Physics-Informed Neural Networks (PINNs)

Kostinoudis Evangelos (112)
Aristotle University of Thessaloniki
Thessaloniki, Greece
ekostino@csd.auth.gr

Nousias Georgios (120)
Aristotle University of Thessaloniki
Thessaloniki, Greece
nousiasg@csd.auth.gr

Palaskos Achilleas (113)
Aristotle University of Thessaloniki
Thessaloniki, Greece
achillap@csd.auth.gr

Spanos Georgios (112)
Aristotle University of Thessaloniki
Thessaloniki, Greece
gspanosd@csd.auth.gr

***Abstract*—Nowadays, the presence of differential equations in the fields of Science and Engineering is quite prevalent. In this paper, a recent methodology, known as Physics-Informed Neural Networks (PINNs) is used to solve some Partial and Ordinary Differential Equations (PDEs and ODEs). PINNs take advantage of the automatic differentiation (Backpropagation algorithm), that various software packages provide, in order to calculate the derivatives of the output with respect to the input. Then, the problem is converted into an optimization problem, where the loss function is the Mean Squared Error (MSE) of the differential equation. The initial and the boundary conditions are also incorporated into the loss function. By minimizing it, an approximate solution is attained. Moreover, the results are compared with the corresponding analytical solutions, were available. The same equations are solved using the ode45 and pdepe solvers of MATLAB, as well, and the solutions are compared with the PINNs' solutions. Finally, it is observed that PINNs manage to solve these equations quite efficiently and with high accuracy, making them a promising technique for much more difficult problems in the future.**

Keywords— Physics-Informed Neural Networks (PINNs), Artificial Neural Networks (ANNs), Partial Differential Equations (PDEs), Ordinary Differential Equations (ODEs), ode45, pdepe, analytical solution, derivative, Feed-Forward neural network, Universal Approximation Theorem, Backpropagation, Chain Rule, loss/cost function, Mean Squared Error (MSE), Science, Engineering, Diffusion equation, Burger's equation, Initial/Boundary conditions, PyTorch, MATLAB, Python.

I. Introduction

Physics-Informed Neural Networks (PINNs) are a type of machine learning approach that combines the power of neural networks with the principles of physics. PINNs were proposed to solve Ordinary Differential Equations (ODEs) and Partial Differential Equations (PDEs).

Traditional methods for solving PDEs often rely on numerical techniques such as finite difference, finite element, or spectral methods. While these methods can be effective, they often require a substantial amount of computational resources and struggle to handle complex geometries or uncertain boundary conditions. PINNs offer an alternative approach by leveraging the expressive power of neural networks to approximate the underlying physics and solve PDEs more efficiently.

The proposed methodology is taking advantage of automated differentiation of Artificial Neural Networks (ANNs). More specifically, the calculation of the derivatives of ANNs with respect to the inputs is possible. The initial idea was proposed in the paper of Lagaris et al [1], where the derivatives are calculated using the Backpropagation algorithm and the Chain rule.

PINNs have demonstrated promising results in various scientific and engineering applications. They have been successfully applied to problems in fluid dynamics, heat transfer, structural mechanics, electromagnetics, and many other domains. PINNs offer advantages such as reduced computational costs, flexibility in handling complex geometries, and the ability to incorporate prior knowledge about the problem.

II. Related work

In Raissi et al [2] the concept of PINNs was introduced and demonstrated their effectiveness in solving both forward and inverse problems governed by nonlinear PDEs. PINNs were applied to various physical systems, including diffusion, wave, and Navier-Stokes equations. They highlighted the ability of PINNs to handle complex geometries and sparse data while maintaining high accuracy. Also, in [3] the aim of the authors is focused to set the foundations for a new paradigm in modeling and computation that enriches deep learning with the long-standing developments in mathematical physics. In [5] PINNs are combined with generative adversarial networks (GANs) to solve stochastic PDEs. The authors proposed a physics-informed GAN framework that incorporates the physics constraints into both the generator and discriminator networks. The method demonstrated improved stability and accuracy in solving stochastic PDEs, making it suitable for problems with inherent randomness.

