Indian Institute of Technology Bombay

PH-408 Mathematical Physics

Numerical Techniques to solve non linear PDEs

Submitted To:
Prof. Dibyendu Das
Department of Physics

Submitted By :
Anjali Yadav (215120022)
Harsh Choudhary (215120027)
Aakash Shandilya (215120031)
Sahil Meshram (215120005)

Submitted on: 13-04-2022

Contents

1 Abstract

Non Linear Differential Equation. In these type of equations the dependent variable and its derivatives are of order higher than 1 and the coefficient can depend on the dependent variable. Another way to define non linear differential equations is to say that those equations that can't be to reduced to the form $Ly = f$, where L is some linear operator.
The basic difference between Linear and non linear PDE is that in Linear PDEs if we have some solution to a PDE say u_1 and some other solution say u_2, then the sum of these solutions is also a solution, this is not true for non-Linear PDEs

Numerical Methods. Very few of the non linear PDEs have analytical solutions. Most of the PDEs can only be solved by numerical methods which only give an approximation of the solution. Several numerical methods exist to solve non linear PDEs such as finite difference method, finite volume method, finite element method, etc. In this project we are going to focus on the finite difference method and another method that uses deep learning to give a continuous solution.

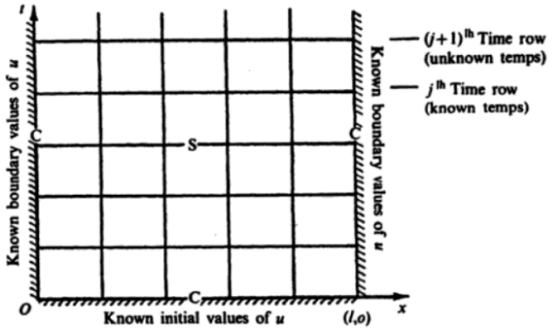
2 Finite Difference Method

The finite difference method is an approximate method for solving differential equations. It can be used to solve a wide range of problems. These include linear and non-linear, time independent and dependent problems. This method can be applied to problems with different kinds of boundary conditions and initial conditions.

In this method we approximate the derivatives of a function by an algebraic formula derived by eliminating terms of higher order from the appropriate Taylor series expansion for the function. We then replace the derivatives in the differential equations by these approximations and get an algebraic equation that can be solved by recursion. Below we first see how to approximate the first and second partial derivatives of a function and we then see how to solve a differential equation using these approximations.

2.1 First derivative

1. **Forward Difference Method**



Taylor series expansion,

$$f(x_0 + h, t) = f(x_0, t) + \frac{\partial f}{\partial x}h + \frac{\partial^2 f}{\partial x^2}\frac{h^2}{2!} + ... \implies f(x_0 + h, t) = f(x_0, t) + \frac{\partial f}{\partial x}h + O(h^2)$$

Calculating $\frac{\partial f}{\partial x}$ from the above formula

$$\frac{\partial f}{\partial x} = \frac{f(x_0 + h, t) - f(x_0, t)}{h} - O(h)$$

We neglect the $O(h)$ term and get finite difference approximation for the first derivative using forward difference method

$$\frac{\partial f}{\partial x} = \frac{f(x_0 + h, t) - f(x_0, t)}{h}$$

Likewise

$$\frac{\partial f}{\partial t} = \frac{f(x, t_0 + h) - f(x, t_0)}{h}$$

2. **Backward Difference Method**
Taylor series expansion,

$$f(x_0 - h, t) = f(x_0, t) - \frac{\partial f}{\partial x}h + \frac{\partial^2 f}{\partial x^2}\frac{h^2}{2!} + ... \implies f(x_0 - h, t) = f(x_0, t) - \frac{\partial f}{\partial x}h + O(h^2)$$

Calculating $\frac{\partial f}{\partial x}$ from the above formula

$$\frac{\partial f}{\partial x} = \frac{f(x_0, t) - f(x_0 - h, t)}{h} + O(h)$$

We neglect the $O(h)$ term and get finite difference approximation for the first derivative using backward difference method

$$\frac{\partial f}{\partial x} = \frac{f(x_0, t) - f(x_0 - h, t)}{h}$$

Likewise

$$\frac{\partial f}{\partial t} = \frac{f(x, t_0) - f(x, t_0 - h)}{h}$$

3. **Central Difference Method**
We subtract the Taylor series expansions for $f(x_0 - h)$ from $f(x_0 + h)$

$$f(x_0 + h, t) - f(x_0 - h, t) = 2\frac{\partial f}{\partial x}h + \frac{2h^3}{3!}\frac{\partial^3 f}{\partial x^3} + ...$$

Calculating $\frac{\partial f}{\partial x}$ from the above formula

$$\frac{\partial f}{\partial x} = \frac{f(x_0 + h, t) - f(x_0 - h, t)}{2h} - \frac{h^2}{3!}\frac{\partial^3 f}{\partial x^3} + ...$$

Neglecting the $O(h^2)$ term, we get

$$\frac{\partial f}{\partial x} = \frac{f(x_0 + h, t) - f(x_0 - h, t)}{2h}$$

Likewise

$$\frac{\partial f}{\partial t} = \frac{f(x, t_0 + h) - f(x, t_0 - h)}{2h}$$

Note that the term that we neglected in the central difference method is of $O(h^2)$, and that in the case of forward and backward difference was of $O(h)$. Therefore central difference method gives much more accurate value of the derivative.

