




# Using Scientific Machine Learning to solve Partial Differential Equations

Miles Cochran-Branson 

Department of Physics and Mathematics, Lawrence University, Appleton, WI.

**Keywords:** SciML, Scientific Computing, ML/AI, PINN, PDEs

## Abstract

Machine learning and artificial intelligence have been used with great success in data science, mathematics, and physics as well as numerous other fields to solve complex or previously impossible problems. Computationally intensive problems in physics such as data analysis using a complex model and little data, or finding numerical solutions to differential equations have traditionally been solved with classical scientific computing techniques such as Taylor-series expansions or approximations using polynomials, which in some way reduce the problem to solving large systems of linear equations. Examples include finite element and finite difference methods. Recent efforts have gone into replacing these techniques with neural networks and machine learning. Neural networks have the potential to fit any complex model or differential equation both accurately and efficiently. In the following paper, we describe the basic principles of the growing field of Scientific Machine Learning (SciML) as it applies to solving partial differential equations (PDEs). We solve several examples using the new package `NeuralPDE.jl` in the Julia programming language. To stretch the capabilities of this method, we solve the Einstein field equations to obtain the Schwarzschild metric as a test case where the answer is known analytically. To do this required applying the new paradigm *differential programming*. This example demonstrates the power a physics-informed neural network has and alludes to enormous future potential in the field.

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Machine Learning</b>	<b>2</b>
2.1	Neural Networks . . . . .	2
2.2	Training Neural Networks . . . . .	4
2.3	The Universal Approximation Theorem . . . . .	4
<b>3</b>	<b>Solving ODEs with Physics-Informed Neural Networks</b>	<b>4</b>
<b>4</b>	<b>Solving PDEs with Physics-Informed Neural Networks</b>	<b>5</b>
4.1	Methods of discretization . . . . .	6
<b>5</b>	<b>Solving Einstein’s Field Equations to obtain the Schwarzschild Metric using NeuralPDE.jl</b>	<b>7</b>
5.1	Physics Background and Problem Set-up . . . . .	7
5.2	Technical Implementation . . . . .	9
5.3	Results and Discussion . . . . .	11
<b>6</b>	<b>Conclusion</b>	<b>13</b>
<b>A</b>	<b>Example ODE problem solved with a PINN</b>	<b>13</b>
<b>B</b>	<b>Example using NeuralPDE.jl</b>	<b>15</b>
<b>C</b>	<b>Solving Similarity Solution of Navier-Stokes using NeuralPDE.jl</b>	<b>16</b>
C.1	Problem Derivation . . . . .	18
C.2	Solution with PINN and comparison to analytical solution . . . . .	18

## 1. Introduction

Differential equations are the language of the physical world. Newton’s famous second law of motion,

$$F = m \frac{d^2x}{dt^2} = ma \quad (1.1)$$

where  $m$  is the mass of an object and  $x$  is its position, is just one example of a differential equation which describes how an object moves in the presence of a force. For instance, the path of a ball thrown in the air under the influence of gravity obeys this law and the motion of planets *approximately* obeys this law. Many techniques currently exist for solving ordinary differential equations (ODEs)—differential equations in one variable—and partial differential equations (PDEs)—differential equations in more than one variable—both analytically [1] and numerically [2]. Unfortunately, solving these problems in practice is not trivial. In particular, PDEs in most real-world problems are impossible to solve analytically and are very computationally expensive to solve numerically. Enter scientific machine learning (SciML): an interdisciplinary field combining techniques from machine learning and artificial intelligence with scientific computing. SciML has the potential to provide a semi-universal interface for implementing differential equation problems and inverse problems in an efficient and accurate way.

An initiative laid out by the DOE in 2019 outlines the importance and potential of the ever growing field of Scientific Machine Learning [3]. In this report, they indicate two computational challenges SciML could potentially solve: lack of data in analysis, and solving differential equations. In the first case, in problems where data is either expensive or impossible to collect, standard implementations of machine learning become impossible. This can be overcome if we incorporate physics knowledge of the system into the neural network. Such implementations make use of *Physics-Informed Neural Networks* (PINN) [4]. Additionally, PINN can solve differential equations by slight modification of the lack of data problem [5, 6, 7]. We can even combine these two techniques by including data into solving differential equations to better fit a model or aid in solving the equations [8]. A robust implementation of many of these techniques has been underway in the `Julia` programming language. Packages that make use of machine learning are part of the `SciML.jl` universe which includes a basic syntax for code and basic functions in using SciML. Further information on the evolution and current state of the SciML universe can be found from talks from last year’s SciML Con [9].

Because of the rapidly growing nature of the field of SciML, documentation on how the techniques and new tools incorporating PINN are used is not always easy to access. This paper seeks to fill in some of the gaps, describing succinctly how SciML can be used to solve differential equations. In particular, we put the newly-released package `NeuralPDE.jl` to the test to solve several complex differential equations and compare the computational solutions to known analytical solutions. In section 2, we will outline the basics of machine learning. In sections 3 and 4 we will describe how to solve ODEs and PDEs with PINNs. Finally, in section 5 we experiment with solving Einstein’s field equations to obtain the Schwarzschild Metric using PINN. This problem stretches the limits of PINNs in the application of a complex additional loss term passed to the network during training in order to meet the conditions of the problem. We ultimately find that while we can train a network to approximate the solution of the desired system, more technical development is needed in order to solve such an ambitious problem. For example, in order to have enough physical memory to solve, we would like to use static arrays<sup>1</sup> which are not yet supported in `Julia` implementations of automatic differentiation.

Although our final application is not perfectly successful, it demonstrates the potential this field has in solving ever more complex problems. The implementation in solving Einstein’s field equations makes use of differential programming, something not yet attempted with the `NeuralPDE.jl` package. Future work in this field will likely make solving this and even more complex problems more feasible.

---

<sup>1</sup>Static arrays have a fixed size and are created on the stack. This means the memory they occupy is free after use in a function and the array is subsequently destroyed. Moreover, for relatively small arrays, accessing the memory address of these arrays is much faster than allocations to the heap due to the physical location of memory in the machine.

## 2. Machine Learning

Machine learning and artificial intelligence are prominent features of our everyday life from personalized advertisements to facial recognition software. Machine learning is also a powerful analytical tool in scientific research, in particular with model development and validation. Below, I will describe how neural networks can be trained to make predictions and give some motivation for why neural networks are well suited for function approximation, in particular approximating solutions to differential equations.

### 2.1. Neural Networks

A neural network is simply a function composed primarily of layers of matrix multiplication, but also of non-linear activation functions and bias vectors within the function [10, 11]. As input, a neural network can take a scalar, vector, matrix, or sometimes tensor with a user-specified size or dimension and gives as an output a vector or scalar. Thus, our network  $\mathcal{N}$  takes the form

$$\mathcal{N} : \mathbb{R}^\Theta \rightarrow \mathbb{R}^\ell \quad (2.1)$$

where  $\Theta$  is the user-defined input space, and  $\ell$  is the number of desired output features. Networks have a fixed size defined by the number of *hidden layers* and the number of *nodes* within each hidden layer.

Hidden layers are composed of nodes which cannot be directly accessed in the training process. A node within a layer describes a connection point between layers. Connections are achieved via multiplication by a weight stored in the network. Modification of these weights changes the way in which nodes are connected to each other and thus how inputs are interpreted by the network. At each node, the weight passed to the node is modified by a non-linear function called the *activation function*. This allows us to describe a complex input space more accurately [12]. Some examples of activation functions include the sigmoid defined by

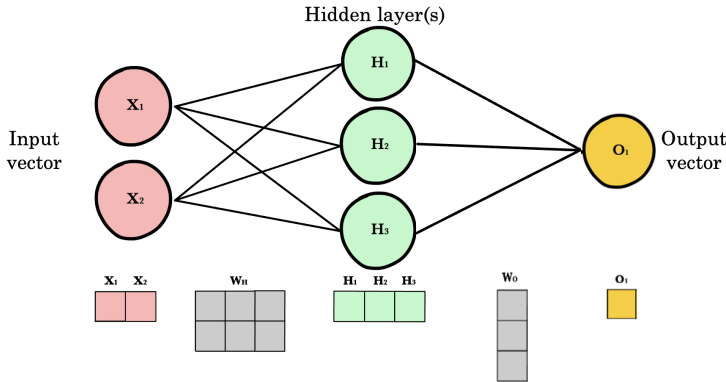
$$\sigma(w) = \frac{1}{1 + e^{-w}}, \quad (2.2)$$

the hyperbolic tangent  $\tanh(w)$ , and LSTM defined by  $f(w) = \max\{0, w\}$  where in each case  $w$  is a vector of weights of the network at a node. Typically, the sigmoid is well suited for classification problems, while LSTM is best used in regression problems due to its simplicity. In all examples below we use a sigmoid activation function.

