International Journal of Mathematical Education in Science and Technology

ISSN: (Print) (Online) Journal homepage: https://www.tandfonline.com/loi/tmes20

Literate programming for motivating and teaching neural network-based approaches to solve differential equations

Alonso Ogueda-Oliva & Padmanabhan Seshaiyer

To cite this article: Alonso Ogueda-Oliva & Padmanabhan Seshaiyer (19 Sep 2023): Literate programming for motivating and teaching neural network-based approaches to solve differential equations, International Journal of Mathematical Education in Science and Technology, DOI: 10.1080/0020739X.2023.2249901

To link to this article: https://doi.org/10.1080/0020739X.2023.2249901

INTERNATIONAL JOURNAL OF MATHEMATICAL EDUCATION IN SCIENCE AND TECHNOLOGY
https://doi.org/10.1080/0020739X.2023.2249901

CLASSROOM NOTE

Literate programming for motivating and teaching neural network-based approaches to solve differential equations

Alonso Ogueda-Oliva and Padmanabhan Seshaiyer

Department of Mathematical Sciences, George Mason University, Fairfax, VA, USA

ABSTRACT

In this paper, we introduce novel instructional approaches to engage students in using modelling with data to motivate and teach differential equations. Specifically, we introduce a pedagogical framework that will execute instructional modules to teach different solution techniques for differential equations through repositories and notebook environments during real-time instruction. Each of these teaching modules employs a literate programming approach that uses the notebook environment to explain the concepts in a natural language, such as English, interspersed with snippets of macros and traditional source code on a web browser. The pedagogical approach employed is reproducible and leads to openaccess material for students to motivate and teach differential equations efficiently. We will share examples of this framework applied to teaching advanced concepts such as machine learning and neural network approaches for solving ordinary and partial differential equations as well as estimating parameters in these equations for given datasets. More details of the work can be accessed from https://aoguedao.github.io/teaching-ml-diffeq.

ARTICLE HISTORY
Received 17 March 2023

KEYWORDS
Literate programming; differential equations; machine learning

1. Introduction

Undergraduate students are first exposed to differential equations in Calculus when they learn about the notion of a derivative followed by the anti-derivative and finally connecting these two big concepts through the Fundamental Theorem of Calculus. Once exposed, they learn that these differential equations or systems of differential equations often do not admit exact solutions and hence they need to understand numerical methods to solve them (Seshaiyer, 2017).

While there have been several efforts to revitalise the calculus curriculum and its instruction, less well-publicised and far less well-researched are the changes occurring in the content, pedagogy, and learning of differential equations. With evolving interests in

CONTACT P. Seshaiyer pseshaiy@gmu.edu

Supplemental data for this article can be accessed here. https://doi.org/10.1080/0020739X.2023.2249901

dynamical systems and motivated by calculus reform efforts, there have been technological advances to incorporate graphical and numerical approaches to understand and analyse solutions to differential equations that were previously reserved for advanced undergraduate or graduate study. Technology is also often used to enhance the student learning of differential equations and to help them appreciate the applications of these equations to real-world problems through mathematical modelling (Ding et al., 2019; McCarthy et al., 2019; Seshaiyer & Solin, 2017). An example is the application to understand and study infectious diseases using compartmental models described via coupled differential equations (Seshaiyer, 2017).

One of the innovative instructional approaches to engage students in learning content along with structured programming was introduced about four decades back (Knuth, 1984). Specifically, the goal was to change a traditional attitude about the construction of the programmes. The main task was to instruct a computer on what to do giving more time for the instructor to explain to students what we want a computer to do. This idea of programming came to be known as *literate programming*, where instruction drives the coding and the output of the piece of code informs the instruction in real-time. Over the years the literate programming style has been used to enhance student learning of differential equations (Nedialkov, 2011; Seshaiyer & Solin, 2017).

A grand challenge that still exists is the need to develop a coherent learning framework that enables researchers to blend differential equations with the vast amount of data sets that are available. This is also needed to estimate the parameters used in the governing differential equations efficiently (Raissi, Ramezani, et al., 2019). In recent years, mathematical modelling has gained the attention of researchers and educators with a need to develop mathematical proficiency for students to be able to apply the mathematics they know to solve problems arising in everyday life, society, and the workplace. Specifically, mathematical modelling is an iterative problem-solving process that goes through the following steps.

- **Observe**: Here an observation process leads to help posing open-ended problems from a real-world context.
- **Theorise**: Next, one makes appropriate assumptions that are physically meaningful and identifies constraints and variables that are relevant to the situation.
- **Formulate**: Once theorised, the next step is to build a mathematical formulation that describes relationships.
- **Analyse**: With a possible equation or expression formulated, the next step is to analyse and interpret the mathematical solution.
- **Simulate**: To understand the qualitative behaviour of the solution, this next step often involves performing numerical computations.
- **Validate**: Once we have potential solutions, the next step would be to validate them against known models or datasets (if appropriate).
- **Predict**: In this final step, one can use what has been created to make predictions and evaluate how close the model predicts the observation, completing the modelling cycle.

In this work, we introduce a novel instructional approach using literate programming to engage students in using modelling with data to motivate and teach differential equations. In Section 2, we introduce the methodology that we have used to develop the four

instructional modules along with an instructional toolkit with a workflow for teaching and learning. Section 3 presents review material on what the students may independently prepare as they move through the scaffolded modules as the instructor presents each via literate programming. In Section 4, the students are introduced to solving system of two differential equations using a classical numerical approach that is often used to solve ordinary differential equations. Also, in this section, an alternate approach called Physics Informed Neural Networks (PINNs) is introduced to solve the problem in a forward fashion and compared to the traditional Runge–Kutta algorithm and also used to solve the problem in an inverse fashion to determine the optimal parameters in the system. Section 5 provides an opportunity for the students to engage a real-world problem of solving infectious diseases modelled by governing differential equations by building on the PINNs technique presented in the previous section. Section 6 summarises this work as a conclusion and discusses potential extensions for future work.