2.2 Second derivative

1. **Central Difference method**
   We add the Taylor series expansions for $f(x_0 + h, t)$ and $f(x_0 - h, t)$ to get

$$f(x_0 + h, t) + f(x_0 - h, t) = 2f(x_0, t) + \frac{2h^2}{2!}\frac{\partial^2 f}{\partial x^2} + \frac{2h^4}{4!}\frac{\partial^4 f}{\partial x^4} + ...$$

   Solving for $\frac{\partial^2 f}{\partial x^2}$, we get

$$\frac{\partial^2 f}{\partial x^2} = \frac{f(x_0 + h, t) - 2f(x_0, t) + f(x_0 - h, t)}{h^2} - \frac{2h^2}{4!}\frac{\partial^4 f}{\partial x^4} - .....$$

   Neglecting the $O(h^2)$ term, we get

$$\frac{\partial^2 f}{\partial x^2} = \frac{f(x_0 + h, t) - 2f(x_0, t) + f(x_0 - h, t)}{h^2}$$

   Likewise

$$\frac{\partial^2 f}{\partial t^2} = \frac{f(x, t_0 + h) - 2f(x, t_0) + f(x, t_0 - h)}{h^2}$$

   Again the central difference method is more accurate compared to forward and backward difference methods, and central difference method is what we are going to use going forward.

2. **Forward Difference method**
   Subtracting $2f(x_0 + h, t)$ from $f(x_0 + 2h, t)$ and then solving for $\frac{\partial^2 f}{\partial t^2}$ gives us

$$\frac{\partial^2 f}{\partial x^2} = \frac{f(x_0 + 2h, t) - 2f(x_0 + h, t) + f(x_0, t)}{h^2} + O(h)$$

   Likewise

$$\frac{\partial^2 f}{\partial t^2} = \frac{f(x, t_0 + 2h) - 2f(x, t_0 + h) + f(x, t_0)}{h^2} + O(h)$$

3. **Backward Difference method**
   Subtracting $2f(x_0 - h, t)$ from $f(x_0 - 2h, t)$ and then solving for $\frac{\partial^2 f}{\partial t^2}$ gives us

$$\frac{\partial^2 f}{\partial x^2} = \frac{f(x_0, t) - 2f(x_0 - h, t) + f(x_0 - 2h, t)}{h^2} + O(h)$$

   Likewise

$$\frac{\partial^2 f}{\partial t^2} = \frac{f(x, t_0) - 2f(x, t_0 - h) + f(x, t_0 - 2h)}{h^2} + O(h)$$

2.3 Solving a non-linear PDE

Using the finite difference method we solve **Burger's equation**

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}$$

We approximate the function $u(x, t)$ at points

$$x_1 = x_0 + h,\ x_2 = x_0 + 2h, ..., \ x_i = x_0 + ih, ..., \ x_{n-1} = x_0 + (n-1)h$$

$$t_1 = t_0 + h,\ t_2 = t_0 + 2h, ..., \ t_j = t_0 + jh, ..., \ x_{n-1} = t_0 + (m-1)h$$

where x_0, x_n, t_0 and t_m are the boundary conditions.
The central difference formula for the first derivative becomes

$$\frac{\partial u}{\partial x} = \frac{u_{i+1,j} - u_{i-1,j}}{2h} \qquad \& \qquad \frac{\partial u}{\partial t} = \frac{u_{i,j+1} - u_{i,j-1}}{2h}$$

and that for the second derivative becomes

$$\frac{\partial^2 f}{\partial x^2} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} \qquad \& \qquad \frac{\partial^2 f}{\partial t^2} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}$$

where $u_{i,j} = u(x_i, t_j)$.

We now substitute the central difference formulas for first and second derivative into Burger's equation, to get

$$\frac{u_{i,j+1} - u_{i,j-1}}{2h} + u\left(\frac{u_{i+1,j} - u_{i-1,j}}{2h}\right) = \nu\left(\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}\right)$$

After doing some algebra we get

$$u_{i,j+1} - u_{i,j-1} = \frac{2\nu}{h}(u_{i+1,j} + u_{i-1,j}) - u_{i,j}(u_{i+1,j} - u_{i-1,j} + 4\nu)$$

We use recursion to obtain the values of all $u_{i,j}$.

2.4 Examples

1. **Example 1**
   Let us consider Burger's equation with $\nu = 1$,

   $$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \frac{\partial^2 u}{\partial x^2}$$