An example of a simple neural network with one hidden layer is shown in Fig. 1. This network has one input dimension and, as shown in the graphic, we feed one data point  $X$  with two features  $x_1, x_2$  into the network. Parameters of the network are stored in the matrix  $W_0$  such that when we do the multiplication  $XW_0$  we get a vector of three values corresponding to the three nodes of the single hidden layer. The activation function is now applied to these values and optionally a bias vector can be added to these modified weights. We then multiply by the weights  $W_1$  to get our output of a single value which is passed to one final activation function. In most cases, the activation in the final node is the identity function  $f(x) = x$ . Notice that we can have as many inputs, layers, nodes, and output nodes as we would like. The network described above is typically called a *feed-forward* neural network as there are no cycles between nodes, no reference to nodes previously touched, and each node from the previous layer is connected to each node of the subsequent layer. There are more complex neural networks such as recurrent neural networks, however, we will exclusively use feed-forward networks in our work.

### 2.2. Training Neural Networks

Now that we have built some neural network, we would like to have it make some useful predictions. This involves picking a set of special weights through a process called *training*.



**Figure 1.** An example of a simple neural network with two inputs, one hidden layer with three nodes, and a single output. The network is defined by the weights connecting the layers which are represented as matrices. At each node, the weights are passed to a function called the activation function which scales the weights appropriately. In principle, the size of the network can change in every dimension.

In order to train a neural network, we need a way in which to evaluate how well the network is doing given a specific task. This is achieved with a *loss function*. The loss function takes an input of all of the parameters of the network that can change,  $X$ , and outputs a single number, thus, our loss function is of the form

$$\mathcal{L} : \mathbb{R}^{\text{size}(X)} \rightarrow \mathbb{R}. \quad (2.3)$$

Such a function is defined such that the smaller the output value, the better the network performs. Thus, training a neural network becomes an optimization problem where we want to minimize the function  $\mathcal{L}(X)$ . Typically, the space  $X$  is composed of the weights and biases of the network.

For a very basic idea of how this works, imagine the graph of  $\mathcal{L}$  as a surface with some collection of minima. We want to find a minimum, so we need to travel in the direction of one. Intuitively, we can compute the gradient and then travel in the direction of the negative gradient to go “downhill.” More specifically, we travel in the  $-\nabla \mathcal{L}(X)$  direction taking discrete steps of size  $\Delta x$ . After each step we update the network parameters, re-evaluate the loss, compute  $-\nabla \mathcal{L}(X)$  and repeat. We continue in this way for as many steps as the user desires. This process is called gradient descent and the algorithm for finding the gradient is called backpropagation [13]. In most cases, it is not necessary to find the global minimum and may even be computationally disadvantageous as finding global minimum can lead to over-fitting [14]. Since the algorithm can get stuck in a “saddle point” or inadequate local minimum, we often use stochastic methods whose randomness allows the algorithm to escape these pitfalls and continue learning to a better local minimum. This process is called stochastic gradient descent. In the following examples, we will use the ADAM optimizer [15].

### 2.3. The Universal Approximation Theorem

Why should neural networks be able to solve PDEs? The solutions to PDEs are often complicated nonlinear functions, and a neural network is basically just a string of matrix multiplications sandwiched between max or sigmoid functions. At first glance, this is preposterous. The answer comes down to a famous theorem in machine learning: the Universal Approximation Theorem (UAT):

**Theorem 2.1.** *Let  $x \in \Omega \subseteq \mathbb{R}^n$ ,  $u(x)$  be a continuous function, and  $N(x; w)$  be a neural network with weights  $w$ . Then, we can find some arbitrarily small  $\epsilon \in \mathbb{R}$  such that*

$$\|\mathcal{N}(x; w) - u(x)\| < \epsilon. \quad (2.4)$$

Proofs of this theorem are beyond the scope of this report, but are readily available for several different contexts including for classical feed-forward networks [16, 17], finding probability distributions [18], and complex-valued neural networks where  $\Omega \subseteq \mathbb{C}^n$  [19].

There is, however, one big caveat to this theorem: nothing about the problem tells us how big our network should be in order to fulfill the theorem. In some cases, the size of the network may be computationally infeasible to model. Nonetheless, it is this theorem which allows us to be able to have some hope that differential equations can be arbitrarily approximated by neural networks.

In the rest of the paper, we take these theoretical concepts and apply them to the problem of solving differential equations, starting with ODEs and then moving to PDEs.

### 3. Solving ODEs with Physics-Informed Neural Networks

As an illustration of the application of PINN, consider an ordinary differential equation of the form

$$\frac{du}{dt} = f(u, t). \quad (3.1)$$

The UAT tells us that we should be able to achieve  $\mathcal{N}(t; w) \approx u(t)$  where  $\mathcal{N}(t; w)$  is a neural network with weights  $w$ . Consequently, our neural network must satisfy  $\mathcal{N}'(t; w) \approx f(\mathcal{N}(t; w), t)$ . To solve, we discretize the space  $\Omega \subseteq \mathbb{R}$  where  $t \in \Omega$ . Subsequently, we define the loss function by

$$\mathcal{L}(w) = \sum_i \left( \frac{d\mathcal{N}(w; t_i)}{dt} - f(\mathcal{N}(t_i; w), t_i) \right)^2, \quad (3.2)$$

where we are summing over some number of samples, hopefully dense enough to accurately capture the difference between our prediction and its supposed values. In order to find a unique solution, we apply the initial condition  $u(0) = u_0$ . We can simply add this as an additional term to our loss function as

$$\mathcal{L}(w) = (\mathcal{N}(w, 0) - u_0)^2 + \sum_i \left( \frac{d\mathcal{N}(t_i; w)}{dt} - f(\mathcal{N}(t_i; w), t_i) \right)^2. \quad (3.3)$$

Notice that, in order to make solving slightly more efficient, we can encode the initial condition in our neural network solution by defining the function

$$g(t; w) = u_0 - t\mathcal{N}(t; w). \quad (3.4)$$

This ensures that the initial condition is satisfied as  $g(0) = u_0$ , and  $g$  remains a universal approximator. Thus, in this special case, the loss function becomes:

$$\mathcal{L}(w) = \sum_i \left( \frac{dg(t_i; w)}{dt} - f(g(t_i; w), t_i) \right)^2. \quad (3.5)$$

Now that we have defined our problem, we can use standard minimization techniques to find a solution. For a complete example using this technique, see Appendix A. For additional information and further examples see [20].

#### 4. Solving PDEs with Physics-Informed Neural Networks

We now turn our attention to solving partial differential equations using techniques from scientific machine learning. We will use the package `NeuralPDE.jl` [21] to solve some examples.

Consider a partial differential equation given by

$$f(\partial_x^k u(x), \partial_x^{k-1} u(x), \dots, \partial_x u(x), u(x), x; \lambda) = 0, \quad (4.1)$$

where  $\Omega$  is an open subset of  $\mathbb{R}^n$ ,  $x = (x_1, x_2, \dots, x_n) \in \Omega$  are the independent variables,  $u(x)$  is the solution,  $k \geq 1$  is an integer representing the highest order (partial) derivative<sup>2</sup> of  $u(x)$ , and  $\lambda$  are the parameters of the equation. Thus,  $f$  takes the form

$$f : \mathbb{R}^{n^k} \times \mathbb{R}^{n^{k-1}} \times \dots \times \mathbb{R}^n \times \mathbb{R} \times \Omega \rightarrow \mathbb{R}, \quad (4.2)$$

and  $u(x)$  takes the form

$$u : \Omega \rightarrow \mathbb{R} \quad (4.3)$$

as discussed in [1].

