Discrete & Continuous Models
& Applied Computational Science
ISSN 2658-7149 (Online), 2658-4670 (Print)

2025, 33 (3) 284–298
http://journals.rudn.ru/miph

Research article

UDC 004.021:519.6
PACS 07.05.Tp
DOI: 10.22363/2658-4670-2025-33-3-284-298 EDN: HHCVPK

Using NeuralPDE.jl to solve differential equations

Daria M. Belicheva[1], Ekaterina A. Demidova[1], Kristina A. Shtepa[1], Migran N. Gevorkyan[1], Anna V. Korolkova[1], Dmitry S. Kulyabov[1, 2]

[1] RUDN University, 6 Miklukho-Maklaya St, Moscow, 117198, Russian Federation

[2] Joint Institute for Nuclear Research, 6 Joliot-Curie St, Dubna, 141980, Russian Federation

(received: January 20, 2025; revised: February 25, 2025; accepted: March 1, 2025)

Abstract. This paper describes the application of physics-informed neural network (PINN) for solving partial derivative equations. Physics Informed Neural Network is a type of deep learning that takes into account physical laws to solve physical equations more efficiently compared to classical methods. The solution of partial derivative equations (PDEs) is of most interest, since numerical methods and classical deep learning methods are inefficient and too difficult to tune in cases when the complex physics of the process needs to be taken into account. The advantage of PINN is that it minimizes a loss function during training, which takes into account the constraints of the system and th e laws of the domain. In this paper, we consider the solution of ordinary differential equations (ODEs) and PDEs using PINN, and then compare the efficiency and accuracy of this solution method compared to classical methods. The solution is implemented in the Julia programming language. We use NeuralPDE.jl, a package containing methods for solving equations in partial derivatives using physics-based neural networks. The classical method for solving PDEs is implemented through the DifferentialEquations.jl library. As a result, a comparative analysis of the considered solution methods for ODEs and PDEs has been performed, and an evaluation of their performance and accuracy has been obtained. In this paper we have demonstrated the basic capabilities of the NeuralPDE.jl package and its efficiency in comparison with numerical methods.

Key words and phrases: physics-informed neural networks, numerical methods, differential equations, Julia programming language, NeuralPDE

For citation: Belicheva, D. M., Demidova, E. A., Shtepa, K. A., Gevorkyan, M. N., Korolkova, A. V., Kulyabov, D. S. Using NeuralPDE.jl to solve differential equations. *Discrete and Continuous Models and Applied Computational Science* **33** (3), 284–298. doi: 10.22363/2658-4670-2025-33-3-284-298. edn: HHCVPK (2025).

1. Introduction

Physics-informed neural networks (PINNs) are deep learning methods designed to solve various types of differential equations (DEs) that arise in computational science and engineering [1]. By integrating data and physical laws, PINNs construct a neural network that approximates the solution of a DE system. Such a network is obtained through the minimization of a loss function that encodes any prior knowledge about the DEs and the available data.

To demonstrate the functionality of PINNs, we employ the NeuralPDE.jl library [1], implemented in Julia programming language [2, 3]. We consider a simple mathematical model based on a system of ordinary differential equations (ODEs). The system is solved numerically and using PINNs, then the resulting solutions are compared with the first integral of a system.

2. Physics-informed neural networks

Physics-informed neural networks (PINNs) have become an increasingly effective approach for solving differential equations and constructing neural equivalents of physical models. Classical neural networks derive solutions solely based on data represented as pairs of "state–value." A key feature of PINNs is that they take into account the physical laws behind the problem, which are expressed as ordinary or partial differential equations. In other words, the loss function explicitly includes the ODE/PDE terms as well as the initial and boundary conditions. The term PINN was introduced in [4], where it was defined as a new class of universal function approximators capable of encoding fundamental physical laws that can be described by partial differential equations.

2.1. Overview

Consider a differential equation of the following form:

$$F(u(x);\lambda) = 0,$$

where F is a differential operator, u is the solution of the differential equation, λ denotes the parameters of the equation, and $x = x_1, \dots, x_n \in \Omega$ is an n-dimensional coordinate vector defined on the domain Ω.

Let B denote the boundary operator, and let the function u satisfy the following boundary conditions:

$$B(u(x);\lambda) = 0$$

And I denotes the initial condition operator, while the function u satisfies the following initial conditions:

$$I(u(x);\lambda) = 0.$$

PINNs solve partial differential equations (PDEs) by utilizing the Universal Approximation Theorem [5], which states that for any measurable function u, there exists a sufficiently large neural network N with weights w such that $||N(x;w) - u(x)|| < \varepsilon$ for all $x \in \Omega$. This implies that an arbitrary differential equation can be solved by replacing the unknown solution $u(x)$ with a neural network $N(x;w)$ and finding the weights w such that $F(N;\lambda) \approx 0$ for all $x \in \Omega$. Formally, this condition can be expressed as a single equation by summing the the residuals over all points x.

$$L(w) = \int_{\Omega} ||F(N(x;w);\lambda)||dx, \tag{1}$$

where the objective is to determine the neural network weights w that minimize the loss function $L(w)$. In contrast to the exact analytical solution, if $L(w) = 0$, the neural network can be regarded, by definition, as the solution of the corresponding differential equation.