   where, $x \in [0, 2]$ We take linear initial condition,

   $$u(x, 0) = x \text{ for all } t > 0$$

   and boundary conditions,

   $$u(0, t) = 0$$
   $$u(2, t) = 2$$

   Analytic solution of above equation is given by,

   $$u(x, t) = \frac{x}{1 + t}$$

   Using finite difference method, We obtained the solution as below,

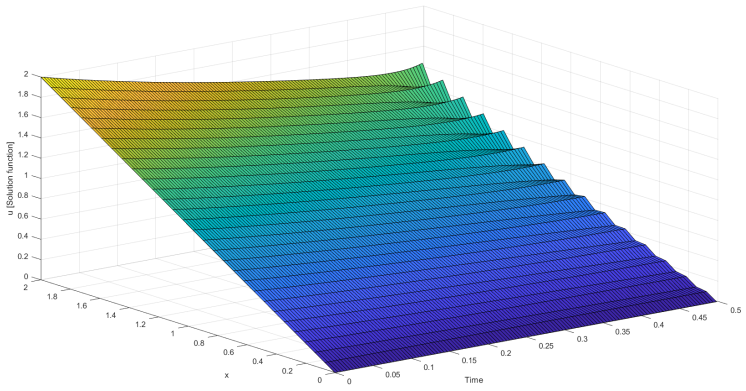


Figure 1: Numerical solution of Burger's equation with linear initial condition

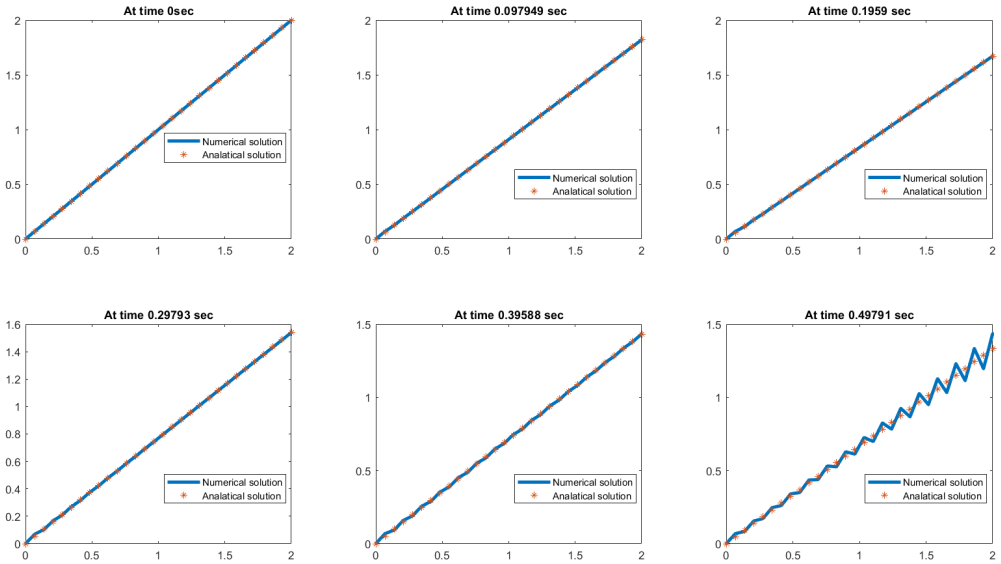


Figure 2: Comparison of Burger's equation with linear initial condition numerical and analytical solution at different time intervals

2. **Example 2**
   We have inviscid burger's equation with $\nu = 0$

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0$$

   where, $x \in [0, 2]$
   We take initial condition,

$$u(x, 0) = \sin \pi x \text{ for all } t > 0$$

   and boundary conditions,

$$u(0, t) = 0$$

$$u(2,t) = 0$$

Solving the equation numerically using finite difference method gives us solution as,

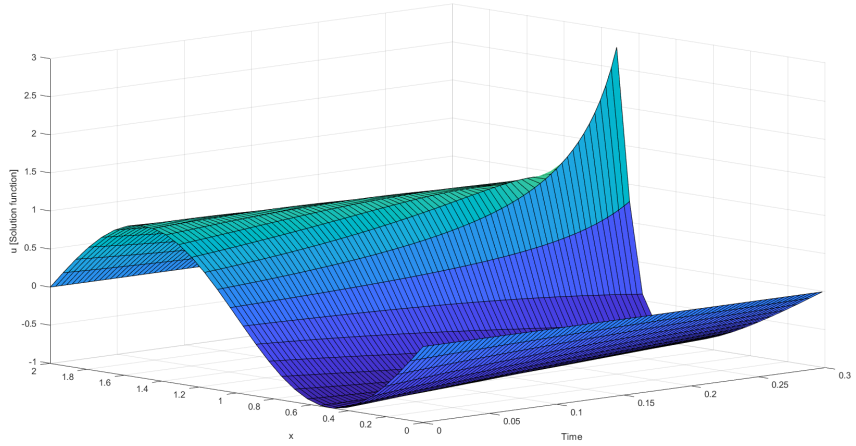


Figure 3: Numerical solution of Burger's equation with initial condition, $\sin \pi x$

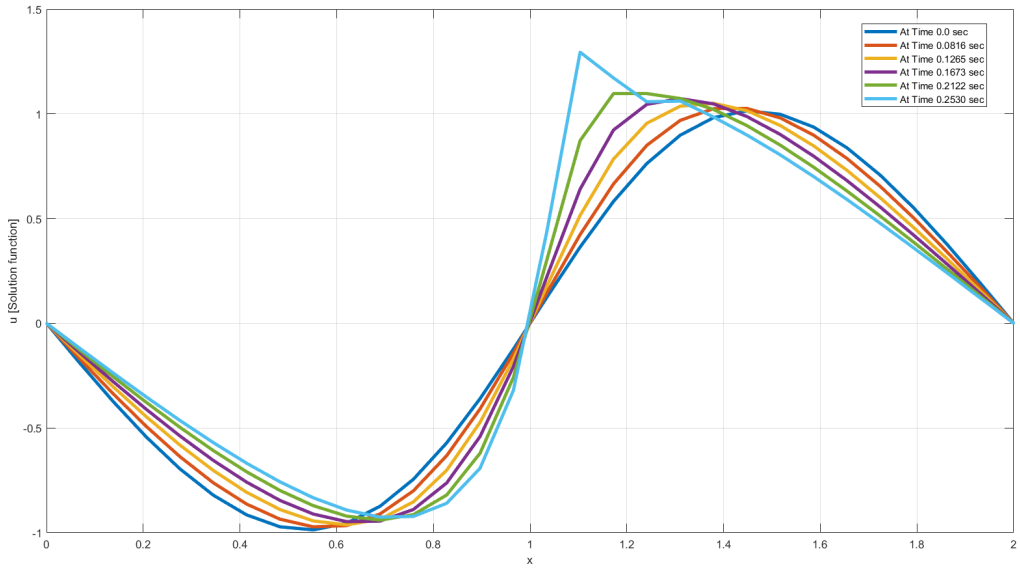


Figure 4: Numerical solution of Burger's equation with initial condition, $\sin \pi x$ at different time intervals