2. Methodology

Along with the ability to learn to model physical systems by differential equations, it is also important for students to learn to compute a model's parameter values from known data sets (Raissi, Ramezani, et al., 2019). Often some of the parameters in the equations may be estimated from patterns in the data, but most of them have to be computed using heuristic algorithms that are computationally motivated such as inverse methods, least-squares approach, agent-based modelling and more or statistically motivated such as maximum-likelihood, Bayesian inference and Poisson regression methods. While these are sophisticated techniques that have their advantages and disadvantages, students often do not become aware of these unless they work on a research project with a mentor (often at a graduate level). So one of the ideas is to see if there are techniques that we can bring into an undergraduate classroom that can be integrated with the teaching of differential equations that will help determine unknown parameters. One of the alternative and powerful approaches, that teaches computers to process data in a way that is inspired by the human brain is using a neural network. The connection between the neurons in our brain to communicate and make data-driven decisions has helped areas such as machine learning to evolve over the past decade.

There is however, a great need to take advanced concepts such as machine learning and neural networks to create a structured curriculum to teach aspects of differential equations that involves a scaffolded learning progression consisting of introductions to concepts grounded in contextual experiences, hands-on activities, interactive web-based explorations, and analysing and modifying existing code used to build, train and test models within cloud-based notebooks. This is the focus of this paper. Specifically, we introduce four teaching and learning modules that successively build student knowledge as they progress through them. These include:

- **Module 0**: This is a required prerequisite reading for the students to get motivated to learn about differential equations and exposure to some numerical methods for solving these differential equations. For simplicity, we are focusing on Ordinary Differential Equations (ODEs) and applications.

- **Module 1**: Here we develop the understanding of applying the background techniques learnt in Module 0 to simulate the numerical solution to a coupled system of differential equations using two different approaches including the traditional Runge–Kutta approach and a Physics Informed Neural Network (PINNs) approach. For simplicity, we consider a predator-prey example that is presented using a literate programming approach to **simulate** the equations using the two different approaches.
- **Module 2**: Now that the student learns to solve a system of differential equations in a forward fashion in multiple ways, we introduce the application of PINNs as an inverse approach to compute and **validate** parameters in the differential equations that correspond to a given data set.
- **Module 3**: Finally, we engage the students in applying what they have learnt through a system of two differential equations to an example from public-health, specifically, disease dynamics that consists of a compartmental model with four coupled differential equations along with a known data set. The challenge for the students is to make the necessary revisions and updates to the code to reflect the new model; employ PINNs to identify the parameters in this model and; use the value of the parameters obtained to run the model to help **predict** the dynamics of the disease.

Next, we introduce some of the foundational computing tools that would be needed to help guide the instructor to use the instructional modules presented herein within their own classrooms to enhance student learning of differential equations. The material in this work has been created with the intention to not use technology as simply a tool for simulation or computation. But it is really to help educators move beyond using technology as an afterthought to technology-enhanced instruction.

2.1. Technology-enhanced instructional toolkit

As educators, we constantly seek new technologies to engage and intrigue our students. With the amount of open source software and the availability of free web-based tools, it is important to select appropriate technology that will first engage and capture their interest, then improve and maximise their understanding and retention of the content, and finally encourage and reinforce their own independent creative work. We also need to pick and introduce tools to our students that help them to practice peer-reviewing. Finally, it is important to choose free or low-cost technology in the classroom, that will permit instructors to employ technology that they might otherwise be unable to afford. In Table 1, we list a collection of technology resources that will form an instructional toolkit that goes with the four modules outlined earlier to help collectively support the learning of differential equations for students.

2.2. Workflow

One of the great tools for teaching literate programming is Jupyter Notebooks, since instructors can write rich text with a markdown format, which allows one to display mathematical formulas and explanations along with an interactive code students can write and practice in the same document, run and modify it multiple times. Jupyter notebooks support several programming languages, in particular, we have used Python since it is open

Table 1. Summary of tools for the proposed methodology, including a brief description and advantages.

Name	Description	Advantages
Python	General purpose programming language	Used of Scientific computation, simulation, machine learning. Scalable and platform-agnostic
Jupyter Notebook	Web-based interactive computing platform	Contains a complete record of the user's sessions and includes code, narrative text, equations, and rich output
Jupyter Book	A wrapper around Python tools	Build books, documentation and lecture notes from computational content
Git	Free and open source distributed version control system	Handles small to very large projects with speed and efficiency
GitHub	A code hosting platform for version control and collaboration	Lets students to work together on projects from anywhere
Google Colab	A hosted Jupyter notebook service	Requires no setup to use, while providing access free of charge to computing resources including GPUs. Enables students to write and execute arbitrary python code through the browser, and is especially well suited to machine learning, data analysis and education

source and has a strong support community. There are several scientific packages that make Python one of fastest-growing programming Languages in the world. Since the proposed modules focus on ordinary differential equations, we would need to use built-in packages such as `NumPy`, `SciPy` and `DeepXDE`.

While Jupyter notebooks are useful, there are also challenges. For instance, the process of sharing Jupyter notebooks between students and instructors sometimes can become very tedious, because of the need to upload/download each file under each lecture. That adds another layer of complexity. Jupyter Notebooks are JSON files that need to be rendered in order to show rich text and code. Learning management systems (LMS) like BlackBoard or Moodle do not offer this functionality by default, and even if it was possible, they only offer a reading mode. This does not allow one to run code there. Storage platforms like Google Drive, Dropbox, OneDrive, etc. are also not solutions for this problem either.

Git-based source code repository hosting services such as GitHub, BitBucket, GitLab, etc. can renderise Jupyter Notebooks easily, but they still lack interactivity. This is a feature of these platforms since these tools are made for hosting code and track changes in a rigorous way. Finally, there are cloud-based solutions such as Google Colab, Binder, Amazon Sagemaker, Azure Notebook, Kaggle, etc. that include a nice feature of allowing students and instructors to open an interactive session from a URL. So one of the popular choices is to store modules or lessons as Jupyter Notebooks in GitHub and then allow students to open those links from Google Colab.

There also exists a workflow where instructors can upload Jupyter Notebooks to GitHub and then students will be able to interact with them using Google Colab. However, this approach is not organic and it seems like several scattered lessons that involve single files without communication with each other. We propose a more organic workflow on top of the previous one with three defined goals:

- Contents have to be well organised, showing coherence between lessons (similar to a book);
- It has to be easily accessible by students (avoid installing any software as first task) and;
- It has to be flexible and easy to update by instructors.