Since the boundary conditions must be satisfied only on a certain subset $\partial\Omega$, it is useful to separate the boundary and initial conditions into their own equation. Thus, we obtain:

$$L(w) = \int_{\Omega\setminus\partial\Omega} ||F(N(x,w);\lambda)||dx + \int_{\partial\Omega} ||B(N(x,w);\lambda)||dx + ||I(N(x,w);\lambda)||. \tag{2}$$

Equation (1) is equivalent to (2), but it provides a clearer view of the implementation. Let us rewrite it using the following notation:

$$L(w) = L_r + L_{ic} + L_{bc},$$

where L_r is the differential equation residual, L_{ic} is the initial condition error, and L_{bc} is the boundary condition error.

2.2. General workflow

The workflow of PINNs consists of the following main steps:

1. Definition of the problem based on physical laws and formulation of the governing equations describing the system.
2. Collection of data representing the system's behavior from experiments, simulations or other sources.
3. Selection of the neural network architecture and initialization of its parameters.
4. Formulation of the loss function, which incorporates both the agreement with experimental data and the satisfaction of the physical equations.
5. Training of the neural network by minimizing the loss function.
6. Verification of the training stopping criteria (e.g., reaching a predefined number of epochs or achieving minimal loss).
7. Analysis and interpretation of the obtained results.

A more detailed workflow of PINNs can be outlined as follows:

1. Definition of the physics-based problem:
   - formulate the governing equations that describe the behavior of the system (these equations may be derived from fundamental principles such as conservation laws or constitutive equations);
   - specify the boundary and initial conditions for the problem.
2. Data collection:
   - obtain data representing the behavior of the system from experiments, simulations, or other sources;
   - prepare the training data by selecting the points (spatial or temporal coordinates) at which predictions and losses will be evaluated.
3. Design and configuration of the neural network architecture:
   - define the type of neural network (e.g., fully connected network);
   - select the number of layers and neurons per layer;
   - choose activation functions;
   - initialize network parameters (weights and biases).
4. Formulation of the loss function:
   - include two main components in the loss function:
     - data fidelity term — measures the discrepancy between network predictions and observed data;
     - physics-informed term — enforces the satisfaction of the governing physical equations as constraints;
   - adjust the weighting of the loss components to balance data accuracy and physical consistency.
5. Training of the neural network:
   - feed input data into the network and compute predictions;
   - evaluate the loss using the formulated loss function;
   - update the network parameters using an optimization algorithm;

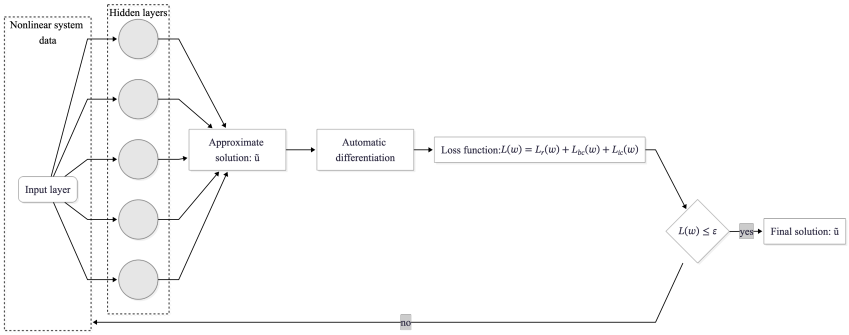

Figure 1. Flow diagram of the PINN training process

- apply automatic differentiation to compute the derivatives required for training.

6. Verification of stopping criteria:
   - monitor whether a predefined number of training epochs (iterations) has been reached;
   - track the minimization of the loss function or other convergence criteria;
   - assess the stability and quality of the predictions.
7. Iterative refinement of training and validation until the stopping conditions are met. This process can be represented by a block diagram (Fig. 1).
8. Evaluation of results:
   - analyze network predictions for consistency with both experimental data and physical laws;
   - assess the accuracy and interpretability of the obtained solution.
9. Interpretation and application of results:
   - use the trained model to predict system behavior under new conditions;
   - apply the model to solve forward and inverse problems, data assimilation tasks, and related applications.
10. Model optimization and refinement (if required):
    - adjust the network architecture, loss function, or training process to improve performance;
    - retrain the model with updated parameters if the results are unsatisfactory.

2.3. Overview of the NeuralPDE package

The `NeuralPDE` package [6] is part of the SciML [7] ecosystem for the Julia programming language. This collection includes packages that enable the computation of mathematical models based on various types of differential equations, combining traditional numerical methods with machine learning techniques.

The `NeuralPDE` package employs neural networks whose loss function incorporates the differential equations defining the mathematical model. This enables the training process to account for the underlying physical laws governing the problem, thereby implementing the concept of PINNs [8].

NeuralPDE is used to address three broad classes of problems:

- approximation of solutions to systems of ordinary differential equations (ODEs);
- approximation of solutions to partial differential equations (PDEs);

– solution of inverse problems, which involve determining the coefficients of ODEs and PDEs from known solutions.