III. Proposed work

In order to compare the proposed methodology with already implemented methodologies like ode45 and pdepe built-in MATLAB functions, we calculate the error that the above functions achieve and compare it with our implemented PINNs methodology. Where possible, the analytical solutions of the corresponding differential equations are also provided and compared with the aforementioned techniques. Experiments were conducted for various types of differential equations like simple ODEs, second order ODEs, the diffusion equation and the Burger's equation.

A. ODE45

The ode45 solver, used in MATLAB, is built upon an explicit Runge-Kutta (4,5) [8] formula known as the Dormand-Prince pair. This solver belongs to the class of

single-step solvers, which means that when computing the solution $y(t_n)$ at a given time point t_n, it only requires the solution at the immediately preceding time point, $y(t_n - 1)$ [6], [7].

The Dormand-Prince pair is a widely used numerical method for solving ODEs, due to its high accuracy and efficiency. It is a fourth-order method with an embedded fifth-order approximation, allowing for error estimation and adaptive step size control.

B. Pdepe

The ode15s solver is utilized for the time integration process in pdepe. The pdepe function takes advantage of ode15s' capabilities in handling differential-algebraic equations that arise when the PDE contains elliptic equations, as well as dealing with Jacobians that have a specified sparsity pattern.

When the PDE is discretized, the elliptic equations are transformed into algebraic equations. If the elements of the initial-conditions vector associated with the elliptic equations are not consistent with the discretization, pdepe attempts to adjust them before initiating the time integration. Consequently, the solution obtained for the initial time may exhibit a discretization error similar to that at any other time.

With a sufficiently fine mesh, pdepe can find consistent initial conditions that are close to the provided values. However, if pdepe encounters difficulty in finding consistent initial conditions, refining the mesh can be attempted. It's important to note that no adjustment is required for elements of the initial conditions vector corresponding to parabolic equations.

C. PINNs

In this paper, a new method for solving differential equations will be presented, which is quite different from all the existing techniques. As it is known, in almost all of engineering and science fields differential equations of one form or other are quite predominant. Therefore, searching for more efficient techniques to solve them is highly desirable.

In particular, ANNs we will be used to solve ODEs or PDEs. But the important question is whether can ANNs actually be applied to solving PDEs. The answer is positive, but the way ANNs are used towards this direction is quite more different than their usual modes of use. For example, there will not be a training, a validation and a test set.

As it has been already mentioned the main idea behind this method goes back to a set of papers by Lagaris et. al. which were published in the late of 90s, early 2000s. In this case, the equations are not discretized as most finite difference methods do, where an approximate equation is created instead, and no surrogate model is utilized. On the other hand, the approach is entirely different. In particular, every PDE or ODE is converted into an optimization problem. However, the deep idea behind PINNs is to move towards full automation of solving differential equations using ANNs. That is, first the PDE/ODE is solved using ANNs and then a surrogate ANN model is built for the previous ANN.

To present this new method, it would be better to start with a simple example. Consider, for instance the following 2nd order ODE:

$$u_{xx} + au_x = b \quad (1)$$

Subject to the boundary conditions:

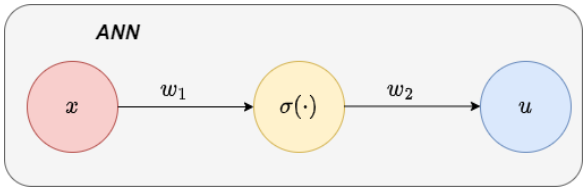
$$u(0) = u_0 \quad (2)$$
$$u(1) = u_1 \quad (3)$$

In order to solve the above ODE, it is assumed that u is some ANN that takes x as input and provides $\hat{u}$ as output, i.e. an ANN is used to approximate the solution u with $\hat{u}$:

$$\hat{u} \cong ANN(x) \quad (4)$$

This approach is valid, because from the Universal Approximation Theorem it is known that the solution u can always be approximated arbitrarily closely by an ANN. Since that is possible, it can always be assumed that u is some neural network of x. This is very helpful, because in this way u is written in a fully functional form.