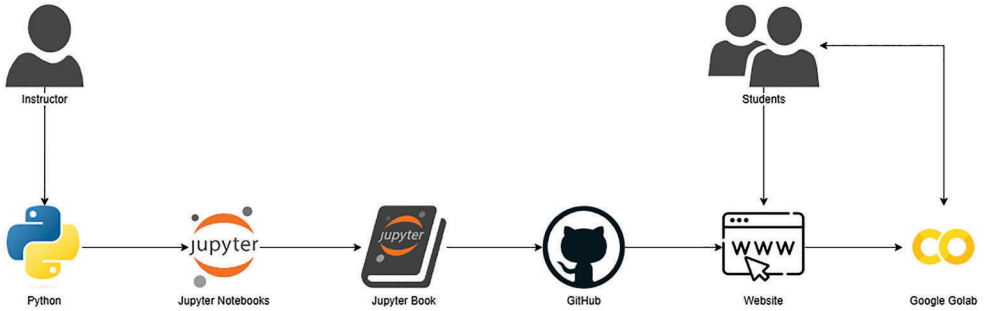

Figure 1. Workflow of tools for students and instructors.

Here, Jupyter Books plays a critical role, it permits one to gather several Jupyter Notebooks and organise them in chapters and sections with only configuration and table of contents files, so instructors do not need to learn another tool. Jupyter Books builds HTML files that are necessary for a website, which can be stored in a GitHub repository in order to enable GitHub pages, a host for websites based in repositories. Now, students can access directly a website with all the lessons (or modules) well organised. Finally, by modifying just a few lines of code in the Jupyter Book's configuration file, one can add a button in each lesson to open it in a Google Colab session. Students do not need to install Python or Jupyter on their computers, but with a single click on the button *Launch on Colab* (upper-right corner) the lesson will be opened on a new and personal Google Colab instance. Students can explore the lesson while they follow the class, but also they are able to modify code on the fly to understand better each execution. Final workflow of this proposed description is shown in Figure 1. Most of the work is to set up the website and then upload content of each lesson while students only interact with the website and Google Colab.

Advanced students would like to use their own laptops or personal computers, for that purpose, they need to install Python and the additional packages mentioned before. This installation process is out of the scope of this work, but it is recommendable to use an environment manager, for instance, `conda`, `mamba` or a Python native solution as `venv + pip`. It is a standard practice to provide a requirements file (list of packages and their versions) on the GitHub repository which users can use for setting their own environments. It is important to notice here each lesson is based on Jupyter Notebook files (`.ipynb`), hence it is recommendable to install Jupyter Lab (or Jupyter Notebook) for reading these files. Students can download every lesson at once from the GitHub repository (direct download or using `git` commands). However, an easier option it is just to press the *Download source file* button in the upper-right corner of each lesson for getting each file at a time.

Next, we provide details of each of the four modules that were itemised earlier. Module 0 forms the backbone foundation that includes a quick review of both motivation for differential equations as well as numerical approaches to solving ODEs. The instructor may use this module as a flipped classroom or asynchronous learning for the students to independently review and then come to class. Modules 1–3, then describe how instructors can use the workflow description provided in the last section with literate programme to engage students in technology-enhanced learning of differential equations.

3. Module-0: background preparation

An equation involving one dependent variable and its derivatives with respect to one or more independent variables is called a *differential equation*. Many of the general laws of nature in science find their most natural expression in the language of differential equations. This also extends to many applications of mathematics, especially in geometry, and in engineering, economics, and many other fields of applied science (Simmons, 2016). Some typical examples of differential equations include:

$$\frac{\mathrm{d}y}{\mathrm{d}t} = -ky$$
$$m\frac{\mathrm{d}^2y}{\mathrm{d}t^2} = -ky$$
$$\frac{\mathrm{d}y}{\mathrm{d}x} + 2xy = \mathrm{e}^{-x^2}$$
$$\frac{\mathrm{d}^2y}{\mathrm{d}x^2} - 5\frac{\mathrm{d}y}{\mathrm{d}x} + 6y = 0$$

where the dependent variable in each of these equations as y, and the independent variable is either t or x. The letters k and m represent constants (usually associated with some physical phenomenon).

An *ordinary differential equation* or ODE is one in which there is only one independent variable so that all the derivatives occurring in it are ordinary derivatives. The *order* of a differential equation is the order of the highest derivative present. A general ordinary differential equation of nth order can be written as

$$F\left(x, y, \frac{\mathrm{d}y}{\mathrm{d}x}, \frac{\mathrm{d}^2y}{\mathrm{d}x^2}, \ldots, \frac{\mathrm{d}^ny}{\mathrm{d}x^n}\right) = 0$$

where F is just an operator. Any adequate theoretical discussion of this equation would have to be based on a careful study of explicitly assumed properties of the function F.

There are several solution methods for different types of ordinary differential equations that yield analytical solutions. However, not all ODEs admit analytical solutions and then it is necessary to apply numerical methods. Next, we will review high-order methods for solving ordinary differential equations numerically.

Single-step Runge–Kutta methods associate a function $\Phi(t, y, h)$ which requires (possibly repeated) function evaluations of $f(t, y)$ but not its derivatives. In general, single-step Runge–Kutta methods have the form:

$$y_0 = y(a)$$
$$y_{k+1} = y_k + h\Phi(t_k, y_k, h)$$

where

$$\begin{aligned}\Phi(t_k, y_k, h) &= \sum_{r=1}^{R} c_r K_r,\\ K_1 &= f(t, y),\\ K_r &= f(t + a_r h, y + h \sum_{s=1}^{r-1} b_{rs} K_s),\\ a_r &= \sum_{s=1}^{r-1} b_{rs}, \quad r = 2, 3, \ldots, R\end{aligned}$$

The most well-known Runge–Kutta scheme (Ackleh et al., 2009) is 4th order; it has the form:

$$\begin{aligned}y_0 &= y(t_0)\\ y_{k+1} &= y_k + \frac{h}{6}\left[K_1 + 2K_2 + 2K_3 + K_4\right]\\ K_1 &= f(t_k, y_k)\\ K_2 &= f\left(t_k + \frac{h}{2}, y_k + \frac{h}{2}K_1\right)\\ K_3 &= f\left(t_k + \frac{h}{2}, y_k + \frac{h}{2}K_2\right)\\ K_4 &= f(t_k + h, y_k + hK_3)\end{aligned}$$

i.e. $\Phi(t_k, y_k, h) = \frac{h}{6}[K_1 + 2K_2 + 2K_3 + K_4]$.