The `NeuralPDE` package has been in active development since 2019 [9]. It can be installed through Julia's standard package manager by executing the command `add NeuralPDE`. Installation involves downloading a substantial number of dependencies, including both additional Julia packages and artifacts. In the Julia ecosystem, artifacts refer to external binary files of libraries or auxiliary utilities required by dependent packages. By default, `NeuralPDE` performs computations on the central processing unit (CPU). To enable graphics processing unit (GPU) acceleration, one must additionally install either `Flux.jl` [10] or `Lux.jl` [11]. The formulation of ordinary and partial differential equations relies on the syntax provided by the `ModelingToolkit.jl` package [12], which is also part of the SciML ecosystem. According to its official documentation, `ModelingToolkit.jl` is a framework for high-performance symbolic–numerical computation designed for mathematical modeling and scientific machine learning. It enables high-level symbolic specification of problems for subsequent numerical computation and analysis. The symbolic representation is built upon the `Symbolics.jl` package [13], which serves as a computer algebra system (CAS) within the Julia environment.

2.4. Solving ODEs in Julia

In mathematical terms, `ODEProblem` represents the following formulation:

$$u' = f(u, p, t)$$

for the interval $t \in (t_0, t_f)$ with the initial condition $u(t_0) = u_0$.

Let us solve a simple ordinary differential equation (ODE):

$$u' = cos(2\pi t)$$

for $t \in (0, 1)$ with $u_0 = 0$, we use `NNODE` and a numerical method, and then compare the results.

The problem is defined using the `ODEProblem` method by specifying the equation, initial condition, and time interval.

```
linear(u, p, t) = cos(t * 2 * pi)
tspan = (0.0, 1.0)
u0 = 0.0
prob = ODEProblem(linear, u0, tspan)
```

We set `NeuralPDE.NNODE()`, an algorithm for solving ordinary differential equations with a neural network. It represents a specialized form of the physics-informed neural network approach that serves as a solver for standard ODE problems. We then specify the neural network architecture using `Lux.jl` by defining a multilayer perceptron (MLP) with one hidden layer of 5 units and a sigmoid activation function, as follows:

```
rng = Random.default_rng()
Random.seed!(rng, 0)
chain = Chain(Dense(1, 5, σ), Dense(5, 1))
ps, st = Lux.setup(rng, chain) |> Lux.f64
```

We use the `solve` method to compute the solution of the defined problem, applying the `Tsit5()` solver with a step size of 0.01 (see Fig. 2):

```
sol_num = solve(prob, Tsit5(), saveat = 0.01)
```

A similar procedure is applied to produce the plot for the time interval $[0, 15]$ (Fig. 3).

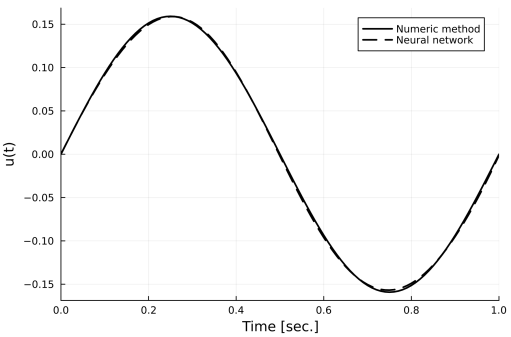

Figure 2. Comparison of solutions over the interval [0, 1]

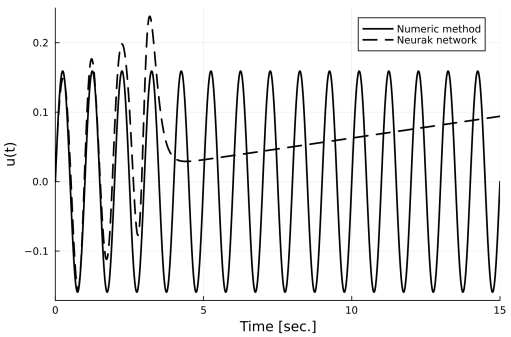

Figure 3. Comparison of solutions over the interval [0, 15]

3. Lotka–Volterra model

3.1. Description of the Lotka–Volterra model

Let us consider the Lotka–Volterra mathematical model (predator–prey model), which describes the interaction between two species of animals. One species preys on the other, while the prey population has access to unlimited food resources [14, 15]. The model is represented by the following system of equations:

$$\begin{cases} \dfrac{dx}{dt} = \alpha x(t) - \beta x(t) y(t), \\ \dfrac{dy}{dt} = -\gamma y(t) + \delta x(t) y(t). \end{cases}$$

In this model x denotes the number of prey and y the number of predators. The coefficient a represents the natural growth rate of the prey population in the absence of predators, while c describes the natural mortality rate of predators deprived of food (prey). The probability of interaction between prey and predators is assumed to be proportional to both their population sizes. Each interaction decreases the prey population but contributes to the growth of the predator population (the terms $-bxy$ and dxy in the right–hand side of the equations).