As we can see in both examples, FDM gives exact solution till a specific time domain and starts to diverge as we go beyond that. Required condition for Finite difference method to work is $r = \dfrac{dt}{dx^2} \leq 0.5$

2.5 Cole-Hopf Transformation

Surprisingly, the Burgers' equation can be converted into the Heat equation (a linear partial differential equation that can be easily solved analytically) by a non-linear transformation, known as **Cole-Hopf transformation**. This is a rather remarkable feat achieved by Julian Cole and Eberhard Hopf.

We have the Burgers' equation

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}$$

To transform a nonlinear equation to a linear equation is a difficult problem, but doing a nonlinear transformation on a linear equation is a comparatively easy task. Hence we start with the Heat equation and perform a nonlinear transformation on it. The Heat equation is given by

$$\frac{\partial p}{\partial t} = \nu\frac{\partial^2 p}{\partial x^2}$$

where $p = p(x,t)$. The nonlinear transformation that we will be performing on this equation is the following

$$p = e^{a\phi}$$

where a is a constant and $\phi = \phi(x,t)$. From the above expression of p, we have

$$\phi = \frac{1}{a}\ln p$$

we also evaluate the following expressions using chain rule:

$$\frac{\partial p}{\partial t} = ae^{a\phi}\frac{\partial \phi}{\partial t} \qquad \& \qquad \frac{\partial p}{\partial x} = ae^{a\phi}\frac{\partial \phi}{\partial x} \qquad \& \qquad \frac{\partial^2 p}{\partial x^2} = ae^{a\phi}\frac{\partial^2 \phi}{\partial x^2} + a^2 e^{a\phi}\left(\frac{\partial \phi}{\partial x}\right)^2$$

Substituting the above expressions in the Heat equation, we get

$$\frac{\partial \phi}{\partial t} = \nu\frac{\partial^2 \phi}{\partial x^2} + \nu a\left(\frac{\partial \phi}{\partial x}\right)^2$$

Differentiating the above equation w.r.t. x, we get,

$$\frac{\partial^2 \phi}{\partial x \partial t} = \nu\frac{\partial^3 \phi}{\partial x^3} + 2\nu a\left(\frac{\partial \phi}{\partial x}\right)\left(\frac{\partial^2 \phi}{\partial^2 x}\right)$$

Substituting $\frac{\partial \phi}{\partial x} = u$ in the above equation, we get

$$\frac{\partial u}{\partial t} = \nu\frac{\partial^2 u}{\partial x^2} + 2\nu a u\frac{\partial u}{\partial x}$$

If we substitute $a = -1/2\nu$ in the above equation, we get

$$\frac{\partial u}{\partial t} = \nu\frac{\partial^2 u}{\partial x^2} - u\frac{\partial u}{\partial x}$$

which is essentially the Burgers' equation. Hence the Cole-Hopf transformation

$$u = \frac{\partial \phi}{\partial x} = \frac{\partial}{\partial x}(-2\nu \ln p) = -2\nu\frac{\partial p}{\partial x}\frac{1}{p}$$

transforms the nonlinear Burgers' equation to the linear Heat equation, which can be easily solved analytically and also numerically. Below we demonstrate the numerical method of solving the Heat equation.

2.6 Numerically Solving the Heat Equation

We solve the Heat equation using Finite Difference Method. We apply the Forward Difference method to evaluate the first derivative of p w.r.t. t and the Central Difference formula to evaluate the second derivative of p w.r.t. x.

$$\frac{\partial p}{\partial t} = \frac{p_{i,j+1} - p_{i,j}}{\Delta t} \qquad \& \qquad \frac{\partial^2 p}{\partial x^2} = \frac{p_{i+1,j} - 2p_{i,j} + p_{i-1,j}}{\Delta x^2}$$

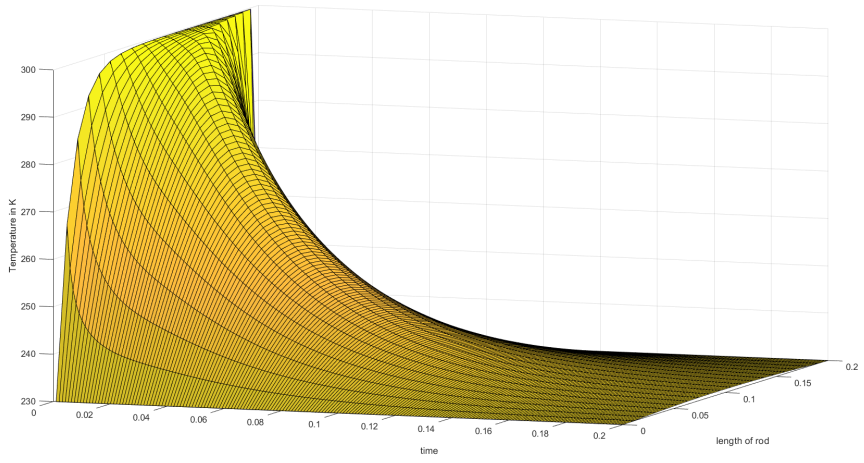
We now substitute these expressions in the Heat equation to get,