To make things simpler, let's suppose that the ANN has the following architecture:



Then:

$$u = w_2\sigma(w_1x) \quad (5)$$

From the above expression, since it is differentiable, the calculation of the derivatives is possible:

$$u_x = w_1w_2\sigma'(w_1x) \quad (6)$$
$$u_{xx} = w_1^2w_2\sigma''(w_1x) \quad (7)$$
$$\vdots$$

The key point is that, similarly, **all derivatives of the output u with respect to the input x can be found**. However, although this is a simple example (only 1 hidden layer with 1 unit), even in the case multiple layers of hidden neurons, all derivatives can be calculated using backpropagation, which takes advantage of the chain rule of calculus. This is also called automatic differentiation and many frameworks, like PyTorch and Tensorflow have inbuilt functions that implement it. In other words, if $\hat{u} \cong ANN(x)$ then we can find u_x, u_{xx} etc.

Taking that into consideration, the previous problem can be posed as follows:

$$minimize\ w \qquad [u_{xx} + au_x - b]^2 \quad (8)$$
$$subject\ to: \qquad u(0) = u_0 \quad (9)$$
$$u(1) = u_1 \quad (10)$$

where:

$$J(w) = [u_{xx} + au_x - b]^2 \quad (11)$$

is the cost function.

Therefore, solving the above constrained minimization problem, an approximate solution can be found. In more

detail, suppose a set of N training points $x_0, x_1, \dots, x_{N-1}$, and that the ANN is initialized with some weights w, then:

1. A forward pass is implemented followed by a backpropagation and the first derivative is found u_x.
2. Next, another forward pass is implemented followed by a backpropagation on the previous graph (graph created in step 1) and the second derivative is found u_{xx}.
3. After steps 1 and 2 are done, the value of the cost function is calculated.
4. This value is backpropagated and the weights of the ANN are updated accordingly using Gradient Descent.

However, the cost function defined is not complete, because the boundary conditions are not included. To incorporate them, our problem can be converted into a **single unconstrained optimization problem** as follows:

$$minimize\ w\ \ [u_{xx} + au_x - b]^2 + [\hat{u}_0 - u_0]^2 + [\hat{u}_1 - u_1]^2 \quad (12)$$

As it is obvious from the previous procedure, this is neither a supervised, because no labels are provided, nor an unsupervised deep learning problem, because we do not try to group the input data. The aforementioned training points can be called the training set but there is neither a validation nor a test set. Therefore, the dataset is provided by the user/solver who tries to solve the differential equation and not from external sources.

Generally, if PDE is the mathematical expression of the partial differential equation such as:

$$PDE = 0 \quad (13)$$

while $IC_1, IC_2, \dots, IC_{N_{IC}}$ the initial conditions and $BC_1, BC_2, \dots, BC_{N_{BC}}$ the boundary conditions, then the problem is formulated as follows:

$$minimize\ w \quad PDE^2 + \sum_{i=1}^{N_{IC}} IC_i^2 + \sum_{j=1}^{N_{BC}} BC_j^2 \quad (14)$$

and the above 4 steps are followed.

IV. Differential Equations

More specifically, the below differential equations with their initial and boundaries conditions, as well as the exact solution (E.S.), when available, were tested and their results are presented in the Results section:

A. Simple ODE

$$u_x - cos\ cos\ (x) = 0, \quad x \in [0, 2\pi)$$

Subject to:

- $u(0) = 0$
- $u(1) = 0$

Analytical solution:

$$u(x) = sin\ (x)$$

B. Second order ODE

$$u_{xx} + au_x + bu + cx = 0, \quad x \in [0, 0.25)$$

For $a = -10, b = 9$ and $c = -5$.

Subject to:

- $u(0) = -1$
- $u_x(0) = 2$

Analytical solution:

$$u(t) = \frac{50}{81} + \frac{5}{9}t + \frac{31}{81}e^{9t} - 2e^t$$

C. Diffusion equation

$$u_t - u_{xx} + (1 - \pi^2)e^{-t}\ sin\ sin\ (\pi x) = 0$$

For $x \in [-1, 1]$ and $t \in [0, 1]$.

Subject to:

1) Initial conditions:

$$u(x, 0) = sin\ (\pi x)$$

2) Boundary conditions:

- $u(-1, t) = 0$
- $u(1, t) = 0$

Analytical solution:

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

D. Burger's equation

$$u_t - uu_{xx} + vu_{xx} = 0$$

For $x \in [-1, 1]$ and $t \in [0, 1]$.