The first integral of the system can be written as follows:

$$\alpha \log y - \beta y + \gamma \log x - \delta x = C, \ C = \text{const}.$$

To solve the system, we set the parameters $\alpha = 1.5$, $\beta = 1.0$, $\gamma = 3.0$, $\delta = 1.0$. We consider the initial value problem (Cauchy problem) with the following initial conditions:

$$\begin{cases} x(0) = 1, \\ y(0) = 1. \end{cases}$$

3.2. Numerical investigation

We solve the system numerically using `Tsit5()` with a step size of 0.01 from the `DifferentialEquations.jl` library [16]. We consider the time interval $[0, 4]$. The results obtained by the numerical method are shown in Figs. 4, 5.

As shown in Fig. 4, the trajectory of the numerical solution forms a closed curve, indicating that the phase volume is conserved, similar to the analytical case.

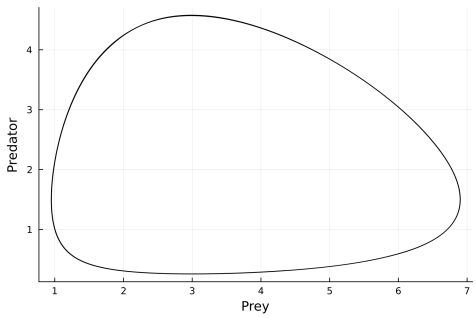

Figure 4. Phase portrait of the Lotka–Volterra model. Numerical method

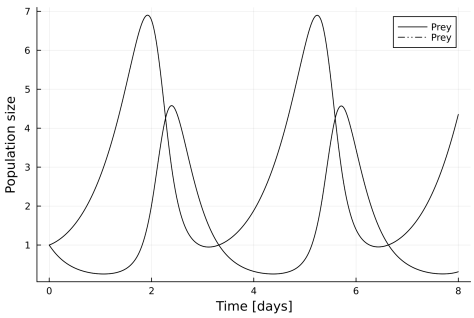

Figure 5. Numerical solution of the Lotka–Volterra model

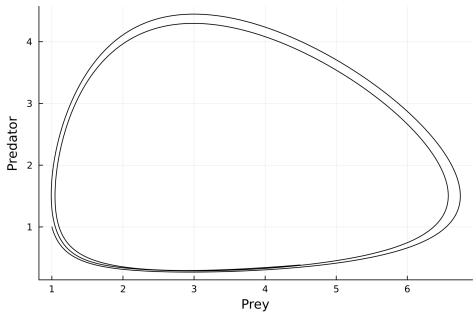

Figure 6. Phase portrait of the Lotka–Volterra model obtained using PINN

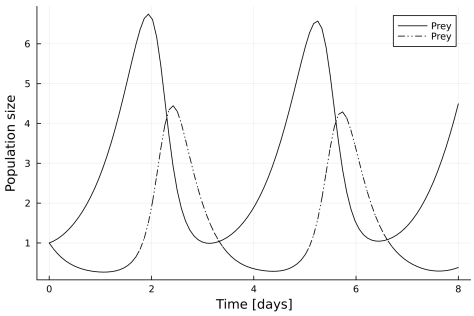

Figure 7. PINN-based solution of the Lotka–Volterra model

3.3. PINN-based solution of the model

We now solve the system using the `NeuralPDE.jl` library. The neural network architecture is defined with the `Lux.jl` library. A three-layer neural network is employed, consisting of one input neuron, two output neurons, and sixteen neurons in the hidden layer. The activation function for the first two layers is the hyperbolic tangent. Parameter optimization is carried out using the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm from the `OptimizationOptimJL.jl` package. The `NeuralPDE.NNODE()` algorithm is used to solve the system of ordinary differential equations. This algorithm is a specialization of the physics-informed neural network (PINN) approach applied to standard ODE problems.

```
chain = Lux.Chain(Lux.Dense(1, 16, tanh),  Lux.Dense(16, 16,tanh),
 ↪  Lux.Dense(16, 2))
opt = OptimizationOptimJL.BFGS()
alg = NeuralPDE.NNODE(chain, opt)
```

We proceed by calling `solve` as in a typical `ODEProblem`. The `verbose` option is enabled to monitor the loss evolution during training. The maximum number of training epochs (iterations) is set to 1000:

```
sol = solve(prob, alg, verbose = true, abstol=1e-8, maxiters = 1000)
```

As a result we obtain the following plots (Fig. 6, 7).

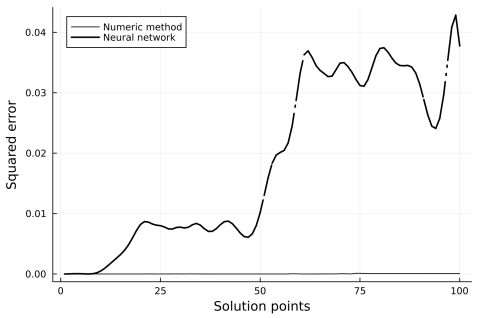

Figure 8. Comparison of solution errors

3.4. Comparative analysis of methods

We compare the phase trajectories obtained from the numerical solution and from the neural network with the first integral of the system. The corresponding squared error plots are shown in Fig. 8.