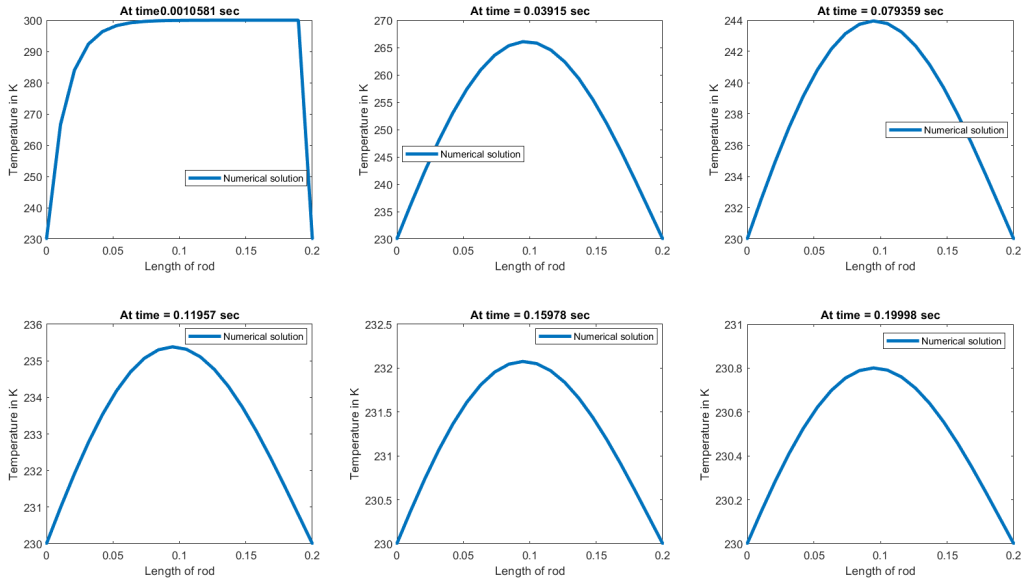
$$\frac{p_{i,j+1} - p_{i,j}}{\Delta t} = \nu \left[\frac{p_{i+1,j} - 2p_{i,j} + p_{i-1,j}}{\Delta x^2} \right]$$

After some algebra, we get,

$$p_{i,j+1} = \nu \Delta t \left[\frac{p_{i+1,j} - 2p_{i,j} + p_{i-1,j}}{\Delta x^2} \right] + p_{i,j}$$

We use this relation to numerically solve the Heat equation in Maltab. We consider a rod of length L which is initially at 300K and impose the boundary condition that the ends of the rod are at 230K. We expect that as time passes, the graph will start to diffuse from the corners slowly. At time t, our graph looks like





which is as expected. Hence we have numerically solved the Heat equation using the Finite Difference method and Matlab.

3 Using Multi Layered Perceptrons for Solving Non-Linear PDEs

3.1 Pretext

Multi Layered Perceptrons or more popularly Neural Networks as we know them today can be employed to solve any type of Ordinary, partial differential equations. The most striking feature of using Neural networks to solve a differential equation is that they will give us continuous and differentiable functional solution as opposed to some discreet array of points in space like Euler, RK4, finite difference methods. This is a relatively novel method for solving PDEs and ODEs with a lot of technical nuances which need to be studied in future. These methods are also better in terms of interpolations as compared to the orthodox numerical methods. In this report we have designed a 2 hidden layered Neural network which is employed for solving the Burgers equation. Due to the shockwave formation properties in Burger's equation, the Solution has been analyzed for linear initial condition in a domain where we know the equation has a smooth solution.

3.2 Introduction

A wealth of finite difference methods is available in the literature in order to solve ordinary differential equations (ODEs) and Partial Differential Equations(PDEs). These methods, whether with single or multiple steps, provide the solution at discrete points in the time domain. In order to obtain continuous approximations with nice generalization properties, research has been carried out to study the approximation of the solution of systems of ordinary and partial differential equations by means of neural networks. In this brief, we are going to present and analyze a new method to solve PDEs in the whole interval of study by means of a neural approximator in the form of a feedforward perceptron with 2 hidden layers. The network $\mathcal{N}$ will be trained in order to give an approximation of the true solution z of the ODE. As we know that neural networks are popularly known as universal function approximators and given a set of data in form of discrete points and in our case a differential equation, a Neural Network can approximate a functional solution arbitrarily close to

the original solution of the original differential equation. The non linearity of the final solution is captured by using some standard non linear functions (popularly known as activation functions) like sigmoid as:

$$sigmoid(x) = \frac{1}{1 + exp(-x)}$$

or tanh defined as:

$$tanh = \frac{exp(x) - exp(-x)}{exp(x) + exp(-x)}$$

convergence of the solution is guranteed by Banach Fixed point Theorem as the activations used are compression maps. The idea is to note that the perceptron is basically a linear combination of differentiable functions and thus, its derivative with respect to its inputs is well defined. To be more specific, a multi-layer perceptron $\mathcal{N}$(x,t) can be differentiated with respect to t and x.