To solve with a neural network  $\mathcal{N}(x, w)$  where  $\mathcal{N}$  is a neural network with weights  $w$ , we want to fulfill

$$f(\partial_x^k \mathcal{N}(x; w), \partial_x^{k-1} \mathcal{N}(x; w), \dots, \partial_x \mathcal{N}(x; w), \mathcal{N}(x; w), x; \lambda) = 0. \quad (4.4)$$

For simplicity, we will represent  $f(\partial_x^k \mathcal{N}(x; w), \partial_x^{k-1} \mathcal{N}(x; w), \dots, \partial_x \mathcal{N}(x; w), \mathcal{N}(x; w), x; \lambda)$  as  $f(\mathcal{N}(x; w); \lambda)$  below. The universal approximation theorem tells us that this should be possible. It follows that we may define the error to be

$$\mathcal{L}(w) = \int_{\Omega} \|f(\mathcal{N}(x; w); \lambda)\| dx. \quad (4.5)$$

We have suggestively named the error  $\mathcal{L}$  as we can use this function as a loss function in training our neural network  $\mathcal{N}$  by minimizing  $\mathcal{L}$ . Our computational problem then reduces to the problem of evaluating the integral in Eqn. 4.5 in order to perform minimization.

In practical problems, we apply boundary conditions  $b_i$  on  $\partial\Omega \subset \Omega$  in order to obtain, ideally, a unique solution. In order to include this information in the network, we add these conditions to our loss function

$$\mathcal{L}(w) = \sum_i \int_{\Omega \setminus \partial\Omega} \|f_i(\mathcal{N}(x; w); \lambda)\| dx + \sum_i \int_{\partial\Omega} \|b_i(\mathcal{N}(x; w); \lambda)\| dx. \quad (4.6)$$

where we have generalized the problem to a system of coupled differential equations  $f_i$ . We must note that  $\mathcal{L}(w)$  does *not necessarily* tell us the accuracy of our solution. It is generally true that smaller values of  $\mathcal{L}(w)$  give better solutions; but because  $\mathcal{L}$  is composed of terms from the PDE *and* from the boundary conditions,  $\mathcal{L}(w)$  no longer tells us purely how closely our solution matches the true solution. It rather is a measure of how closely our solution matches the true solution *and* the boundary conditions or additional conditions we can add to the problem.

If we can find a way to efficiently evaluate this integral, then we can apply standard minimization algorithms to solve. This necessitates the use of a discretization in order to numerically integrate.

<sup>2</sup>We use  $\partial_x^k u(x)$  to denote the partial derivative of  $u$  of order  $k$  with respect to the variable(s)  $x$ , where  $u : \Omega \rightarrow \mathbb{R}$ ,  $\Omega$  is an open subset of  $\mathbb{R}^n$ , and  $x \in \Omega$ .

#### 4.1. Methods of discretization

There are several different methods for evaluating the integral in Eqn. 4.6. There are grid approximation techniques, stochastic grid approximation techniques, and machine learning (quadrature) techniques. A simple grid approximation takes the space  $\Omega$  and divides it into units of volume  $\Delta V$  with a specific length. The PDE is evaluated at each of these points and scaled by the volume, thus our integral is computed via

$$\int_{\Omega} ||f(\mathcal{N}(x; w); \lambda)|| dx = \sum_i \Delta V ||f(\mathcal{N}(x_i; w); \lambda)||. \quad (4.7)$$

This method has two main disadvantages. First, as the dimension  $n$  of  $\mathbb{R}^n$  increases, the number of sample points required in order to maintain the same accuracy for lower  $n$  increases exponentially. This is the so-called “curse of dimensionality” as an exponentially increasing number of sampling points quickly renders a problem computationally impossible or very slow to solve. Second, no evaluation of the integral is done between grid points, thus information is quickly lost. To solve the second problem, we can use stochastic methods of sampling—i.e., Monte Carlo techniques to evaluate the integral. To do this, we take a random sample  $(x_1, x_2, \dots, x_n) \in \Omega$  and then evaluate the integral by

$$\int_{\Omega} ||f(\mathcal{N}(x; w); \lambda)|| dx = \alpha \sum_i ||f(\mathcal{N}(x_i; w); \lambda)||. \quad (4.8)$$

where  $\alpha = V/N$ ,  $N$  is the number of samples  $x_i \in \Omega$ , and  $V$  is the volume element of  $\Omega$  defined by

$$V = \int_{\Omega} dx. \quad (4.9)$$

As we simply want to minimize this integral, there is no need to calculate  $\alpha$ . This problem still suffers from the curse of dimensionality as the number of points required to accurately calculate this integral  $N$  remain fixed. Thus, we turn to a third method: quadrature training with a neural network. Several processes for specifying quadrature are described in [22]. These are typically of the form

$$\int_{\Omega} ||f(\mathcal{N}(x; w); \lambda)|| dx = \sum_i \alpha_i ||f(\mathcal{N}(x_i; w); \lambda)||. \quad (4.10)$$

In the implementation via `NeuralPDE.jl`, this is accomplished using the `Integrals.jl` package which calls the `Julia` differential equation solver [23] to find the correct sample points  $x_i$  and weights  $\alpha_i$  in an implementation of Gaussian quadrature rules. This method is different from those presented above, as the number of points  $N$  are not fixed prior to solving, rather are determined by an error estimate by the algorithm in `Integrals.jl`. Ideally, this algorithm picks many sample points where there are interesting features of the solution and fewer points where there are not as interesting features. Because `NeuralPDE` is still in its infancy, quadrature techniques are often not as accurate as Monte Carlo (called `QuasiRandomTraining` in the `NeuralPDE` implementation) techniques. In particular, with the addition of complex additional-loss terms, the quadrature algorithm is not able to identify which areas in space are more important than others. Moreover, in high-dimensional integrals, [22] find that often Monte-Carlo techniques converge faster and are more accurate than quadrature techniques. Thus, in the examples and in the following implementation, we use both Quadrature methods and Quasi-Random methods and pick which yields the best performance.

For a simple implementation of the techniques discussed in this section using the `NeuralPDE.jl` architecture, see Appendix B and for a more in-depth example see Appendix C.

## 5. Solving Einstein's Field Equations to obtain the Schwarzschild Metric using `NeuralPDE.jl`

We find a numerical solution to Einstein's field equations resulting in the Schwarzschild Metric as first described by Schwarzschild in 1916 [24] by using PINN techniques and mathematical simplifications. This problem has the benefit that it both tests the limits of what neural networks can do while still having a nice analytical solution to which we can compare our result. We also use this example to experiment with using differentiable programming to implement a boundary condition, that the field equations should reproduce Newtonian gravity in certain limiting cases.

### 5.1. Physics Background and Problem Set-up

Solutions to the Einstein field equations seek to describe space-time, mathematically encapsulated in the space-time metric tensor  $g_{\mu\nu}$  and the stress-energy tensor  $T_{\mu\nu}$ . We will consider the problem of a massive black hole which is 1) uncharged and 2) non-rotating following the detailed solution given in [25]. To find  $g_{\mu\nu}$ , consider Einstein's field equations given by

$$R_{\mu\nu} - \frac{1}{2}Rg_{\mu\nu} + \Lambda g_{\mu\nu} = \frac{8\pi G}{c^4}T_{\mu\nu}. \quad (5.1)$$

We will consider space-time outside the black hole which we assume to be empty. Thus,  $T_{\mu\nu} = 0$ . We additionally will not consider cosmological scales so we set  $\Lambda = 0$ . Using some algebra, we then find that

$$\begin{aligned} R_{\mu\nu} - \frac{1}{2}Rg_{\mu\nu} &= 0 \\ R_{\mu\nu}g^{\mu\nu} - \frac{1}{2}Rg_{\mu\nu}g^{\mu\nu} &= 0 \\ R - 2R &= 0 \\ R_{\mu\nu} = 0 &\implies \boxed{R_{\mu\nu} = 0}. \end{aligned} \quad (5.2)$$

We define the Cristoffel symbols as

$$\Gamma_{\mu\nu}^{\alpha} := -\frac{1}{2} \sum_{\beta} g^{\alpha\beta} \left( \frac{\partial g_{\mu\beta}}{\partial x_{\nu}} + \frac{\partial g_{\nu\beta}}{\partial x_{\mu}} - \frac{\partial g_{\mu\nu}}{\partial x_{\beta}} \right) \quad (5.3)$$

and subsequently define the Ricci tensor as:

$$R_{\mu\nu} := \sum_{\alpha} \frac{\partial \Gamma_{\mu\nu}^{\alpha}}{\partial x_{\alpha}} - \sum_{\alpha} \frac{\partial \Gamma_{\alpha\mu}^{\nu}}{\partial x_{\nu}} + \sum_{\alpha\beta} \left( \Gamma_{\alpha\beta}^{\nu} \Gamma_{\mu\nu}^{\alpha} - \Gamma_{\mu\beta}^{\alpha} \Gamma_{\alpha\nu}^{\nu} \right) \quad (5.4)$$

Finally, define the metric tensor as

$$g_{\mu\nu} = \begin{pmatrix} g_{00} & g_{01} & g_{02} & g_{03} \\ g_{10} & g_{11} & g_{12} & g_{13} \\ g_{20} & g_{21} & g_{22} & g_{23} \\ g_{30} & g_{31} & g_{32} & g_{33} \end{pmatrix}, \quad (5.5)$$

where we use the space-time indices  $(0, 1, 2, 3)$  corresponding to the variables  $(\tau = ct, \rho, \theta, \phi)$  in spherical coordinates.