The plot indicates that the numerical solution provides higher accuracy than the PINN-based one. A performance comparison of the two approaches is then conducted using the `BenchmarkTools.jl` package. The efficiency of the numerical method is assessed in terms of computation time and memory consumption:

```
233.154 μs (7052 allocations: 567.62 KiB)
```

A similar evaluation is performed for the `NeuralPDE.jl` library:

```
2463.046 s (3569138613 allocations: 2682.28 GiB)
```

Neural networks require significantly more computational resources and time compared to numerical methods.

4. SIR model

Consider a system of differential equations whose solutions are aperiodic.

Compartmental models represent a general framework for modeling dynamic systems. They are widely used in the mathematical modeling of infectious diseases, where the population is divided into compartments labeled, for example, S, I, or R (susceptible, infectious, or recovered). Individuals can move between compartments according to the transition rules defined by the model.

The SIR model is one of the simplest compartmental models, and many other models are derived from this basic form. The model consists of three compartments:

- S: the number of susceptible individuals. When a susceptible and an infectious individual come into "infectious contact", the susceptible individual becomes infected and moves to the infectious compartment.
- I: the number of infectious individuals. These are individuals who have been infected and are capable of transmitting the disease to susceptible individuals.
- R: the number of recovered (and immune) individuals. These are individuals who have been infected and recovered from the disease and moved to the recovered compartment. This compartment may also be referred to as "resistant."

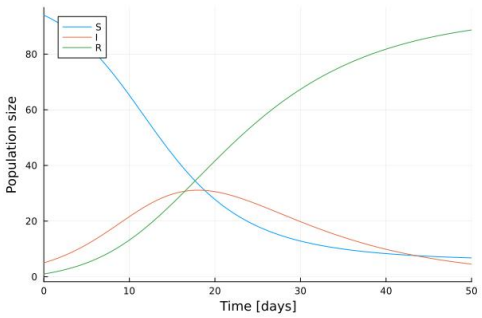

Figure 9. Numerical solution of the SIR model

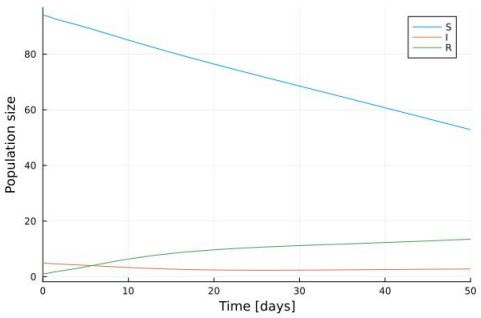

Figure 10. PINN-based solution of the SIR model

As long as the number of infected individuals does not exceed a critical threshold I^*, all infected persons are assumed to be isolated and unable to transmit the disease. Once $I(t) > I^*$, the infected individuals begin to spread the infection among the susceptible population.

The SIR model without vital dynamics (birth and death processes, sometimes referred to as demographic effects) can be formulated as the following system of ordinary differential equations:

$$\begin{cases} \dfrac{dS}{dt} = -\dfrac{\beta IS}{N}, \\ \dfrac{dI}{dt} = \dfrac{\beta IS}{N} - \gamma I, \\ \dfrac{dR}{dt} = \gamma I, \end{cases}$$

where S is the number of susceptible individuals, I is the number of infected individuals, R is the number of recovered individuals, and N is the total population given by the sum of these three compartments. The parameters β and γ represent the infection and recovery rates, respectively.

The system is solved numerically using the `Tsit5()` method with a step size of 0.1 from the `DifferentialEquations.jl` library [16]. We consider the time interval $[0, 50]$ with the initial conditions $S = 990.0$, $I = 10.0$, and $R = 0.0$. The resulting solution is shown in Fig. 9.

Next, we solve the same system using the `NeuralPDE.jl` library. We employ a three-layer neural network with one neuron in the input layer, three neurons in the output layer, and thirty-two neurons in the hidden layer. The activation function in the first two layers is the sigmoid function. The optimization is performed using the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm from the `OptimizationOptimJL.jl` package.

```
chain = Lux.Chain(Lux.Dense(1, 32, σ), Lux.Dense(32, 32, σ), Lux.Dense(32,
↪  3))
opt = OptimizationOptimJL.BFGS()
alg = NeuralPDE.NNODE(chain, opt)
```

With the maximum number of training epochs set to 1000, the resulting solution is shown in Fig. 10.

In the case of a simple epidemic model the solution obtained using a physics-informed neural network demonstrated low accuracy. Therefore, this approach cannot be recommended for solving the given problem.