3.3 Description of method

The proposed approach will be illustrated in terms of the following general differential equation definition:

$$G(\vec{x}, \psi(\vec{x}), \nabla\psi(\vec{x}), \nabla^2\psi(\vec{x})) = 0, \vec{x}\epsilon D \qquad (1)$$

subject to certain boundary conditions (B.Cs) (for instance Dirichlet and/or Neumann), where $\vec{x} = (x_1, ..., x_n)\epsilon R_n, D \subset R_n$ denotes the definition domain and $\psi(\vec{x})$ is the solution to be computed. The proposed approach can be also applied to differential equations of higher order, but we have not considered any problems of this kind in the present work. To obtain a solution to the above differential equation the collocation method is adopted which assumes a discretization of the domain D and its boundary S into a set points $\hat{D}$ and $\hat{S}$ respectively. The problem is then transformed into the following system of equations:

$$G(\vec{x}, \psi(\vec{x}), \nabla\psi(\vec{x}), \nabla^2\psi(\vec{x})) = 0, \forall\vec{(x_i)}\epsilon\hat{D} \qquad (2)$$

subject to the constraints imposed by the B.Cs. If $\psi_t(\vec{x}, \vec{p})$ denotes a trial solution with adjustable parameters $\vec{p}$, the problem is transformed to:

$$min_{\vec{p}} \sum G(\vec{x}, \psi(\vec{x}), \nabla\psi(\vec{x}), \nabla^2\psi(\vec{x}))^2 \qquad (3)$$

subject to the constraints imposed by the B.Cs. In the proposed approach the trial solution ψ_t employs a feedforward neural network and the parameters $\vec{p}$ correspond to the weights and biases of the neural architecture. We choose a form for the trial function $\psi_t(x)$ such that by construction satisfies the BCs. This is achieved by writing it as a sum of two terms:

$$\psi_t(x) = A(\vec{x}) + F(\vec{x}, N(\vec{x}, \vec{p})) \qquad (4)$$

where $N(\vec{x}, \vec{p})$ is a single-output feedforward neural network with parameters $\vec{p}$ and n input units fed with the input vector $\vec{x}$. The term $A(\vec{x})$ contains no adjustable parameters and satisfies the boundary conditions. The second term F is constructed so as not to contribute to the BCs, since $\psi_t(x)$ must also satisfy them. This term employs a neural network whose weights and biases are to be adjusted in order to deal with the minimization problem. Note at this point that the problem has been reduced from the original constrained optimization problem to an unconstrained one (which is much easier to handle) due to the choice of the form of the trial solution that satisfies by construction the B.Cs. In the next section we present a systematic way to construct the trial solution, i.e. the functional forms of both A and F. We treat several common cases that one frequently encounters in various scientific fields. As indicated by our experiments, the approach based on the above formulation is very effective and provides in reasonable computing time accurate solutions with impressive generalization (interpolation) properties.

3.4 Gradient Computation

The efficient minimization of equation (3) can be considered as a procedure of training the neural network where the error corresponding to each input vector $\vec{x}$ is the value $G(x_i)$ which has to become zero. Computation of this error value involves not only the network output (as is the case in conventional training) but also the derivatives of the output with respect to any of its inputs. Therefore, in computing the gradient of the error with respect to the network weights, we need to compute not only the gradient of the network but also the gradient of the network derivatives with respect to its inputs. Consider a multilayer perceptron with n input units, one hidden layer with H sigmoid units and a linear output unit. The extension to the case of more than one hidden layers can be obtained accordingly. For a given input vector $\vec{x} = (x_1, ..., x_n)$ the output of the network is N = $\sum_{i=1}^{H} v_i \sigma(z_i)$ where $z_i = \sum_{j=1}^{N} w_{ij} x_j + u_i$, w_{ij} denotes the weight from the input unit j to the hidden unit i, v_i denotes the weight from the hidden unit i to the output, u_i denotes the bias of hidden unit i and $\sigma(z)$ is the sigmoid transfer function. It is straightforward to show that:

$$\frac{\partial^k N}{\partial x_j^k} = \sum_{i=1}^{H} v_i w_{ij}^k \sigma_i^{(k)} \tag{5}$$

where $\sigma_i = \sigma(z_i)$ and $\sigma(k)$ denotes the kth order derivative of the sigmoid. Moreover it is readily verifiable that:

$$\frac{\partial^{\lambda_1}}{\partial x_1^{\lambda_2}} \frac{\partial^{\lambda_2}}{\partial x_2^{\lambda_2}} .... \frac{\partial^{\lambda_n}}{\partial x_n^{\lambda_n}} = \sum_{i=1}^{n} v_i P_j \sigma_i^{(\Lambda)}$$

$$P_i = \prod_{k=1}^{n} w_{ik}^{\lambda_k} \tag{6}$$

Equation (5) indicates that the derivative of the network with respect to any of its inputs is equivalent to a feedforward neural network $N_g(\vec{x})$ with one hidden layer, having the same values for the weights $w_i j$ and thresholds u_i and with each weight v_i being replaced with $v_i P_i$. Moreover the transfer function of each hidden unit is replaced with the λ^{th} order derivative of the sigmoid. Therefore the gradient of N_g with respect to the parameters of the original network can be easily obtained as:

$$\frac{\partial N_g}{\partial v_i} = P_i \sigma_i^{(\Lambda)}$$

$$\frac{\partial N_g}{\partial u_i} = v_i P_i \sigma_i^{(\Lambda+1)}$$

$$\frac{\partial N_g}{\partial w_{ij}} = x_j v_i P_i \sigma_i^{(\Lambda+1)} + v_i \lambda_j w_{ij}^{\lambda_j - 1} \Big(\prod_{k=1, k \neq j} w_{ik}^{\lambda_k} \Big) \sigma^{(\Lambda)_i}$$