Let us consider the following example of an initial value problem

$$\begin{aligned}y'(t) &= \frac{t}{9}\cos(2y) + t^2\\ y(0) &= 1\end{aligned} \tag{1}$$

After logging into a notebook environment (such as Google Colab), to solve this example, one should first include some python packages that are needed for scientific computing such as (`numpy`), plotting (`matplotlib`) and a tool for solving initial values problems with Runge–Kutta methods (`scipy.integrate.solve_ivp`).

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
```

Next, we define a function by importing from the numerical python (`np`) library which takes as arguments t and y and returns the right side of the equation. In this case,

$$f(t, y) = \frac{t}{9}\cos(2y) + t^2$$

```
def f(t, y):
    return t / 9 * np.cos(2 * y) + t ** 2
```

Now, we need to define the domain over which we will solve the equation. Since our initial condition is given at $t_0 = 0$ our domain must include it. And consider, just as an example, the final time as $t_n = 10$ and $h = 0.5$ (this is an opportunity for the instructor to give the students to later try smaller values to see numerical convergence).

```
t0 = 0
tn = 10
h = 0.5
t_array = np.arange(t0, tn, h)
t_array
```

Note that `solve_ivp` function only needs $f(t, y)$, a time span (`t_span`) and initial conditions (`y_0`). However, we can include the points where the solution we will be evaluated with `t_eval`.

```
sol = solve_ivp(f, t_span=[0, 10], y0=[1], t_eval=t_array)
```

Finally, one can retrieve the solution values with `sol.y`. But for plotting, we need a flat array (just use `.flatten()` method).

```
y_sol = sol.y.flatten()
plt.plot(t_array, y_sol, linestyle="dashed")
plt.xlabel(r"$t$")
plt.ylabel(r"$y(t)$")
plt.title("Numerical solution using Runge-Kutta method")
plt.show()
```

The final result of the numerical solution is seen in Figure 2. At this point, the instructor can allow the students to go back to the different blocks of the code and change parameters such as the step size h and see the effect of doing this in the solution. The students may be asked to halve the step size repeatedly and observe the error between the exact solution (if it is available for the problem) and the numerical solution and ask students to discover the rate of convergence. This will help them to understand why the method gets the name 'Fourth' order Runge–Kutta.

4. Module-1: system of differential equations

After working with first-order equations, it is natural to motivate students to learn about systems such as *Lotka-Volterra equations*. A good example for motivating this is the *predator-prey equations* which are a pair of first-order nonlinear coupled ordinary differential equations, frequently used to describe the dynamics of biological systems in which two species interact. The populations change through time according to the pair of equations:

$$\begin{aligned} \frac{\mathrm{d}x}{\mathrm{d}t} &= \alpha x - \beta xy \\ \frac{\mathrm{d}y}{\mathrm{d}t} &= -\gamma y + \delta xy \end{aligned} \tag{2}$$

where x is the number of prey; y is the number of some predator; $\frac{\mathrm{d}y}{\mathrm{d}t}$ and $\frac{\mathrm{d}x}{\mathrm{d}t}$ represent the instantaneous growth rates of the two populations; t represents time and; α, β, γ and δ are positive real parameters describing the interaction of the two species.

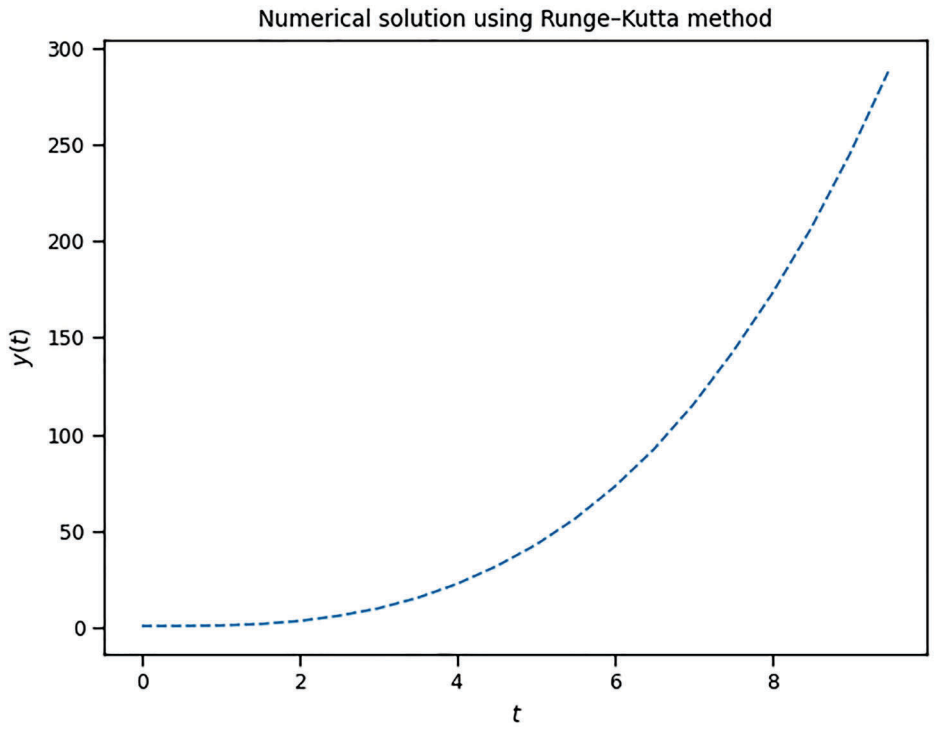

Figure 2. Numerical solution to the Initial Value Problem (1).

4.1. Using classical Runge–Kutta

To run this one will need the following packages as before including `numpy` for array operations, `matplotlib` for visualisations and `scipy` for getting a numerical solution with Runge–Kutta method. The first block of the code then becomes:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
```

For this module, we will use the following known values of α, β, γ and δ.

```
alpha = 2 / 3
beta = 4 / 3
gamma = 1
delta = 1
```

It is important to decide a time interval over which we will work. As an example let's consider between $t = 0$ and $t = 1$. As well as initial conditions $x(0)$ and $y(0)$.

```
t_initial = 0
t_final = 10

x0 = 1.2
y0 = 0.8
```

Now that students already have learned about Runge–Kutta, the instructors help them to apply the algorithm to the system of ODEs. Note we can modularise this step by creating a function, for instance, `runge_kutta` which takes as input an array of time steps, initial

conditions and the values of the parameters that describe the interaction between the two species.

```
def runge_kutta(
    t,
    x0,
    y0,
    alpha,
    beta,
    gamma,
    delta
):
    def func(t, Y):
         x, y = Y
         dx_dt = alpha * x - beta * x * y
         dy_dt = - gamma * y  + delta * x * y
         return dx_dt, dy_dt
    Y0 = [x0, y0]
    t_span = (t[0], t[-1])
    sol = solve_ivp(func, t_span, Y0, t_eval=t)
    x_true, y_true = sol.y
    return x_true, y_true
```