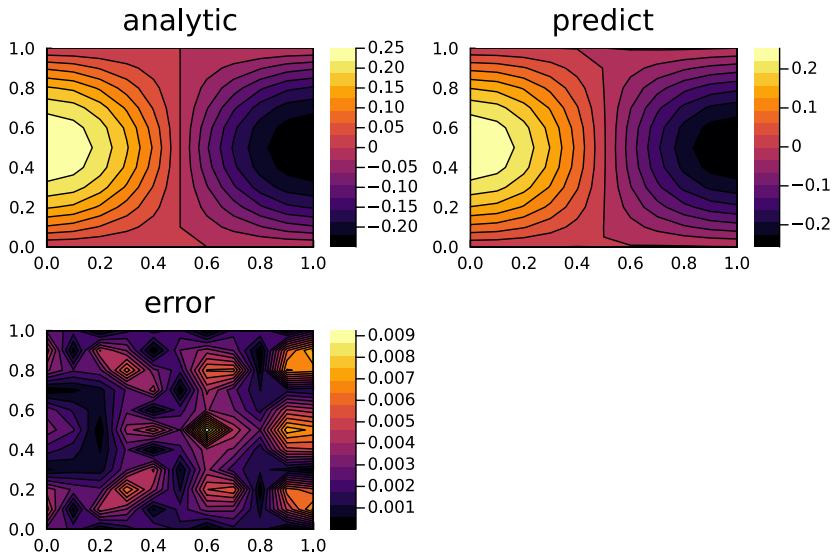

Figure 11. Visualization of the Poisson's equation solution based on the official NeuralPDE documentation

5. Eikonal equation

5.1. Poisson's equation

The official `NeuralPDE` documentation provides an example illustrating the solution of the two-dimensional Poisson's equation.

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = -\sin(\pi x)\sin(\pi y),$$

in a rectangular domain defined by the intervals $x \in [0,1]$ and $y \in [0,1]$, with the following boundary conditions:

$$u(0,y) = 0, u(1,y) = 0,$$
$$u(x,0) = 0, u(x,1) = 0.$$

The computation took approximately 30–40 minutes, resulting in plots consistent with those presented in the official documentation (Fig. 11).

5.2. Solving the Eikonal equation with NeuralPDE

Let us write the Eikonal equation [17–20] in Cartesian coordinates on the plane.

$$\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial y}\right)^2 = n^2(x,y).$$

The function $n(x,y)$ is piecewise continuous.

This example employs the function describing the two-dimensional Maxwell lens [21].

$$n(r) = \begin{cases} \dfrac{n_0}{1 + \left(\dfrac{r}{R}\right)^2}, & r \leqslant R, \\ n_0, & r > R, \end{cases}$$

where $r = \sqrt{(x - x_0)^2 + (y - y_0)^2}$ is the distance from the center of coordinates to points on the lens, which has a circular shape. The center of the lens is located at $(x_0, y_0) = (0, 0)$. The lens radius is $R = 1$, and the refractive index of the medium is $n_0 = 1$.

The example from the official documentation was modified by replacing the Poisson's equation and its boundary conditions with the Eikonal equation. The first modification involved substituting the second derivatives with the squares of the first derivatives:

```
Dx = Differential(x)
Dy = Differential(y)
```

Additionally, for the Maxwell lens:

```
eq = Dx(u(x, y))*Dx(u(x, y)) + Dy(u(x, y))*Dy(u(x, y)) ~ n(x, y)
```

In the `Differential` function, the symbol `^` specifies the order of differentiation rather than exponentiation. For this reason, it was replaced by an explicit multiplication of the derivative by itself. Also the boundary conditions and the domains of x, y were adjusted:

```
# Boundary condition (indicating that the point source is located at the
 ↪ origin)
bcs = [
    u(-1, 0) ~ 0.0
]
# Domain (x, y)
domains = [x in (-1.0, 2.0), y in (-1.0, 2.0)]
```

After several simplifications and the elimination of most internal variables, the `n(x, y)` function was represented as follows:

```
function n(x, y)
  r = hypot(x, y)
  if r <= 1
    return 1 / (1 + r^2)
  else
    return 1
  end
end
```

However, the program failed to execute in this configuration. After neural network initialization (which took about 10 minutes), the program crashed with the error about a non-Boolean value used in a Boolean context: `ERROR: TypeError: non-boolean (Num) used in boolean context`.

According to the official documentation, this error is documented and can arise from two primary causes:

- the `hypot` function is not available in symbolic form, meaning it is not implemented within the `Symbolics.jl` package;
- the **`if`**–**`else`**–**`end`** construct is not supported for symbolic expressions and must be replaced with the `Base.ifelse` function from the standard library.

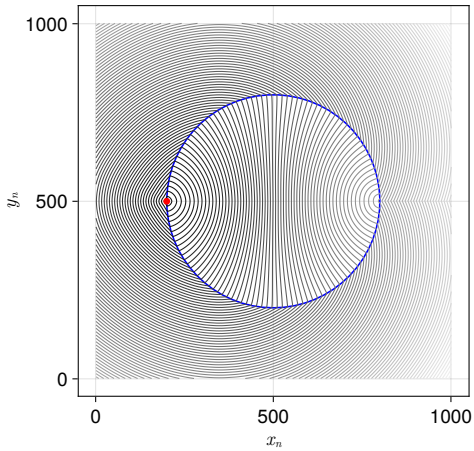

Figure 12. Wavefronts for the Maxwell lens

After applying these modifications, the refraction function was expressed as follows:

```
n(x, y) = ifelse(sqrt(x^2+y^2) <= 1, 1 / (1 + (x^2 + y^2))^2, 1)
```

After these modifications, the program executed successfully.
The processed results are presented in Fig. 12.

