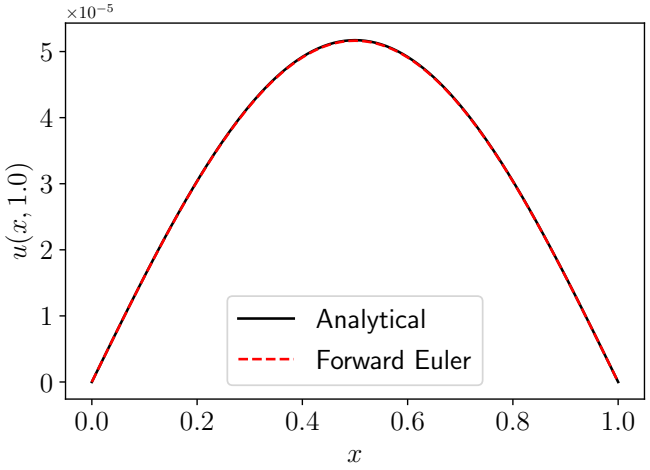


(b) Euler t=1.0.

FIG. 9: Diffusion equation plots solved using the Forward Euler Scheme with $dt = 0.005$, $N = 11$ and $\Delta x = 1/10$



(a) Euler t=0.0.



(b) Euler t=1.0.

FIG. 10: Diffusion equation plots solved using the Forward Euler Scheme with $dt = 0.05$, $N = 101$ and $\Delta x = 1/100$