Next, we generate a time array and solve the initial value problem.

```
t_array = np.linspace(t_initial, t_final, 100)
x_rungekutta, y_rungekutta = runge_kutta(t_array, x0, y0, alpha, beta, gamma,
     delta)
```

Finally we have the students plot their results to get better insights of their simulations or predictions.

```
plt.plot(t_array, x_rungekutta, color="green", label=r"$x(t)$ Runge-Kutta")
plt.plot(t_array, y_rungekutta, color="blue", label=r"$y(t)$ Runge-Kutta")
plt.legend()
plt.xlabel(r"$t$")
plt.title("Lotka-Volterra numerical solution using Runge-Kutta method")
plt.show()
```

The result of the dynamics between the prey and predator is seen in Figure 3. The dynamics are as expected where each of the predator and prey populations cycle through time. As predators decrease numbers of prey, lack of food resources in turn decrease predator abundance, and the lack of predating pressure allows prey populations to rebound. And this cycle continues.

4.2. Using a physics-informed neural networks

For years mathematicians and physicists have been trying to model the world with differential equations. However, since the advent of techniques such as machine learning, neural networks, and deep learning, together with greater computing power, the community has speculated that we could learn automatically (algorithms) anything with enough amount of data. However, it seems this is not really true. Physics-Informed Neural Networks (PINNs) are a type of neural networks that are trained to solve supervised learning tasks while respecting any given law of physics described by general nonlinear partial differential equations (Raissi, Perdikaris, et al., 2019). This approach can approximate solutions by training a neural network to minimise a loss function, including:

- Initial and boundary conditions along the space-time domain's boundary

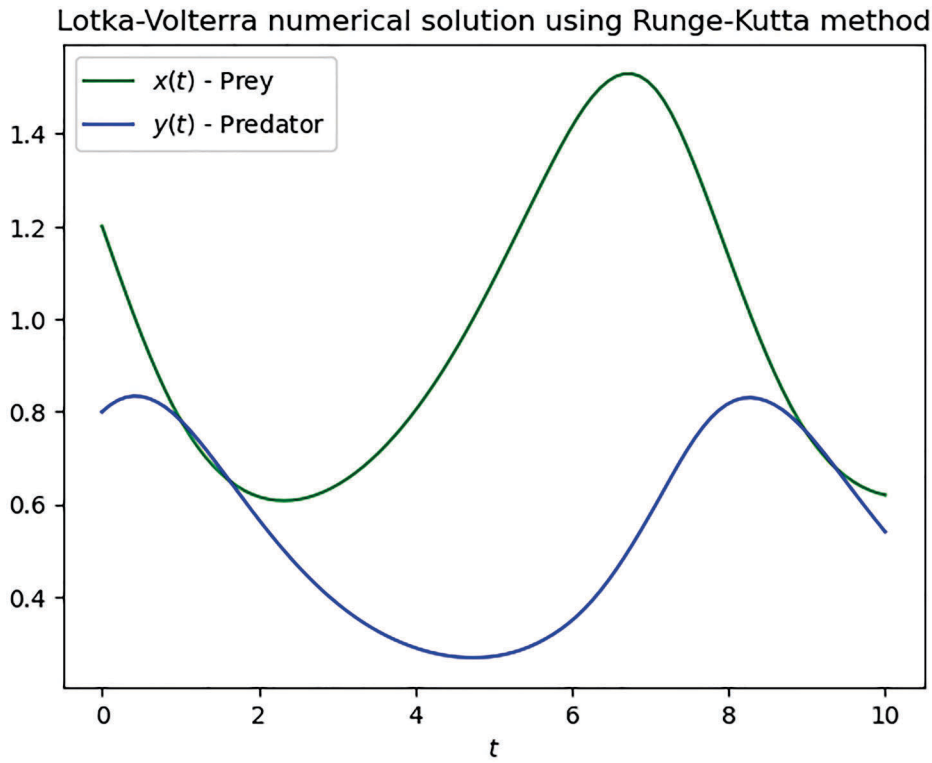

Figure 3. Numerical solution of system (2) using Runge-Kutta.

- Residual from governing Partial Differential Equations (PDE) at selected points in the domain.

A simplified analogy may be thought of as the initial and boundary conditions points being the training dataset. Along with this known information, one also needs to embed physical laws into the neural network. PINNs can solve differential equations expressed, in the most general form, like:

$$\begin{aligned} \mathcal{F}(u(z);\lambda) &= f(z) \quad z \text{ in } \Omega \\ \mathcal{B}(u(z)) &= g(z) \quad z \text{ in } \partial\Omega \end{aligned}$$

defined on the domain $\Omega \subset \mathbb{R}^d$ with the boundary $\partial\Omega$. Where

- $z := (x_1, x_2, \ldots, t)^\top$ indicated the space-time coordinate vector,
- u the unknown function,
- λ the parameters related to the physics,
- $\mathcal{F}$ the non-linear differential operator,
- f the function identifying the data of the problem,
- $\mathcal{B}$ the operator indicating arbitrary initial or boundary conditions, and
- g the boundary function.