6. Discussion

The analysis of the `NeuralPDE` package revealed several drawbacks:

- The package has a large number of dependencies (over one hundred), including both other Julia packages and external binary files (utilities and libraries). As a result, it requires significant disk space and installation time. However, the main issue with the large dependency set is the reduced reliability and stability of the package.
- The differential equation and boundary conditions are defined in symbolic form, which provides only limited support for standard language constructs. Even for a simple function $n(x, y)$, the **`if-else-end`** statement and the standard `hypot` function did not work as expected. It is particularly unintuitive that **`if-else-end`** must be replaced with `Base.ifelse`.
- Computation time is significantly greater compared to classical numerical methods. While traditional numerical schemes complete calculations within tens of seconds, execution time using `NeuralPDE` reaches several tens of minutes.

7. Conclusion

A comparative analysis was carried out for solving the Lotka-Volterra system of differential equations, the epidemiological (SIR) model, and the eikonal equation using both a numerical method and a physics-informed neural network. The implementation was performed in the Julia programming language using the `DifferentialEquations.jl` and `NeuralPDE.jl` libraries. It was concluded

that currently neural network-based numerical methods cannot be regarded as a “silver bullet”. Further research is required to determine the scope and limitations of their applicability.

Author Contributions: Conceptualization, Dmitry S. Kulyabov; methodology, Dmitry S. Kulyabov, Migran N. Gevorkyan; writing, Daria M. Belicheva, Ekaterina A. Demidova, Christina A. Shtepa; writing—review and editing Anna V. Korolkova. All authors have read and agreed to the published version of the manuscript.

Funding: This paper has been supported by the RUDN University Strategic Academic Leadership Program.

Data Availability Statement: Data sharing is not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Declaration on Generative AI: The authors have not employed any Generative AI tools.

References

1. Zubov, K. *et al. NeuralPDE: Automating Physics-Informed Neural Networks (PINNs) with Error Approximations* 2021. doi:10.48550/ARXIV.2107.09443.
2. Engheim, E. *Julia as a Second Language* 400 pp. (Manning Publications, 2023).
3. Engheim, E. *Julia for Beginners. From Romans to Rockets* 472 pp. (Leanpub, 2020).
4. Raissi, M., Perdikaris, P. & Karniadakis, G. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. Journal of Computational Physics. *J. Comput. Phys.* doi:10.1016/j.jcp.2018.10.045 (2018).
5. Hornik, K. Approximation capabilities of multilayer feedforward networks. *Neural networks* **4,** 251–257 (1991).
6. Zubov, K. *et al.* NeuralPDE: Automating Physics-Informed Neural Networks (PINNs) with Error Approximations. doi:10.48550/ARXIV.2107.09443 (2021).
7. *SciML Open Source Scientific Machine Learning* https://github.com/SciML.
8. Bararnia, H. & Esmaeilpour, M. On the application of physics informed neural networks (PINN) to solve boundary layer thermal-fluid problems. *International Communications in Heat and Mass Transfer* **132,** 105890. doi:10.1016/j.icheatmasstransfer.2022.105890 (2022).
9. *SciML Open Source Scientific Machine Learning* https://github.com/SciML/NeuralPDE.jl.
10. *Flux The Elegant Machine Learning Stack* https://fluxml.ai/.
11. *LuxDL DocsElegant and Performant Deep Learning in JuliaLang* https://lux.csail.mit.edu/stable/.
12. Ma, Y., Gowda, S., Anantharaman, R., Laughman, C., Shah, V. & Rackauckas, C. *ModelingToolkit: A Composable Graph Transformation System For Equation-Based Modeling* 2021. arXiv: 2103.05244 [cs.MS].
13. Gowda, S., Ma, Y., Cheli, A., Gwóźdź, M., Shah, V. B., Edelman, A. & Rackauckas, C. High-Performance Symbolic-Numerics via Multiple Dispatch. *ACM Commun. Comput. Algebra* **55,** 92–96. doi:10.1145/3511528.3511535 (Jan. 2022).
14. Volterra, V. *Leçons sur la Théorie mathématique de la lutte pour la vie* (Gauthiers-Villars, Paris, 1931).
15. Demidova, E. A., Belicheva, D. M., Shutenko, V. M., Korolkova, A. V. & Kulyabov, D. S. Symbolic-numeric approach for the investigation of kinetic models. *Discrete and Continuous Models and Applied Computational Science* **32,** 306–318. doi:10.22363/2658-4670-2024-32-3-306–318 (2024).
16. Rackauckas, C. & Nie, Q. DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia. *Journal of Open Research Software* **5,** 15–25. doi:10.5334/jors.151 (2017).