In order to solve we need to make a few further assumptions:

1. As  $r \rightarrow \infty$ ,  $g_{\mu\nu} \rightarrow \eta_{\mu\nu}$  where  $\eta_{\mu\nu}$  is the minkowski metric.



2. Space-time is static meaning that a)  $\partial_t g_{\mu\nu} = 0$  and b)  $t \rightarrow -t$  leaves  $g_{\mu\nu}$  unchanged. This ensures the black hole is non-rotating.
3. Space-time is spherically symmetric.

Applying these assumptions to the metric causes almost all components to vanish, and reduces the unknowns to  $A(r)$  and  $B(r)$ , two functions of a single variable  $r$ . The metric tensor now reads

$$g_{\mu\nu} = \begin{pmatrix} A(r) & 0 & 0 & 0 \\ 0 & -B(r) & 0 & 0 \\ 0 & 0 & r^2 & 0 \\ 0 & 0 & 0 & r^2 \sin^2 \theta \end{pmatrix}. \quad (5.6)$$

Finally, in order to find a unique solution, we need to ensure the metric matches Newtonian gravity in the limits of 1) weak gravity and 2) low velocity. This essentially means we need the Geodesic Equation, given by

$$\frac{d^2 x^\sigma}{d\lambda^2} + \Gamma_{\mu\nu}^\sigma \frac{dx^\mu}{d\lambda} \frac{dx^\nu}{d\lambda} = 0, \quad (5.7)$$

to match Newtonian gravity, given by

$$\frac{d^2 x^i}{dt^2} + \frac{\partial \varphi}{\partial x^i} = 0 \quad (5.8)$$

where  $\varphi = -GM/r$  is the gravitational potential.

1. In the limit of low velocity,  $t \approx \tau = ct$ , and a particle's four-velocity is dominated by the time component so that a particles four-velocity is approximated by  $\mathbf{U} = (c, 0, 0, 0)$ . This reduces Eqn. 5.7 to

$$\frac{d^2 x^i}{dt^2} + \Gamma_{00}^i c^2 = 0 \quad (5.9)$$

which implies that

$$\Gamma_{00}^i = \frac{\partial \varphi}{\partial x^i} \frac{1}{c^2}. \quad (5.10)$$

2. In the limit of weak gravity we assume  $g_{\mu\nu} \approx \eta_{\mu\nu} + h_{\mu\nu}$  where  $||h_{\mu\nu}|| \ll 1$ . Then, we can write

$$\Gamma_{00}^i = \frac{1}{2} \eta^{ii} (\partial_0 h_{i0} + \partial_0 h_{i0} - \partial_i h_{00}) \quad (5.11)$$

as derivatives of the constant matrix  $\eta$  in Cartesian coordinates are zero. Because all off-diagonal elements of our matrix are zero, this reduces to

$$\Gamma_{00}^i = -\frac{1}{2} \eta^{ii} \partial_i h_{00} \quad (5.12)$$

where we note that  $\eta^{ii} = -1$  in Cartesian coordinates.

Applying all the above conditions, we find that the metric is given by

$$g_{\mu\nu} = \begin{pmatrix} 1 - \frac{r_s}{r} & 0 & 0 & 0 \\ 0 & -(1 - \frac{r_s}{r})^{-1} & 0 & 0 \\ 0 & 0 & r^2 & 0 \\ 0 & 0 & 0 & r^2 \sin^2 \theta \end{pmatrix}, \quad (5.13)$$

where

$$r_s = \frac{2GM}{c^2} \quad (5.14)$$

is the Schwarzschild radius or the event horizon of the black hole. This is the analytical solution we will try to reproduce numerically in the next section.

## 5.2. Technical Implementation

While this system has a nice, simple analytical solution, finding a numerical solution is not trivial. In particular, starting with the system of PDEs given by

$$R_{\mu\nu} = 0 \quad (5.15)$$

where the form of  $R_{\mu\nu}$  is given in 5.4, and then applying the assumptions outlined above as initial conditions yields a PDE problem too big to feasibly solve given the time constraints of this capstone. For this reason, we will use the simplified form of  $g_{\mu\nu}$  given in Eqn. 5.6 for our implementation. While this yields a system of highly coupled ODEs, employing initial conditions such that a solution is reached is still not trivial. The implementation provided also demonstrates how the problem may be expanded to more dimensions at the cost of significantly longer training times as we use `NeuralPDE.jl` to solve. Additionally, architecture for expanding to more complex systems is included in the implementation for potential future use.

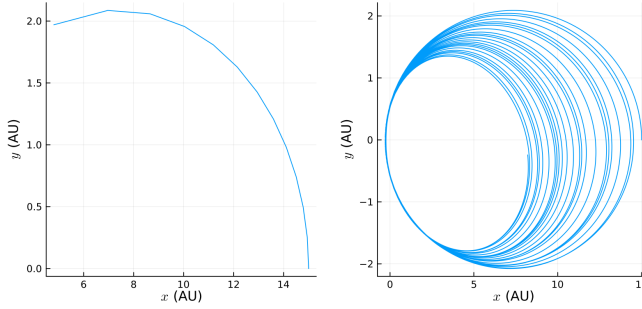
The system of equations we ultimately solve can be found by plugging the simplified form for  $g_{\mu\nu}$  into our system of equations, Eqn. 5.15. This results in the system given by

$$\begin{aligned} 2rABA'' - rAA'B' + 4ABA' - rBA'^2 &= 0 \\ -2rABA'' + rAA'B' + 4A^2B' + rBA'^2 &= 0 \\ -2AB + 2AB^2 - rA'B + rAB' &= 0 \end{aligned} \quad (5.16)$$

where the prime indicates a derivative of  $r$  such that  $A' = dA(r)/dr$ . In order to fulfill the condition  $r \rightarrow \infty$ ,  $g_{\mu\nu} \rightarrow \eta_{\mu\nu}$ , we impose the boundary conditions  $A(r_{max}) = 1$ ,  $B(r_{max}) = 1$ . We solve this problem over the interval  $r = [r_{min}, r_{max}]$  where we choose  $r_{min}$  such that  $r_{min} > r_s$  to avoid the singularity at the black-hole's event horizon, and  $r_{max}$  is big enough such the boundary condition is a reasonable approximation when applied to the analytic solution. Recall that the analytical solution is given by

$$\begin{aligned} A(r) &= 1 - \frac{r_s}{r} \\ B(r) &= -\left(1 - \frac{r_s}{r}\right)^{-1} \end{aligned} \quad (5.17)$$

where  $r_s$  is the Schwarzschild radius.



**Figure 2.** ODE solution using Newtonian prediction for particle motion. Initial conditions are (15AU, 0AU) with velocity (0AU/yr, 100AU/yr). On the left panel, the problem is solved over the time interval  $t = (0, 0.03)$  with a step-size of 0.0025 while the right panel shows a more complete solution of the interval  $t = (0.0, 1.0)$ .