Subject to:

1) Initial conditions:

$$u(x, 0) = -sin\ (\pi x)$$

2) Boundary conditions:

- $u(-1, t) = 0$
- $u(1, t) = 0$

V. Hyperparameters and Results

For Pdepe and ode45 the corresponding in-built functions of MATLAB-R2021 a were used, whereas the PINNs methodology was implemented in Python using the PyTorch framework. All subsequent PINNs experiments were run into Google Colab on the available CPUs.

A. Simple ODE

In order to solve this differential equation a Feed-Forward neural network was used with 5 hidden layers with sizes 50, 50, 20, 50, 50 respectively and $Tanh$ activations, except for the last layer, where the $Linear$ activation function was used. 20,000 was the maximum number of epochs, while the parameter "patience", which indicates the maximum number of epochs, where the loss function does not decrease, was set to 100. The "Adam" optimizer was selected with 10^{-3} learning rate, while no regularization was applied. The model was trained in 2 minutes and the results are shown below:

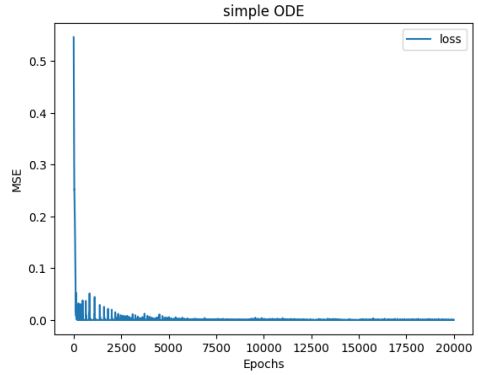


Figure 1

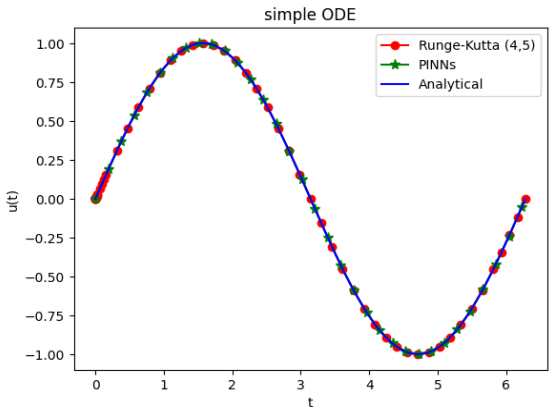


Figure 2

B. *Second order ODE*

For this differential equation a Feed-Forward neural network was used with 3 hidden layers with size 50 each and $Tanh$ activations, except for the last layer, where the $Linear$ activation function was used. The number of epochs was set to 10,000. The "Adam" optimizer was selected with 10^{-3} learning rate, while no regularization was applied. The model was trained in less than a minute and the results are shown following figures:

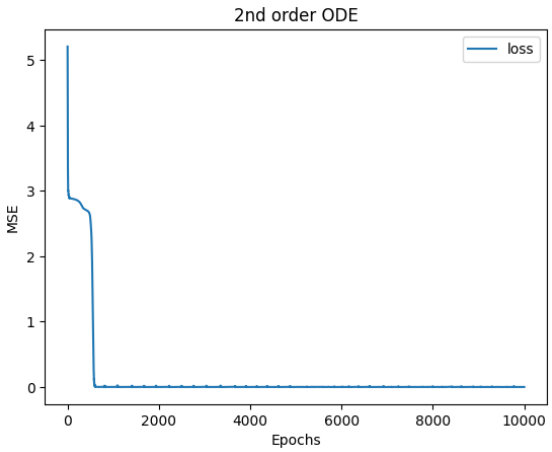


Figure 3

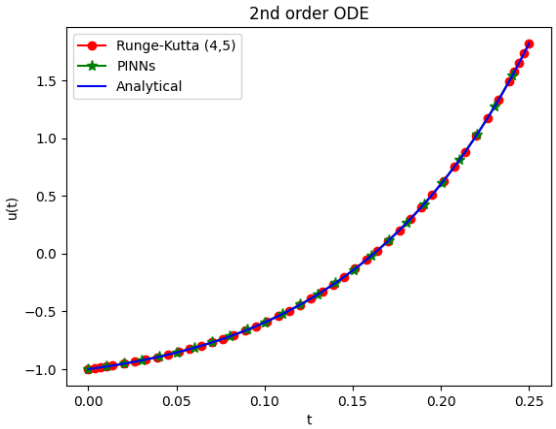


Figure 4

C. *Diffusion equation*

For this differential equation the same hyperparameters as in case B were used. The only difference is that because in this equation there are 2 independent variables x, t, the first layer of the neural network has 2 units, instead of 1. The training last about 40 minutes and the results are shown following figures:

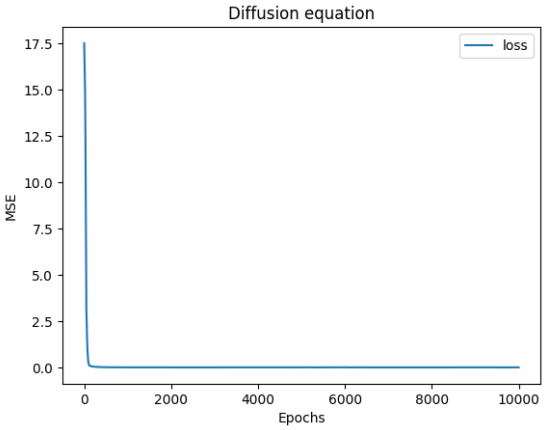


Figure 5

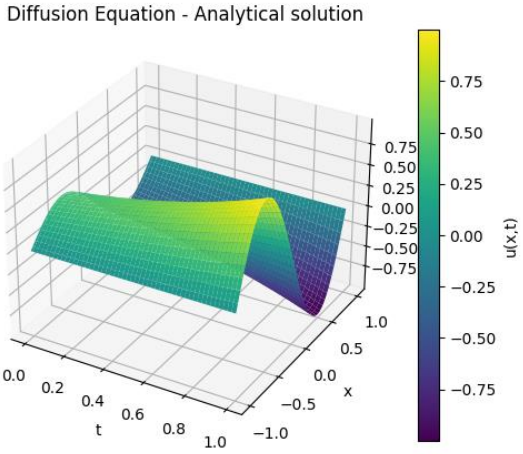


Figure 6

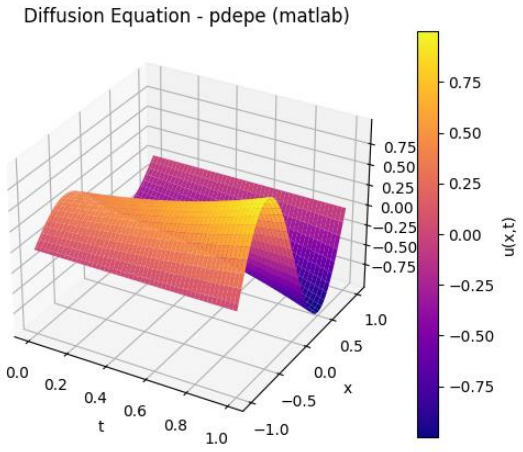


Figure 7

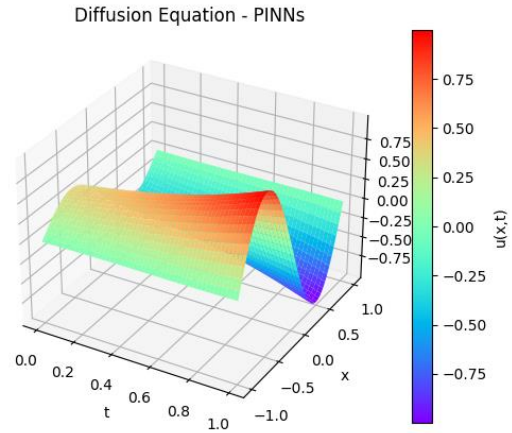


Figure 8

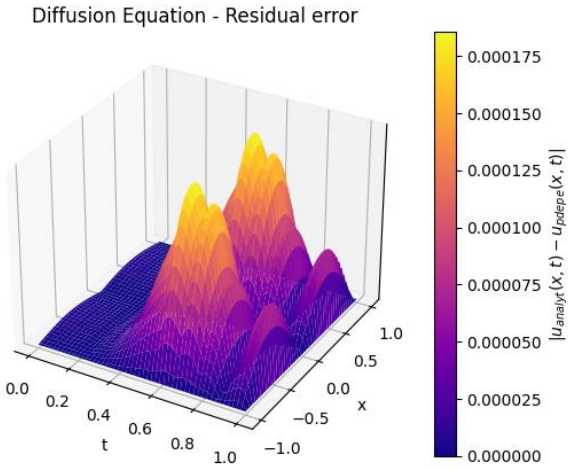


Figure 9

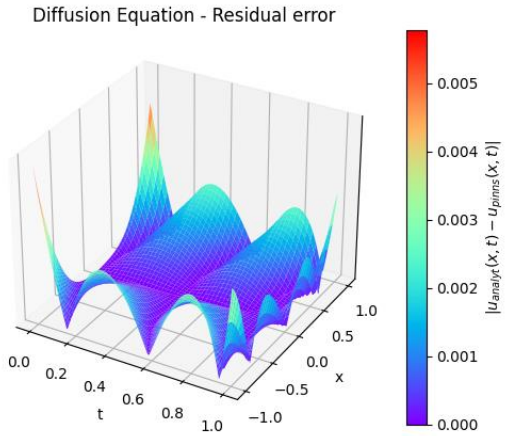


Figure 10

D. Burger's equation

In order to solve the Burger's equation a Feed-Forward neural network was used with 8 hidden layers with sizes 20 each and $Tanh$ activations, except for the last layer, where the $Linear$ activation function was used. The number of epochs was set to 20 epochs. The "LBFG" optimizer was selected with learning rate being set to 1, while no regularization was applied. The whole training session took about 10 minutes to be completed and the results are shown in the next figures:

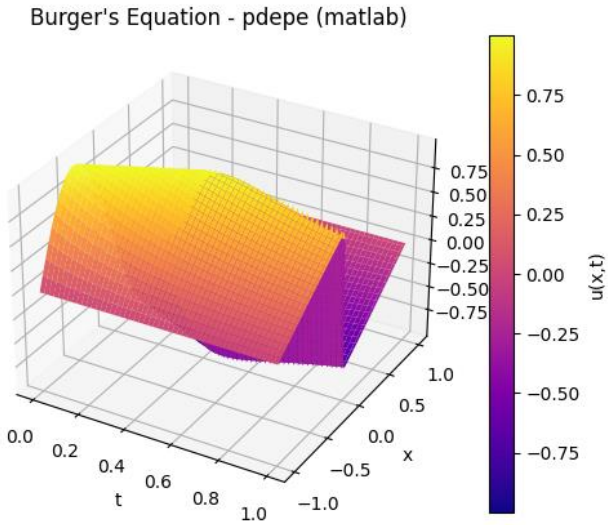


Figure 11

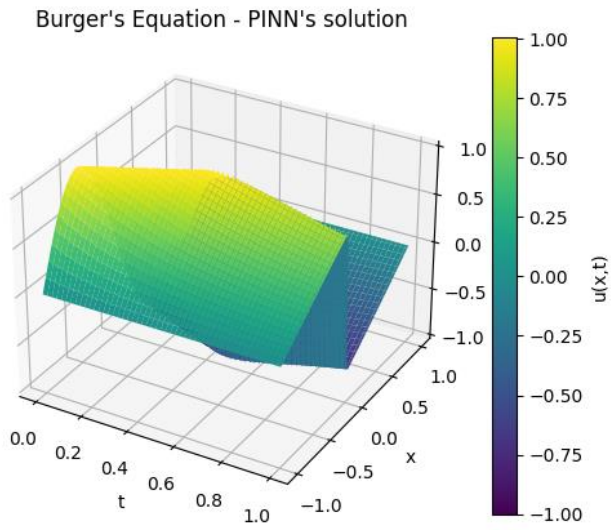


Figure 12

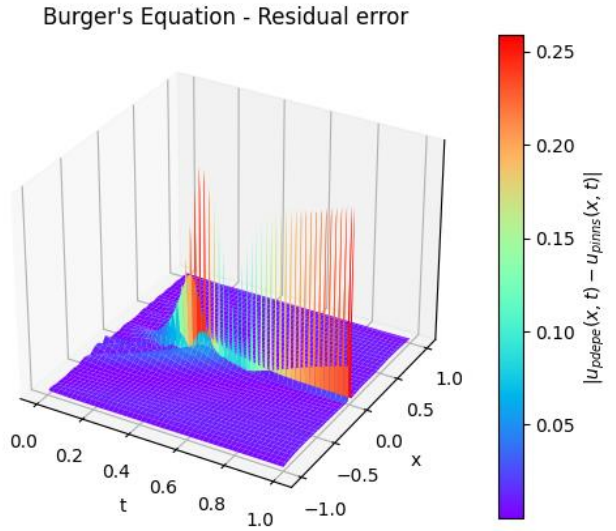


Figure 13

VI. Discussion

In section V, the differential equations were presented in order of increasing difficulty, which means that the Burger's equation was the most difficult to solve. As we can see from the previous diagrams, PINNs manage to solve all the aforementioned equations quite adequately. The only case, where the error is not negligible, is in the solution of the Burger's equation for $x = 0$. As it is shown in figure 13, the error in that case is of the order of the 10^{-1}, which is quite big. However, for the Burger's equation no analytical solution was provided and therefore we can not be sure whether this error comes from the PINN's solution or the pdepe method of MATLAB. On the other hand, in all the other equations the analytical solution was provided and the errors can be considered negligible.

VII. Conclusion

In this paper we proved through multiple experiments that a new technique, called PINN's, can be used very efficiently for solving ODEs and PDEs. In fact, PINNs is a very promising methodology for solving differential equations where either there is not a closed solution or the already existing methods fail. In other words, although PINNs are very sensitive to the selection of their hyperparameters, it is possible that they will be the only sufficient technique for solving more complicated differential equations in the future.

References

[1] Lagaris, I. E., Likas, A., & Fotiadis, D. I. (1998). Artificial neural networks for solving ordinary and partial differential equations. *IEEE transactions on neural networks*, *9*(5), 987-1000.

[2] Sirignano, J., & Spiliopoulos, K. (2018). DGM: A deep learning algorithm for solving partial differential equations. *Journal of computational physics*, *375*, 1339-1364.

[3] Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2017). Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10561*.

[4] Rasht-Behesht, M., Huber, C., Shukla, K., & Karniadakis, G. E. (2022). Physics-Informed Neural Networks (PINNs) for Wave Propagation and Full Waveform Inversions. *Journal of Geophysical Research: Solid Earth*, *127*(5), e2021JB023120.

[5] Yang, L., Zhang, D., & Karniadakis, G. E. (2020). Physics-informed generative adversarial networks for stochastic differential equations. SIAM Journal on Scientific Computing, 42(1), A292-A317.

[6] *Solve nonstiff differential equations — medium order method - MATLAB ode45*. (n.d.). https://www.mathworks.com/help/matlab/ref/ode45.html

[7] Shampine, L. F., & Reichelt, M. W. (1997). The MATLAB ODE Suite. SIAM Journal on Scientific Computing, 18(1), 1-22.

[8] Bogacki, P., & Shampine, L. F. (1996). An efficient runge-kutta (4, 5) pair. *Computers & Mathematics with Applications*, *32*(6), 15-28.

[9] Lu, L., Meng, X., Mao, Z., & Karniadakis, G. E. (2021). DeepXDE: A deep learning library for solving differential equations. *SIAM review*, *63*(1), 208-228.

[10] *DeepXDE — DeepXDE 1.9.2.dev3+g6478131 documentation*. (n.d.). https://deepxde.readthedocs.io/en/latest/?fbclid=IwAR0cGHGmcb91EASvCHlCwdagKHrGf7XiyVF60s5vfWMB_mZN6Em__khAZ6E

[11] NPTEL-NOC IITM. (2019, May 6). *Application 4 - Solution of PDE/ODE using Neural Networks* [Video]. YouTube. https://www.youtube.com/watch?v=LQ33-GeD-4Y