17. Bruns, H. *Das Eikonal* in *Abhandlungen der Königlich-Sächsischen Gesellschaft der Wissenschaften* (S. Hirzel, Leipzig, 1895).
18. Klein, F. C. Über das Brunssche Eikonal. *Zeitscrift für Mathematik und Physik* **46,** 372–375 (1901).
19. Fedorov, A. V., Stepa, C. A., Korolkova, A. V., Gevorkyan, M. N. & Kulyabov, D. S. Methodological derivation of the eikonal equation. *Discrete and Continuous Models and Applied Computational Science* **31,** 399–418. doi:10.22363/2658-4670-2023-31-4-399-418 (Dec. 2023).
20. Stepa, C. A., Fedorov, A. V., Gevorkyan, M. N., Korolkova, A. V. & Kulyabov, D. S. Solving the eikonal equation by the FSM method in Julia language. *Discrete and Continuous Models and Applied Computational Science* **32,** 48–60. doi:10.22363/2658-4670-2024-32-1-48-60 (2024).
21. Kulyabov, D. S., Korolkova, A. V., Sevastianov, L. A., Gevorkyan, M. N. & Demidova, A. V. *Algorithm for Lens Calculations in the Geometrized Maxwell Theory* in *Saratov Fall Meeting 2017: Laser Physics and Photonics XVIII; and Computational Biophysics and Analysis of Biomedical Data IV* (eds Derbov, V. L. & Postnov, D. E.) **10717** (SPIE, Saratov, Apr. 2018), 107170Y.1–6. doi:10.1117/12.2315066. arXiv: 1806.01643.

Information about the authors

Belicheva, Daria M.—Master student of Department of Probability Theory and Cyber Security of RUDN University (e-mail: dari.belicheva@yandex.ru, ORCID: 0009-0007-0072-0453)

Demidova, Ekaterina A.—Master student of Department of Probability Theory and Cyber Security of RUDN University (e-mail: eademid@gmail.com, phone: +7 (495) 955-09-27, ORCID: 0009-0005-2255-4025)

Shtepa, Kristina A.—Assistent Professor of Department of Probability Theory and Cyber Security of RUDN University (e-mail: shtepa-ka@rudn.ru, ORCID: 0000-0002-4092-4326, ResearcherID: GLS-1445-2022)

Gevorkyan, Migran N.—Docent, Candidate of Sciences in Physics and Mathematics, Associate Professor of Department of Probability Theory and Cyber Security of RUDN University (e-mail: gevorkyan-mn@rudn.ru, ORCID: 0000-0002-4834-4895, ResearcherID: E-9214-2016, Scopus Author ID: 57190004380)

Korolkova, Anna V.—Docent, Candidate of Sciences in Physics and Mathematics, Associate Professor of Department of Probability Theory and Cyber Security of RUDN University (e-mail: korolkova-av@rudn.ru, ORCID: 0000-0001-7141-7610, ResearcherID: I-3191-2013, Scopus Author ID: 36968057600)

Kulyabov, Dmitry S.—Professor, Doctor of Sciences in Physics and Mathematics, Professor of Department of Probability Theory and Cyber Security of RUDN University; Senior Researcher of Laboratory of Information Technologies, Joint Institute for Nuclear Research (e-mail: kulyabov-ds@rudn.ru, ORCID: 0000-0002-0877-7063, ResearcherID: I-3183-2013, Scopus Author ID: 35194130800)

УДК 004.021:519.6
PACS 07.05.Tp
DOI: 10.22363/2658-4670-2025-33-3-284-298 EDN: HHCVPK

Применение NeuralPDE.jl для решения дифференциальных уравнений

Д. М. Беличева[1], Е. А. Демидова[1], К. А. Штепа[1], М. Н. Геворкян[1], А. В. Королькова[1], Д. С. Кулябов[1, 2]

[1] Российский университет дружбы народов им. Патриса Лумумбы, ул. Миклухо-Маклая, д. 6, Москва, 117198, Российская Федерация

[2] Объединённый институт ядерных исследований, ул. Жолио-Кюри, д. 6, Дубна, 141980, Российская Федерация

Аннотация. Работа описывает применение Physics Informed Neural Network (PINN) для решения уравнений в частных производных. Physics Informed Neural Network — это вид глубокого обучения, который учитывает физические законы для более эффективного решения физических уравнений по сравнению с классическими методам. Наибольший интерес представляет решение уравнений в частных производных (УЧП), так как численные методы и классические методы глубокого обучения не эффективны и слишком сложно настраиваемы в случаях, когда необходимо учесть сложную физику процесса. Преимуществом PINN является то, что при обучении она минимизирует функцию потерь, которая учитывает ограничения системы и законы предметной области. В работе мы рассматриваем решение обыкновенных дифференциальных уравнений (ОДУ) и УЧП с помощью PINN, а затем сравниваем эффективность и точность этого метода решения по сравнению с классическими. Решение реализовано на языке программирования Julia. Мы используем NeuralPDE.jl - пакет, содержащий методы решения уравнений в частных производный с помощью нейронных сетей, основанных на физике. Классический метод решения УЧП реализован посредством библиотеки DifferentialEquations.jl. В результате был проведен сравнительный анализ рассматриваемых методов решения для ОДУ и УЧП, а также получена оценка их производительности и точности. В этой статье мы продемонстрировали базовые возможности пакета NeuralPDE.jl и его эффективность по сравнению с численными методами.

Ключевые слова: нейронные сети на основе физики, численные методы, дифференциальные уравнения, язык программирования Julia, пакет NeuralPDE.jl