Once the derivative of the Loss function is calculated wrt different weight parameters, we then have to apply Gradient descent which adjusts the weights parameters such as to minimize our loss function. The general formula for gradient descent can be written as:

$$b = a - \gamma \nabla f(a)$$

What this formula does is calculates the gradient of our Neural Network function w.r.t. various weight parameters in our network, it then walks into the direction of the gradient, It continuously keeps doing so until it has found the absolute minimum of the function. While the system is learning the weights we are never sure when our system has reached the absolute minimum. To take care of this, we run our iterations to a sufficient number of epochs such that the loss could be minimized. Here γ is the learning rate which

can be adjusted accordingly and the gradient is w.r.t. the different weight values in our neural network. the adjustment of hyper parameters takes hit and trials and then verifications to find the best learning rate the only way is to run the network and check for some standard values which lie in range(0.01-0.1). The learning rate we used here is 0.1. Learning rate basically adjusts how fast our gradients move. There are a lot of methods for gradient descent like Stochastic Gradient Descent, Gradient Descent with momentum (where the learning rate is changed in every iteration as the neural network learns learning rate drops because the now the gradients need to be shifted by smaller and smaller amount to reach to the minima). Sometimes our network may get stuck in some local minima so in that case, if we find that our model is not learning significant amount of information, we can increase the learning rate. Using an adaptable learning rate is the best option in many cases. In our model we have used ADAM(adaptive moment estimation) optimizer which is another popular optimization method for Gradient Descent that works on adaptable learning rate with momentum. Now we move to the actual problem of concern which is the Burger's equation.

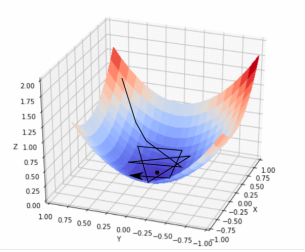


Figure 5: Gradient Descent Visualized

3.5 Solving Burger's equation using Neural Networks

To solve the Burger's equation, we transform the equation into initial value problem. The burger's equation is written as:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}$$

For simplicity purposes, we use the inviscid burger's equation, where the diffusion term (ν) is absent and we use linear initial condition as we know that analytical solutions exist for this:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0 \qquad\qquad u(x,0) = x$$

Subrahmanyan Chandrasekhar provided the explicit solution in 1943 when the initial condition is linear, i.e., $f(x) = ax + b$, where a and b are constants. The explicit solution is:

$$u(x,t) = \frac{ax+b}{at+1}$$

Let us consider $NN(x,t)$ be our neural network. We want to fit this neural network as our solution $u(x,t) = NN(x,t)$. That means $N(x,t)$ should satisfy the differential equation $NN_t + NN_x = 0$. Now there is one important thing that we should not miss is the initial condition. If we want our assumed solution to satisfy the initial condition as well we can force our neural network solution on those initial conditions. We can safely assume our solution to be $u(x,t) = x + tNN(x,t)$ instead. We see that this choice of solution satisfies the initial conditions as well. In some cases, constructing the function according to initial conditions can be cumbersome, so another approach we can take is: assume the solution to be $u(x,t) = NN(x,t)$ simply and then force the boundary conditions on the error function. In that case we have to minimize the following function.

$$Loss = (NN_t + NN_x)^2 + (NN(x,0) - x)^2$$

If however we force the initial/boundary value conditions on our assumed solution itself then, the function to minimize is the following:

$$Loss = (u_t + uu_x)^2$$

where $u(x,t) = x + tNN(x,t)$

3.6 Solution by fitting into Neural Network

Neural networks (NNs) are a collection of nested functions that are executed on some input data. These functions are defined by parameters (consisting of weights and biases), which in PyTorch are stored in tensors.

To solve the differential equation, we have used the Pytorch library. To compute the gradients, we have used the autograd function that calculates a Vector Jacobain product where the Jacobian contains the partial derivatives w.r.t. the weight paramaters and it keeps storing those values in a computational graph. We consider a 2 layered Neural Network. The initial weight matrices are of the dimensions: $W_0[2][20], W_1[20][20], W_2[20][1]$. Also, we take basis values as well which are one dimensional arrays $b_0[20], b_1[20], b_2[20]$ that take care of the shifts from origin for the function. The weights and biases are initialized with The xavier initialization method. Xavier initialization is calculated as a random number with a uniform probability distribution (U) between the range $-(\frac{1}{\sqrt{n}})$ and $(\frac{1}{\sqrt{n}})$, where n is the number of inputs to the node.