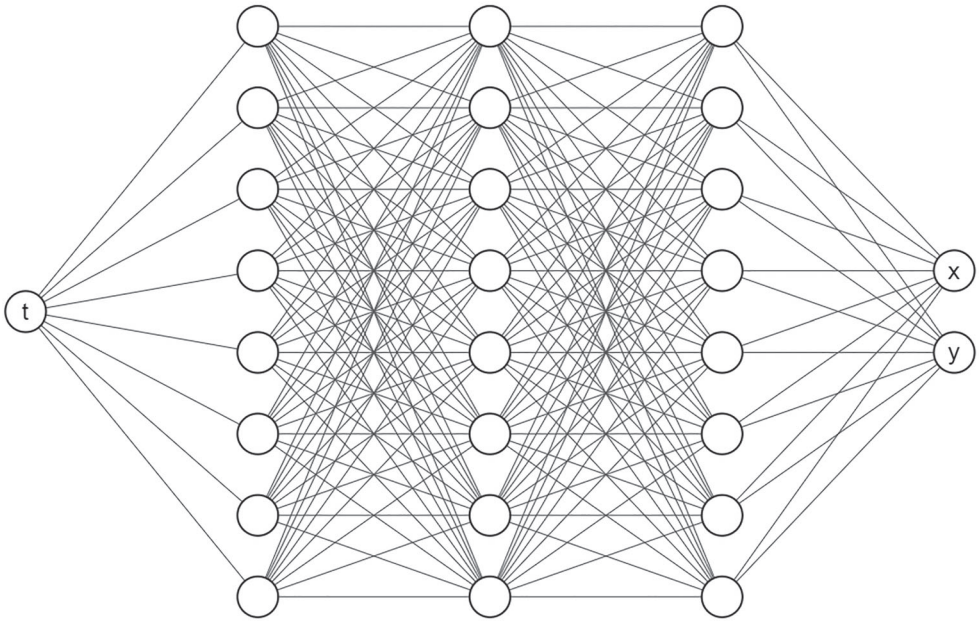

Figure 4. Neural Network architecture of Lotka-Volterra system (2).

In the PINNs methodology, $u(z)$ is computationally predicted by a neural network, parameterised by a set of parameters θ, giving rise to an approximation $\hat{u}_\theta(z) \approx u(z)$. The optimisation problem one has to solve then is given by:

$$\min_\theta \ (\omega_{\mathcal{F}}\mathcal{L}_{\mathcal{F}}(\theta) + \omega_{\mathcal{B}}\mathcal{L}_{\mathcal{B}}(\theta) + \omega_{\text{data}}\mathcal{L}_{\text{data}}(\theta))$$

where the three weighted loss functions in least-squared sense are defined by:

- $\mathcal{L}_{\mathcal{F}}$, differential equation,
- $\mathcal{L}_{\mathcal{B}}$, boundary conditions, and
- $\mathcal{L}_{\text{data}}$, (eventually) some known data.

Even though PINNs are a more recent technique there are several studies that have already applied the method to real-world applications. For example, fluids dynamics (Raissi et al., 2020), turbulent fields (Mathews et al., 20218), optics and electromagnetism (Chen et al., 2020Apr), molecular dynamics and materials (Fang & Zhan, 2020), geosciences (Smith et al., 202108), industrial applications (Yucesan & Viana, 2021) and infectious diseases (Shaier et al., 2022). Next, we will describe how one can implement PINNs to solve coupled system of governing differential equations instead of using the classical Fourth-order Runge–Kutta as before.

4.2.1. PINNs for solving ODEs

Now, let us revisit our Lotka-Volterra system where the solution u will be a vector such that $u(t) = (x(t), y(t))^\top$ and there are only initial conditions. We then want to train a network that looks like Figure 4.

Once a notebook is initiated, we start by including the package `deepxde`, which allows us to implement Physics-Informed Neural Networks approaches with just a few lines of code.

```
import deepxde as dde
from deepxde.backend import tf
```

If the users (instructor or students) are running this experiment on Google Colab (or similar) they will need to install `DeepXDE` before importing the packages by running the following extra line of code before the import statements.

```
!pip install deepxde
```

The reason behind this is because Google Colab has only common packages installed by default, for example `NumPy`, `Pandas`, `SciPy`, `TensorFlow`, `PyTorch`, etc. In order to take advantage of the resources Google Colab offers, we recommend running these codes using Graphical Processing Unit (GPU) hardware accelerator. This can be done by going to `Runtime > Change runtime type` and selecting `GPU` under the option of Hardware Accelerator.

Now, since we are trying to embed the physics onto the neural networks we need to define the system of ODEs as follows:

```
def ode(t, Y):
    x = Y[:, 0:1]
    y = Y[:, 1:2]

    dx_dt = dde.grad.jacobian(Y, t, i=0)
    dy_dt = dde.grad.jacobian(Y, t, i=1)

    return [
         dx_dt - alpha * x + beta * x * y,
         dy_dt + gamma * y  - delta * x * y
    ]
```

Here `t` is an array corresponding to the independent variable and `Y` is an array with two columns (since our system considers two equations). To define the first derivative, we use `dde.grad.jacobian`, with component `i=0` corresponding to the variable $x(t)$ and `i=1` to $y(t)$.

Next, we introduce the initial conditions by declaring this element for our neural network.

```
geom = dde.geometry.TimeDomain(t_initial, t_final)
```

Then we create a function for defining boundaries and since it is only one time we will use the default one as follows:

```
def boundary(_, on_initial):
    return on_initial
```

And then we have to define the initial conditions for the learning process:

```
ic_x = dde.icbc.IC(geom, lambda x: x0, boundary, component=0)
ic_y = dde.icbc.IC(geom, lambda x: y0, boundary, component=1)
```

Next, we define everything related to the differential equations and initial conditions as a new object `dde.data.PDE` as follows:

```
data = dde.data.PDE(
    geom,
```

```
    ode,
    [ic_x, ic_y],
    num_domain=512,
    num_boundary=2
)
```

Note that to test our model, we have considered 512 points inside our domain with `num_domain=512`. Finally, since we are working on a time domain there are only two points on its boundary (`num_boundary=2`) which are the initial and final times.

The next step is to choose the neural network architecture. For simplicity, we will use a fully-connected neural network (`dde.nn.FNN`). The most important things to remember when defining the neural network is that:

- Input layer (the first layer) needs only one node/neuron since our independent variable is only time *t*.
- The output layer (the last layer) needs two nodes/neurons since we are working with a system of two equations.