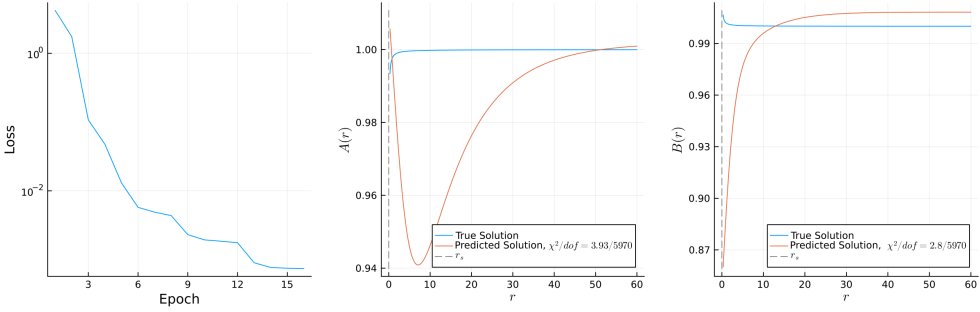
For this problem, we will consider space-time around a black hole with a mass  $10^5$  times that of the sun and will use astronomical units for length and years for time. Thus,  $GM = 3.94\text{AU}^3/\text{year}^2$  and  $c = 6.3 \times 10^4\text{AU}/\text{year}$  so that  $r_s = 1.2 \times 10^{-4}\text{AU}$ .

We still need to find a way to match Newtonian gravity in the limit of low velocity and weak gravity. In order to do this, we put a test particle into the space-time predicted by our network and let this particle move according to the geodesic equation, Eqn. 5.9. We then drop this same particle into a Newtonian space-time and let it move according to Newton's laws, Eqn. 5.8. We can then compare both solution paths and add a measure of their difference to the loss function for training a better solution. To measure the difference between the two paths we sample both paths at  $\sim 15$  points, and square the Euclidean distances between corresponding points, where the Euclidean distance between two points  $p_1 = (x_1, y_1, z_1), p_2(x_2, y_2, z_2) \in \mathbb{R}^3$  is defined as

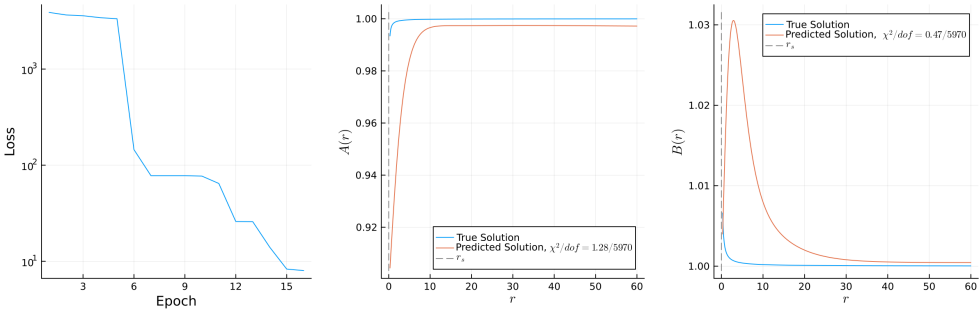
$$d(p_1, p_2) := \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}. \quad (5.18)$$

Unfortunately, implementing this is not a trivial task. Notice that in order for the network to learn or perform backpropagation, the differential equation given in Eqn. 5.9 must be solved *inside the loss function*. Thus, in order to move towards a better solution, we must be able to take the derivative of an ODE solver: a clear application for the recent paradigm called differentiable programming [26, 27]. Moreover, solutions to the simple Newtonian version can quickly blow-up or require advanced solving techniques for slight changes in initial parameters of the problem. This makes solving within the loss function potentially very unstable.

In order to partially overcome an unstable loss, we pick a solution to the Newtonian problem which slowly moves away from a stable orbit. In addition, we solve only in the  $xy$ -plane so as to eliminate as much computational pressure as possible. By placing our particle on the  $x$ -axis with velocity completely in the  $y$ -direction, namely we place it at (15AU, 0AU) with velocity (0AU/yr, 100AU/yr), we accomplish these conditions. Solving over the time interval  $t = (0, 0.03)$  with `DifferentialEquations.jl` yields the solution shown in Fig. 2. This is, of course, only a part of the trajectory of the particle. In order to show what a complete solution would look like, we plot over the time interval  $t = (0.0, 1.0)$ . To see our solution implementation, refer to the data-availability statement at the end of this paper.



**Figure 3.** Solution of simplified ODEs in Eqn. 5.16. Solutions are not forced to match Newtonian gravity. The left panel shows the loss over the training internal, while the right two panels show the solutions  $A(r)$  and  $B(r)$ .

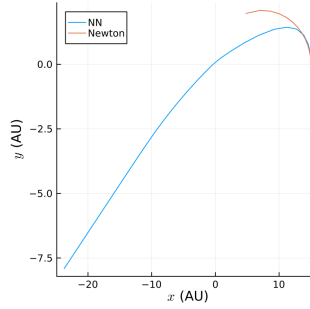


**Figure 4.** Solution of simplified ODEs in Eqn. 5.16. Solutions is forced to match Newtonian gravity. The left panel shows the loss over the training internal, while the right two panels show the solutions  $A(r)$  and  $B(r)$ .

### 5.3. Results and Discussion

Now that we have the system setup, we can attempt to solve for the remaining component of the metric tensor  $g_{\mu,\nu}$ , namely for the functions  $A(r)$  and  $B(r)$  as described in the system of ODEs we derived above in Eqn. 5.16. We use a network with two hidden layers both with the sigmoid activation function. Each hidden layer contains ten nodes each. Weights in the network are initialized uniformly according to the Glorot paper [28]. Training is done with the BFGS and limited memory BFGS algorithm [29] over fifteen epochs. One solution obtained through these methods is shown in Fig. 4 while Fig. 3 shows a solution where matching Newtonian gravity was *not* employed. Notice that quantitatively there is no real difference between these solutions, as measured by the loss function. However, qualitatively, after applying the condition to match Newtonian gravity, our solution better matches the shape of the true solution. In particular, the solution with Newtonian gravity better matches the shape of  $A(r)$  and largely matches the shape of  $B(r)$ . The evident peak in  $B(r)$  in Fig. 4 can be explained by the proximity of the solution to the mathematical singularity at  $r_s$ . Because the true solution diverges to  $+\infty$  and  $-\infty$  at  $r_s$ , our algorithm could be tricked by the asymptote to  $-\infty$ , giving our solution for  $B(r)$  its sharply peaked shape.

One of the greatest challenges in solving this problem is managing physical memory of the system. When adding an additional loss term to the solving scheme, each operation added by the user takes a certain amount of memory. In particular, when performing back-propagation over a differential equation solver in addition to attempting to solve a large system of ODEs results in huge memory allocation. If too much memory is required in order to solve, the system crashes and a segmentation fault is reported.



**Figure 5.** Illustration of how well our solution matches Newtonian gravity after training. The label *Newton* refers to the ODE solution obtained before training which we are trying to match while *NN* refers to the neural network solution after training.

In order to try to solve this problem, we do all calculations with 32-bit floating point numbers rather than the standard 64-bit. Additionally, we cannot ask the ODE solver in the additional loss function to solve for many points: around 15 points seems to be the limit. In addition, all attempts to automatically interpolate by `DifferentialEquations.jl` must be turned off so as not to waste memory.

Notice that trying to match Newtonian gravity with only  $\sim 15$  points may not work super well, but memory constraints force this upon us. Additionally, our choice of the Newtonian solution with which to compare highly influences how we match. Because we are limited in the number of points we can choose, we cannot pick an orbit which goes completely around the black hole as this would be too discontinuous to model with so few points. Thus, after training, the typical comparison with Newtonian gravity is not such an ideal match as shown in Fig. 5. This could be solved by adding more points to the solutions of additional loss; however, as mentioned previously, this results in a segmentation fault.

Another problem that could be addressed in the future is the fact that we are using only one example solution to match the network solution to Newtonian gravity. Randomly picking initial conditions for solving the ODE problem may help in solving this problem but would require putting not one but *two* ODE solvers in the additional loss function: one for solving the Newtonian problem, and one for solving the Geodesic equation including the network solution. A more achievable future improvement would be to previously solve many examples of the Newtonian ODE for various initial conditions and try to match all of these within the additional loss function. Again, this would very quickly run into memory problems as this would necessitate solving more than one ODE per epoch within this additional loss function. In order to avoid this, every epoch could focus on trying to match a different set of ODE initial conditions, however this may not work as this would mean an ever-changing problem which a network may not be able to predict.