Our Neural Network Architecture looks like this:

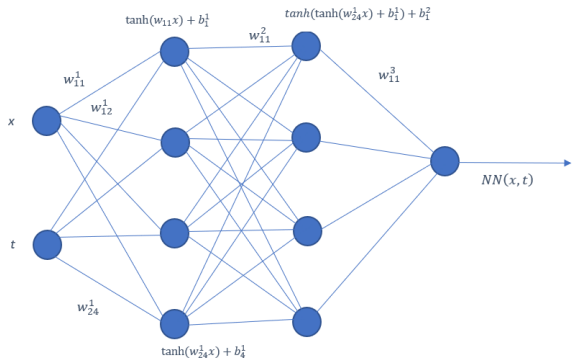


Figure 6: A toy model of a simple 2 layered Neural Net used to solve a PDE. Here, with 4 nodes in each hidden layer

3.7 Result

The domain in which the solution was calculated was chosen to be $x\epsilon(0,2)$ and $t\epsilon(0,2)$. We took 50 values of x and t each in the domain for iteration. The Neural Net was trained over 500 epochs. The loss initially fluctuated and then started decreasing. In the first run, the loss (Mean squared error) finally settled to 0.1769 and in the second run it settled to 0.679. The difference comes because of the random initialization of the weights. Sometimes the Network is able to find the direction in which it will find the Absolute minima easily. At other times, the system loss has to fluctuate continuously to eventually be able to find the direction of minimum. One way or the other, the system finally is able to find the direction of minimum loss and the loss

converged in all the test cases. The output obtained was a smooth and continuous function over the whole domain of consideration.

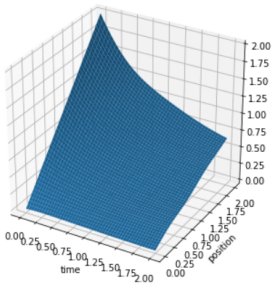


Figure 7: The solution approximated by the neural network plotted for given initial condition

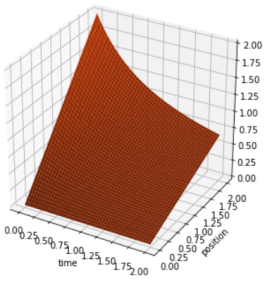


Figure 8: Analytical Solution: u = x/(t+1)

The loss can be further reduced by increasing the number of layers. The solution was checked for 3 different activation functions: sigmoid, tanh. Where sigmoid gave better performance than tanh in terms of loss convergance. Sihgmoid and tanh suffer from vanishing gradients problem as we take gradients the derivatives of sigmoid and tanh tend to zero, however tanh handles the vanishing gradient problem a bit better than as sigmoid does.

The loss curve/learning curve over 500 epochs is shown below.

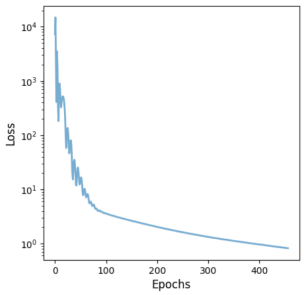


Figure 9: The loss curve for inviscid Burgers equation fitted to a neural net

Plotting the solution's position profile for various timestamps:

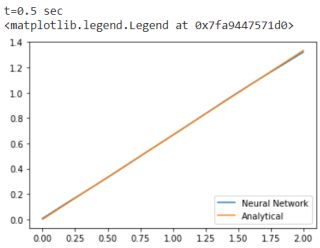


Figure 10: t=0.5 sec

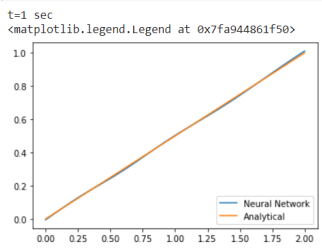


Figure 11: t=1.0 sec

Hence we can see from the solution profile that for this method the error doesn't propagate as the time

increases, it is due to the fact that we have solved the problem to obtain a functional solution by setting boundary conditions beforehand and the function is approximated within those boundaries only.

4 Conclusion

Hence we find that both the methods give very accurate results in the selected time domain. The results produced by finite difference method start to show diversion from the actual value. This is because we are breaking the derivatives into linear equations and using the previous value to calculate the next value in the discrete interval. This means that if an error occurs (which can be due to truncation, disretization etc.) the error can propagate and as the interval of consideration grows error also magnifies and hence gives bad results for larger intervals. The good thing about this method is it's speed. As we are using matrix to populate the values of different discreet values of function at different points the effective time complexity is of the order of $O(n^2)$ where n are the number of operations. The method is really fast and gives results almost instantaneously. In contrast if we compare the method of Neural networks, it exceeds the finite difference in the sense that error doesn't propagate as we increase the domain of consideration. The method is also good in a sense that it gives functional solutions to a differential equation by approximating to arbitrary accuracy, a non-linear/linear (depending upon case) for the differential equation. The department where this method struggles is it's ridiculously high time and space complexity which is due to matrix multiplication operations of high orders then the training process where we have to run it for longer epochs. This tradeoff between the accuracy and speed can be the major factor on deciding between which method to use depending upon the situation and use case.

References

[1] *Numerical solution of one-dimensional Burgers equation: explicit and exact-explicit finite difference methods*
S. Kutluay, A.R. Bahadir, A. Ozdes

[2] *ANALYTIC SOLUTIONS OF THE VECTOR BURGERS' EQUATION*
Steven Nerney, Edward J. Schmahl , and Z. E. Musielak

[3] *A REVIEW OF NUMERICAL METHODS FOR NONLINEAR PARTIAL DIFFERENTIAL EQUATIONS*
EITAN TADMOR

[4] *On a Neural Approximator to ODEs* Cristian Filici

[5] *Artificial Neural Networks for solving Ordinary and Partial Differential Equations*
I.E. Lagari, A.Likas, D.I. Fotiadis

[6] *The Cole-Hopf And Miura Transformations*
Fritz Gesztesy, Helge Holden