It should be noted that the students do not need to have much background on the amount of layers or neurons in each hidden layer. As a rule of thumb the error from the system should decrease as more layers and neurons are added, but adding more comes with more computational time. Other important items that the students must choose include the Activation functions and the initialiser. Usually `Glorot normal` works well as initialiser and the students can have the opportunity to consider different activation functions, for example, `relu`, `sigmoid` or `swish`.

```
neurons = 64
layers = 6
activation = "tanh"
initialiser = "Glorot normal"
net = dde.nn.FNN([1] + [neurons] * layers + [2], activation, initialiser)
```

Next, we create the model in a new object using the library with just one line of code.

```
model = dde.Model(data, net)
```

For training, one may choose the Adam optimiser (Bock & Weiß, 2019) and a learning rate of 0.001 (smaller learning rates may give you better results but it will take many more iterations). Just for simplicity, we will take 50,000 iterations. Another rule of thumb it is that as you increase the number of iterations the loss value should decrease as well.

```
model.compile("adam", lr=0.001)
losshistory, train_state = model.train(iterations=50000, display_every=10000)
```

Since neural networks rely on gradient descent algorithms, a good practice is to observe how the value of the loss function changes with each iteration. This can be plotted in a graph called *Loss History*, where the horizontal axis corresponds to iterations and the vertical axis to values of the loss function. See Figure 5 for the loss history of this particular example. This figure can be created by executing the following line

```
dde.utils.external.plot_loss_history(losshistory)
```

Notice in Figure 5 both train and test loss are the same since we are using the same points. There are scenarios where the user would like to test in a small dataset for avoiding overfitting or computational cost issues. This can be accomplished setting the argument

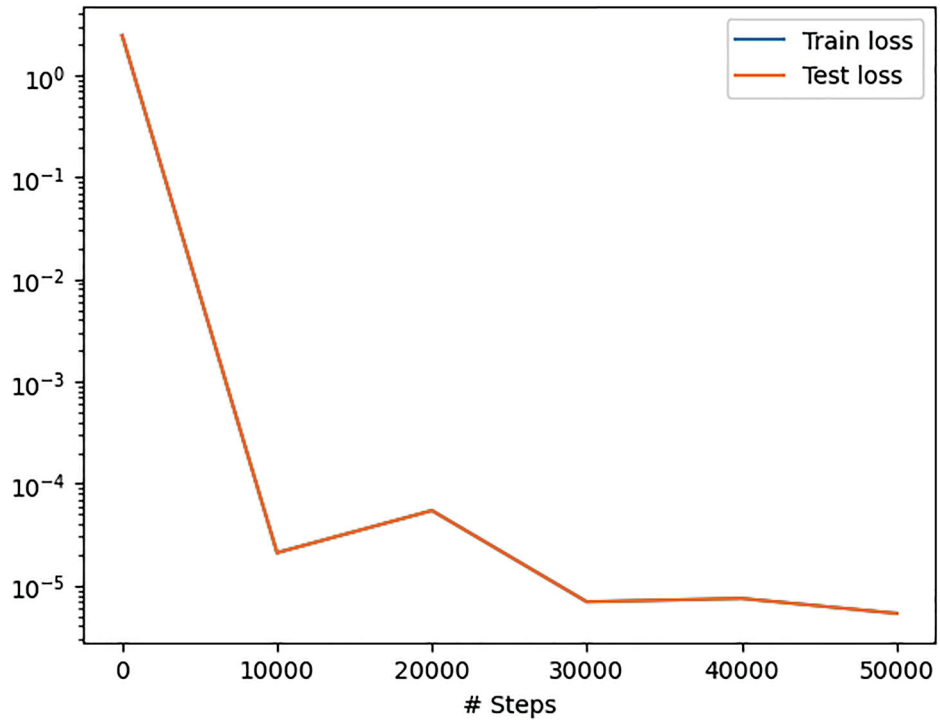

Figure 5. Loss history of PINNs approach for system (2).

`num_test` in the `dde.data.PDE` command. Since this example is working with a relative small dataset we considered it was not necessary to set this argument.

Once, we have the solution, we can then make predictions of the prey (x variable) and predator (y variable) species with PINNs as follows:

```
pinn_pred = model.predict(t_array.reshape(-1, 1))
x_pinn = pinn_pred[:, 0:1]
y_pinn = pinn_pred[:, 1:2]

plt.plot(t_array, x_pinn, color="green", label=r"$x(t)$ PINNs")
plt.plot(t_array, y_pinn, color="blue", label=r"$y(t)$ PINNs")
plt.legend()
plt.xlabel(r"$t$")
plt.title("Lotka-Volterra numerical solution using PINNs method")
plt.show()
```

As it may be noted from Figures 3 and 6, both the Runge–Kutta and PINNs algorithms gave us almost identical results. One of the pros that we would like to point out about PINNs is that for more complex systems one only needs to change a few items, specifically residuals. Most of the numerical work is done automatically by machine learning libraries such as `TensorFlow`, `Torch`, `JAX`, etc. so it is easy to scale up. The performance is even better when we can take advantage of GPUs. For students as users, they also will have the opportunity to play with the number of choices they have in picking suitable hyper-parameters (e.g. number of layers, number of neurons, activation function, number of iterations, etc.).

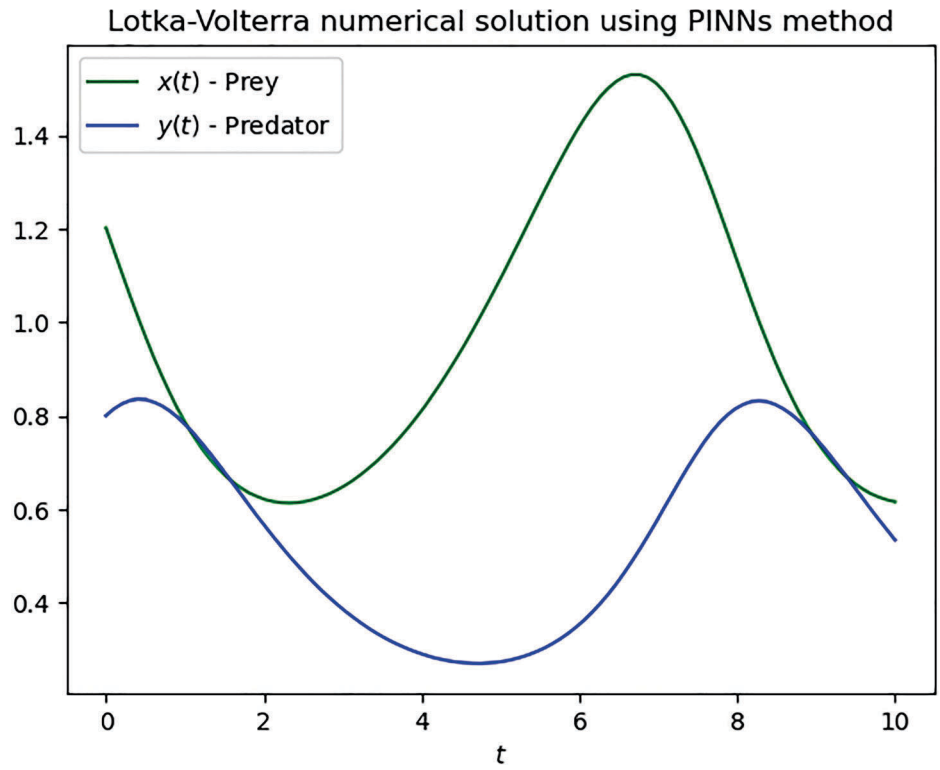

Figure 6. Numerical solution of system (2) using PINNs.