There are several potential ways to solve this memory problem including using `StaticArrays.jl` to store the vectors of points from the ODE solver. Unfortunately, the architecture in `Julia` for automatic differentiation—the method used in differential programming to compute derivatives—does not support statically typed arrays. Additionally, running on the GPU would both speed up computation as well as manage memory better. However, the GPU architecture in `Julia` for automatic differentiation in this context has yet to be implemented. Both of these limitations exhibit how new this research is. The fact that the above implementation still works despite these memory limitations is a clear indication of how important it is to continue developing tools in SciML.

## 6. Conclusion

Physics-informed neural networks are a versatile, efficient, and accurate method in solving systems of partial differential equations. In particular, the newly released package `NeuralPDE.jl`, provides a

symbolic interface to easily implement many different PDE structures. These include but are not limited to integral-PDEs, systems of PDEs, and non-homogeneous, non-linear PDEs. Moreover, this method has the ability to include extra information in solving PDEs via addition of yet another term to the loss function. Moreover, very complex objects such as an ODE solver can be put directly into the loss function in order to implement a boundary condition, or include extra information in solving. This lets us not only solve inverse problems with ease, but also include complex structures into our PDE models.

Our implementation in solving Einstein's field equations to obtain the Schwarzschild metric shows how addition of a complex additional loss term can work at the cost of huge memory use, often resulting in segmentation fault. This error may be fixed in future versions of `NeuralPDE.jl` or with the continued development of the SciML ecosystem in `Julia`. Because this package is so new, differential programming functionality has not been extensively studied in many of the dependencies of `NeuralPDE`. To our knowledge, this is the only attempt at using PINN to solve a system of PDEs which requires use of differential programming. As such, while not perfectly successful, this implementation is a step in understanding how tools such as `NeuralPDE.jl` and PINN can find wide applications in solving PDEs.

### A. Example ODE problem solved with a PINN

Consider the equation

$$\frac{du}{dt} = e^t \cos t \quad (\text{A.1})$$

with the initial condition  $u(0) = 0.1$ . Notice that we can find the solution to this equation via integration, i.e., the solution is given by

$$u(t) = \int e^t \cos t \, dt. \quad (\text{A.2})$$

Applying integration by parts twice, we find the solution

$$u(t) = \frac{1}{2}e^t(\sin t + \cos t) + C \quad (\text{A.3})$$

where  $C = -0.4$  via use of the initial condition.

Let's now solve using the techniques defined above using `Julia` and the machine learning package `Flux.jl`. We begin by defining a neural network with two hidden layers one of 64 nodes the other with 32 nodes using the tanh activation function and a linear output:

```
using Flux
NNODE = Chain(x -> [x], # Take in a scalar and transform it into an array
              Dense(1, 64, tanh),
              Dense(64, 32, tanh),
              Dense(32, 1),
              first) # Take first value, i.e. return a scalar
```

We then parametrize the solution and define the loss function in which the differentiation is computed via discretization:

```
g(t) = t*NNODE(t) + 0.1

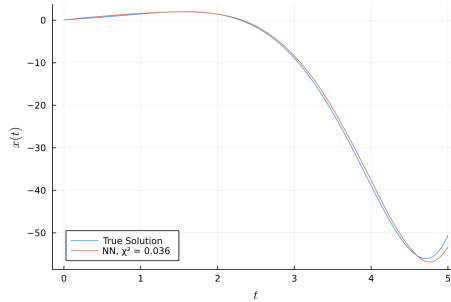
using Statistics
ϵ = sqrt(eps(Float32))
loss() = mean(abs2((g(t+ϵ)-g(t))/ϵ) - (exp(t) * cos(t))) for t in 0:1f-2:5f0)
```

Notice that we use machine  $\epsilon$  here to discretize our integral as this is the smallest possible discretization before floating point error disrupts the solution. We will use gradient descent to solve with a learning rate of  $10^{-4}$  over  $10^4$  epochs. Because there is no data we are providing to train on, we feed the network empty data. Instead, all of the learning information comes from the loss function defined above. This is accomplished with the following code:

```
epochs = 10_000
learning_rate = 1e-4

opt = Flux.Descent(learning_rate) # use gradient Descent
data = Iterators.repeated((), epochs) # no data here : repeat empty data
iter = 0
loss_history = []
cb = function () #callback function to observe training
    global iter += 1
    push!(loss_history, loss())
    if iter % 500 == 0
        display(loss())
    end
end
display(loss())
Flux.train!(loss, Flux.params(NNODE), data, opt; cb=cb)
```

We can then see how our network does in comparison to the true solution as shown in the below plot:



and we see that our network does pretty well! We could, of course, improve performance by training longer or playing with the size of the network or learning rate, however the above remains a good example of how to utilize neural networks to solve ODEs as is.

## B. Example using NeuralPDE.jl

We show how NeuralPDE.jl can be used to solve a simple example and describe the important components of the code.

Consider the homogeneous system of linear PDEs given by

$$\begin{aligned}\partial_t u - \partial_x v + (v + u) &= 0 \\ \partial_t v - \partial_t u + (v + u) &= 0\end{aligned}\tag{B.1}$$

subject to the initial conditions

$$u(0, x) = \sinh x, \quad v(0, x) = \cosh x.\tag{B.2}$$

By means of the so-called Homotopy analysis method, the authors in [30] find the analytical solution to this system to be given by

$$u(x, t) = \sinh(x - t), \quad v(x, t) = \cosh(x - t). \quad (\text{B.3})$$

We now solve with `NeuralPDE.jl`, describing important aspects of the code. To begin, we set up the system symbolically with the aid of `ModelingToolkit.jl` as follows:

```
@parameters t x
@variables u(..) v(..)

Dx = Differential(x)
Dt = Differential(t)

eqns = [Dt(u(t,x)) - Dx(v(t,x)) + (u(t,x) + v(t,x)) ~ 0,
        Dt(v(t,x)) - Dx(u(t,x)) + (u(t,x) + v(t,x)) ~ 0]

bcs = [u(0,x) ~ sinh(x), v(0,x) ~ cosh(x)]

domains = [x ∈ Interval(0.0, 1.0),
           t ∈ Interval(0.0, 1.0)]

@named pde_sys = PDESystem(eqns, bcs, domains, [t,x], [u(t,x), v(t,x)])
```

For systems of equations, store each equation as an element of the vector `eqn` as with the boundary conditions, `bcs`. We now define a neural network to solve the problem using the machine learning library `Lux.jl`. Our network has one hidden layer with 30 nodes, and takes two input variables—one for each parameter of the problem. We output a single number. Notice that each variable  $u, v$  of our problem gets its own network. This is implemented simply as

```
dim = length(domains) # number of dimensions
n = 30
chains = [Lux.Chain(
    Dense(dim, n, Lux.σ),
    Dense(n, n, Lux.σ),
    Dense(n, 1)) for _ in 1:2]
```

We then discretize the system using quadrature training as follows

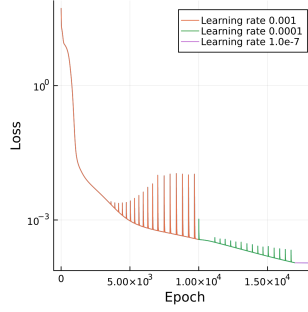
```
strategy = QuadratureTraining()
discretization = PhysicsInformedNN(chains, strategy)
@time prob = discretize(pde_sys, discretization)
```

After this is set-up and a loss function is defined through calling `PhysicsInformedNN`, we simply need to train the network to find a nice solution. This is done with the `Optimization.jl` library. We choose the ADAM optimizer over three learning rates:  $10^{-3}$ ,  $10^{-4}$ , and  $10^{-7}$  over 10000, 7000, and 1000 epochs respectively. By training the network with successively smaller learning rates, we can improve the accuracy of the solution. This is accomplished in the following code:

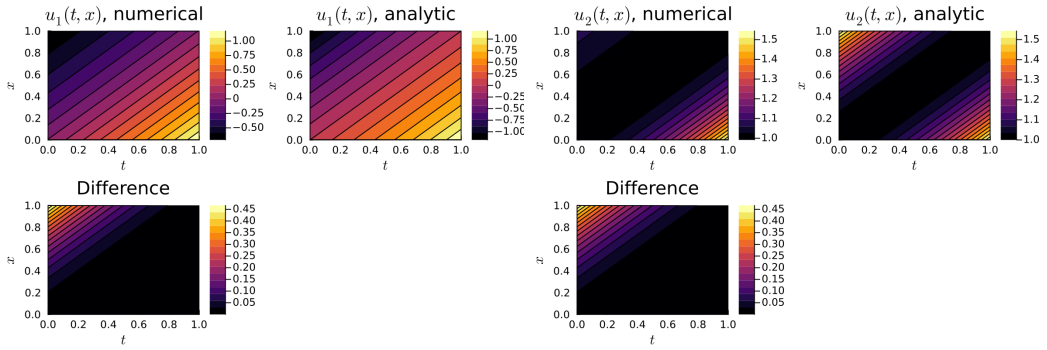
```
i = 0
loss_history = []

callback = function (p,l)
    global i += 1
    if i % 100 == 0
        println("Current loss is: $l")
    end
    append!(loss_history, l)
    return false
end
```





**Figure 6.** Value of loss as a function of epoch for the linear system of homogenous PDEs. Three stages of learning are represented with the learning rate labeled. While there are some instabilities in the loss, we ultimately find a nice solution with the final learning rate of  $10^{-7}$ .



**Figure 7.** Results of training a neural network to learn the solution of the linear homogenous PDE. Solution  $u$  is represented by  $u_1$  and  $v$  by  $u_2$ . In both cases, numerical and analytic solutions are shown as well as the difference between numerical and analytic defined by  $|u_{\text{numerical}} - u_{\text{analytic}}|$ .

```
learning_rates = [1e-3, 1e-4, 1e-7]
res = @time Optimization.solve(prob, ADAM(learning_rates[1])); callback =
    callback, maxiters=10000)
loss_history1 = loss_history
loss_history = []
prob = remake(prob, u0=res.minimizer)
res = @time Optimization.solve(prob, ADAM(learning_rates[2])); callback =
    callback, maxiters=7000)
loss_history2 = loss_history
loss_history = []
prob = remake(prob, u0=res.minimizer)
res = @time Optimization.solve(prob, ADAM(learning_rates[3])); callback =
    callback, maxiters=1000)
loss_history3 = loss_history
loss_history = vcat(loss_history1, loss_history2, loss_history3)
phi = discretization.phi
```

The loss is displayed in Fig. 6 and the results of the above learning are shown in Fig. 7. Notice the instability of the loss at higher learning rates. Fixing these instabilities is accomplished with a lower learning rate as this requires that over each epoch movement in loss-space is limited. The results demonstrate that our learning was successful.

### C. Solving Similarity Solution of Navier-Stokes using `NeuralPDE.jl`

In the 1994 paper investigating the Blow-up of solutions to Navier-Stokes equations, Budd *et al.* consider the system of PDEs arising from the Navier-Stokes equations under the geometry of an infinite channel [31]. In particular, they explore the parabolic partial differential equations perturbed from a similarity solution of the Navier-Stokes given by

$$\begin{aligned}\partial_t u_1 + \mu u_2 \partial_x u_1 &= \partial_x^2 u_1 + u_1^2 - K^2, & 0 < x < 1, 0 < t < T \\ \partial_x u_2 &= u, & 0 < x < 1, 0 < t < T\end{aligned}\tag{C.1}$$

with the boundary conditions

$$\partial_x u_1(t, 0) = \partial_x u_1(t, 1) = u_2(t, 0) = u_2(t, 1) = 0\tag{C.2}$$

and initial condition

$$u_1(x, 0) = u_0(x).\tag{C.3}$$

These equations are parameterized by  $\mu$ , which perturbs the equations from Navier-Stokes. It is the goal of the paper to determine for which values of  $\mu$  the solutions “blow up.” We will consider exclusively the case where  $\mu = 1$ , as this has a nice solution and the resulting system arises from the Navier Stokes equations.

#### C.1. Problem Derivation

We consider the incompressible Navier-Stokes equations in an infinitely long channel in the  $x$ - and  $z$ -direction and finite in the  $y$  direction; namely we bound  $y$  by  $0 < y < 1$ . Let  $u_1, u_2, u_3$  be velocity fields,  $1/\nu$  be the Reynolds number, and  $p$  be the pressure. Then, the Navier-Stokes equations read

$$\begin{aligned}\partial_t u_1 - u_1^2 + u_2 \partial_y u_1 &= \nu \partial_y^2 u_1 - c(t) \\ \partial_t u_2 + u_2 \partial_y u_2 &= \nu \partial_y^2 u_2 - \partial_y p \\ \partial_t u_3 - u_3^2 + u_2 \partial_y u_3 &= \nu \partial_y^2 u_3 - e(t) \\ \partial_y u_2 &= u_1 + u_3.\end{aligned}\tag{C.4}$$

In order to simplify, let  $u_3(y, t) = 0$ ,  $e(t) = 0$  and  $\nu = 1$  so that our system becomes

$$\begin{aligned}\partial_t u_1 - u_1^2 + u_2 \partial_y u_1 &= \partial_y^2 u_1 - c(t) \\ \partial_t u_2 + u_2 \partial_y u_2 &= \partial_y^2 u_2 - \partial_y p \\ \partial_y u_2 &= u_1.\end{aligned}\tag{C.5}$$

Now note that  $p(t)$  can be determined once we know  $u_1, u_2$  from the above first and last equation as the middle equation of Eqn. C.5 is only dependent on  $u_2$ . Hence, we write our system as

$$\begin{aligned}\partial_t u_1 + u_2 \partial_y u_1 &= \partial_y^2 u_1 + u_1^2 - c(t) \\ \partial_y u_2 &= u_1.\end{aligned}\tag{C.6}$$

which matches Eqn. C.1 when  $\mu = 1$  and under the condition that  $y \rightarrow x$ . In order to have a complete solution, we need only finding appropriate initial conditions. The condition  $u_2(0, t) = u_2(1, t)$  follows naturally to disallow flow through the walls of the channel. In [31], they also employ the boundary condition  $\partial_y u_1 = 0$  to obtain the above complete initial conditions. Finally, we require  $c(t) \geq 0$  so that we can write  $c(t) = K^2$ .

### C.2. Solution with PINN and comparison to analytical solution

Let's now solve this system! Notice that via integration by parts of the first equation in C.1 we get

$$K^2 = 2 \int_0^1 u_1^2 dx. \quad (\text{C.7})$$

Thus, we solve the system

$$\begin{aligned} \partial_t u_1 &= \partial_x^2 u_1 - u_2 \partial_x u_1 + u_1^2 - 2 \int_0^1 u_1^2 dx \\ 0 &= \partial_x u_2 - u_1 \end{aligned} \quad (\text{C.8})$$

for  $0 < x < 1$  and  $0 < t < 1$ . Using the same initial conditions as above, we set  $u_0(x) = \cos \pi x$ , coinciding with the analytical solution given in [32]. Using power-series methods, the authors find the analytical solution to the above system to be

$$\begin{aligned} u_1(t, x) &= e^{-\pi^2 t} \cos \pi x \\ u_2(t, x) &= \frac{1}{\pi} e^{-\pi^2 t} \sin \pi x. \end{aligned} \quad (\text{C.9})$$

We can then solve the problem using `NeuralPDE.jl`. This differs from the above code only in the definition of an integral through `ModelingToolkit.jl` which can simply be done with

```
Ix = Integral(x in DomainSets.ClosedInterval(0, 1))
```

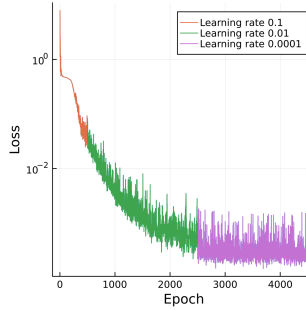
In this example we use the sigmoid as our activation function and build a network with one hidden layer of fifteen nodes. The input layer of our network must be the length of the number of parameters defined in our problem. Additionally, we use a different neural network for each solution  $u_1$  and  $u_2$  of our system, thus we define two neural networks with the set-up described above. The discretization we choose is the quasi random training with 100 points.

For optimization we use the ADAM optimizer and vary the learning rate starting with a rate of 0.1 over 500 epochs, then 0.01 over 2000 epochs, then finally  $10^{-4}$  over a final 2000 epochs. The loss over these epochs and learning rates can be seen in Fig. 8 and the solution can be seen in Fig. 9.

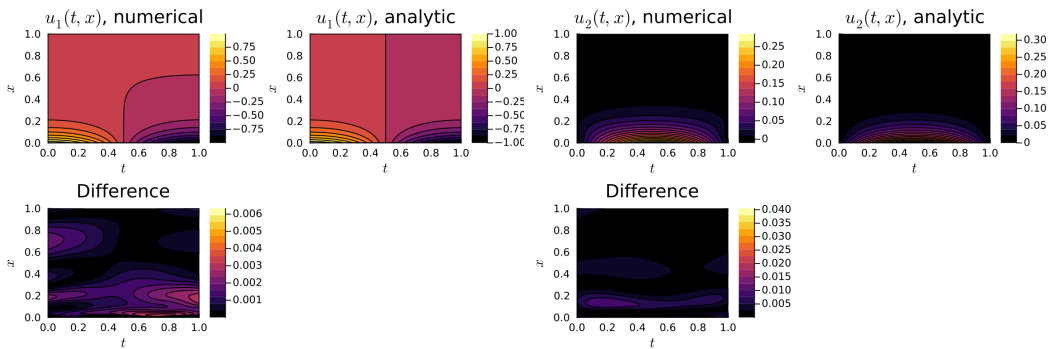
To see the complete implementation of the above, see the data availability statement.

**Acknowledgments.** I am very grateful for the mentorship of Alex Heaton and Megan Pickett in working on this project. Additionally, none of this would have been possible without the education I have received at Lawrence University. I would like to thank the Physics and Mathematics department for their support and all of the opportunities I have had through both departments.

**Data Availability Statement.** All code presented above can be found in my Github, [github.com/lvb5,9](https://github.com/lvb5,9) in the repository `solve_PDEs_with_PINN`. Find examples in the `src` folder. All code for solving the Einstein field equations is in `src/solve_einstein`.



**Figure 8.** Loss as a function of epoch in training a neural network to learn the solution of the integral PDE presented above. Learning is broken up into three sections with differing learning rate as shown.



**Figure 9.** Results of training a network to learn the solution of the integral PDE. Both numerical and analytical solutions are given with the difference between the two. We see that the numerical solution matches the analytic solution very well..

## References

- [1] L. C. Evans, *Partial Differential Equations*. No. v. 19 in Graduate Studies in Mathematics, Providence, R.I: American Mathematical Society, 2nd ed ed., 2010.
- [2] David Cook, *Strengthening Computation in Undergraduate Physics Programs (CPSUP)*. Lawrence University Physics Department, 2007.
- [3] N. Baker, F. Alexander, T. Bremer, A. Hagberg, Y. Kevrekidis, H. Najm, M. Parashar, A. Patra, J. Sethian, S. Wild, K. Willcox, and S. Lee, “Workshop Report on Basic Research Needs for Scientific Machine Learning: Core Technologies for Artificial Intelligence,” Tech. Rep. 1478744, US Department of Energy, Feb. 2019.
- [4] G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, and L. Yang, “Physics-informed machine learning,” *Nature Reviews Physics*, vol. 3, pp. 422–440, June 2021.
- [5] I. E. Lagaris, A. Likas, and D. I. Fotiadis, “Artificial Neural Networks for Solving Ordinary and Partial Differential Equations,” tech. rep., University of Ioannina, May 1997.
- [6] J. Han, A. Jentzen, and W. E, “Solving high-dimensional partial differential equations using deep learning,” *Proceedings of the National Academy of Sciences*, vol. 115, pp. 8505–8510, Aug. 2018.
- [7] J. Blechschmidt and O. G. Ernst, “Three ways to solve partial differential equations with neural networks — A review,” *GAMM-Mitteilungen*, vol. 44, no. 2, p. e202100006, 2021.
- [8] M. Raissi, P. Perdikaris, and G. Karniadakis, “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations,” *Journal of Computational Physics*, vol. 378, pp. 686–707, Feb. 2019.
- [9] “SciMLCon 2022.” <https://scimlcon.org/2022/>.
- [10] G. Strang, *Linear Algebra and Learning from Data*. Wellesley: Wellesley-Cambridge press, 2019.
- [11] C. M. Bishop, *Pattern Recognition and Machine Learning*. Information Science and Statistics, New York: Springer, 2006.
- [12] S. R. Dubey, S. K. Singh, and B. B. Chaudhuri, “Activation functions in deep learning: A comprehensive survey and benchmark,” *Neurocomputing*, vol. 503, pp. 92–108, Sept. 2022.

- [13] Y. Chauvin and D. E. Rumelhart, eds., *Backpropagation: Theory, Architectures, and Applications*. New York: Psychology Press, Feb. 1995.
- [14] A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous, and Y. LeCun, “The Loss Surfaces of Multilayer Networks,” Jan. 2015.
- [15] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” Jan. 2017.
- [16] K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators,” *Neural Networks*, vol. 2, pp. 359–366, Jan. 1989.
- [17] K. Hornik, “Approximation capabilities of multilayer feedforward networks,” *Neural Networks*, vol. 4, pp. 251–257, Jan. 1991.
- [18] Y. Lu and J. Lu, “A Universal Approximation Theorem of Deep Neural Networks for Expressing Probability Distributions,” in *Advances in Neural Information Processing Systems*, vol. 33, pp. 3094–3105, Curran Associates, Inc., 2020.
- [19] F. Voigtlaender, “The universal approximation theorem for complex-valued neural networks,” Dec. 2020.
- [20] C. Rackauckas, “SciML/SciMLBook: Parallel Computing and Scientific Machine Learning (SciML): Methods and Applications (MIT 18.337J/6.338J),” <https://github.com/SciML/SciMLBook>.
- [21] K. Zubov, Z. McCarthy, Y. Ma, F. Calisto, V. Pagliarino, S. Azeglio, L. Bottero, E. Luján, V. Sulzer, A. Bharambe, N. Vinchhi, K. Balakrishnan, D. Upadhyay, and C. Rackauckas, “NeuralPDE: Automating Physics-Informed Neural Networks (PINNs) with Error Approximations,” July 2021.
- [22] J. A. Rivera, J. M. Taylor, Á. J. Omella, and D. Pardo, “On quadrature rules for solving Partial Differential Equations using Neural Networks,” *Computer Methods in Applied Mechanics and Engineering*, vol. 393, p. 114710, Apr. 2022.
- [23] C. Rackauckas and Q. Nie, “DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia,” *Journal of Open Research Software*, vol. 5, p. 15, May 2017.
- [24] K. Schwarzschild, “On the gravitational field of a mass point according to Einstein’s theory,” May 1999.
- [25] eigenchris, “(1) Relativity 108a: Schwarzschild Metric - Derivation - YouTube.” <https://www.youtube.com/>.
- [26] M. Innes, A. Edelman, K. Fischer, C. Rackauckas, E. Saba, V. B. Shah, and W. Tebbutt, “A Differentiable Programming System to Bridge Machine Learning and Scientific Computing,” July 2019.
- [27] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, “Automatic Differentiation in Machine Learning: A Survey,” *Journal of Machine Learning Research*, vol. 18, no. 153, pp. 1–43, 2018.
- [28] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 249–256, JMLR Workshop and Conference Proceedings, Mar. 2010.
- [29] A. Sami Bataineh, M. S. M. Noorani, and I. Hashim, “Approximate analytical solutions of systems of PDEs by homotopy analysis method,” *Computers & Mathematics with Applications*, vol. 55, pp. 2913–2923, June 2008.
- [30] C. J. Budd, J. W. Dold, and A. M. Stuart, “Blow-up in a System of Partial Differential Equations with Conserved First Integral. Part II: Problems with Convection,” *SIAM Journal on Applied Mathematics*, vol. 54, pp. 610–640, June 1994.
- [31] B. Benhammouda and H. Vazquez-Leal, “Analytical solutions for systems of partial differential–algebraic equations,” *SpringerPlus*, vol. 3, p. 137, Mar. 2014.