Finally, we can compare both models and notice they are practically the same curves (see Figure 7).

```
plt.plot(t_array, x_rungekutta, color="green", label=r"$x(t)$ Runge-Kutta")
plt.plot(t_array, y_rungekutta, color="blue", label=r"$y(t)$ Runge-Kutta")
plt.plot(t_array, x_pinn, color="red", linestyle="dashed", label=r"$x(t)$
    PINNs")
plt.plot(t_array, y_pinn, color="orange", linestyle="dashed", label=r"$y(t)$
    PINNs")
plt.legend()
plt.xlabel(r"$t$")
plt.title("Lotka-Volterra numerical solution comparison between Runge-Kutta
    and PINNs")
plt.show()
```

4.3. PINNs for parameter estimation

So far, we have considered solving the given system of differential equations in a forward fashion to determine the unknown dependent variables with initial conditions and parameter values in the equations provided. Now, let us consider another scenario. Imagine the students have access to sampled data of the two unknown variables $x(t)$ and $y(t$ every day. In other words, let us suppose that $x_{observed} = (x(t_1), x(t_2), \ldots, x(t_n))$ and $y_{observed} = (y(t_1), y(t_2), \ldots, y(t_n))$ where $t_1, t_2, \ldots, t_n$ are both provided as observed dataset over time. The goal here will then be to determine the optimal values α, β, γ and δ in the system of

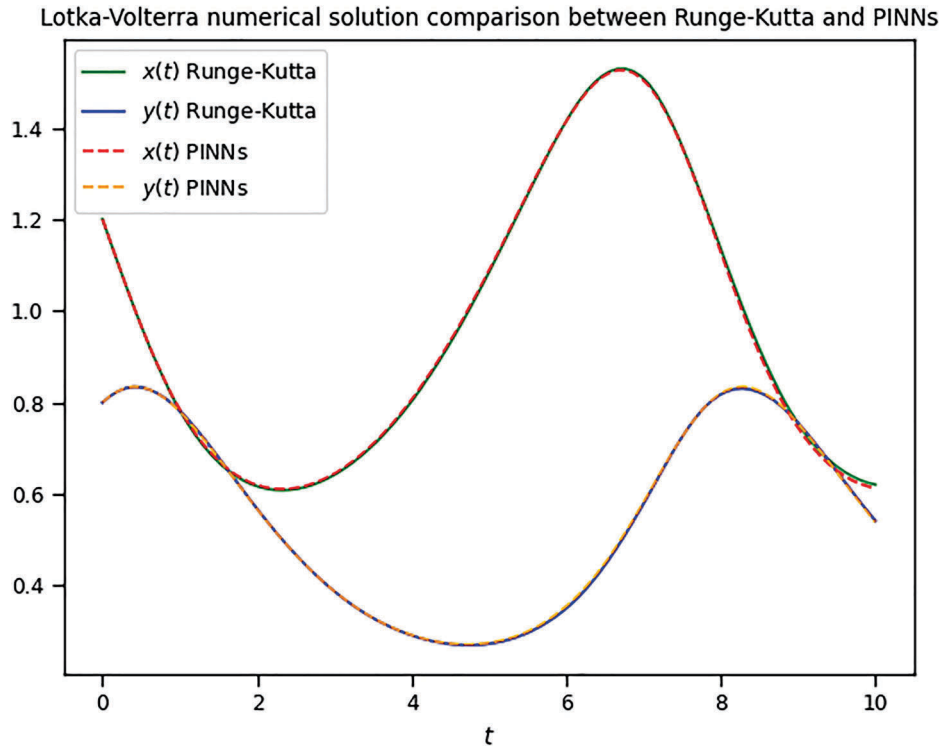

Figure 7. Comparison of numerical solution of system (2) using Runge–Kutta and PINNs.

differential equations (2) that correspond to this sampled data. This problem is referred to as an *Inverse Problem*, but one may also consider this to be a *Data-Driven Discovery*. A great example of this is when the CDC has to make public health decisions based on everyday infected data in real-time. We will see in this section, how one can use PINNs for this purpose in an inverse fashion.

As before, the first step would be to open a fresh notebook and start with a block that imports all the necessary libraries as described before.

```
import re
import numpy as np
import matplotlib.pyplot as plt
import deepxde as dde

from scipy.integrate import solve_ivp
from deepxde.backend import tf
```

The next step would be to train data that is prescribed. To help students understand the process of validation, one can employ the method of manufactured solutions. Specifically, one can generate synthetic data by choosing a time interval, initial conditions, and a known set of values of parameters. Then we feed that data (with some noise) back into a PINNs algorithm to check for robustness.

```
t_initial, t_final = 0, 10  # Equivalent to 10 days
x0 = 1.2
y0 = 0.8
```

```
alpha_real = 2 / 3
beta_real = 4 / 3
gamma_real = 1
delta_real = 1
parameters_real = {
    "alpha": alpha_real,
    "beta": beta_real,
    "gamma": gamma_real,
    "delta": delta_real
}  # We will use this later for studying errors
```

We can use a similar approach as in the last module for generating synthetic data that will allow us to study errors.

```
def generate_data(
    t,
    x0,
    y0,
    alpha,
    beta,
    gamma,
    delta
):
    def func(t, Y):
         x, y = Y
         dx_dt = alpha * x - beta * x * y
         dy_dt = - gamma * y  + delta * x * y
         return dx_dt, dy_dt

    Y0 = [x0, y0]
    t = t.flatten()
    t_span = (t[0], t[-1])
    sol = solve_ivp(func, t_span, Y0, t_eval=t)
    return sol.y.T
```

The synthetic data can then be generated and plotted (see Figure 8).

```
t_train = np.linspace(t_initial, t_final, 100).reshape(-1, 1)
Y_train = generate_data(t_train, x0, y0, alpha_real, beta_real, gamma_real,
    delta_real)

x_train = Y_train[:, 0:1]
y_train = Y_train[:, 1:2]
plt.scatter(t_train, x_train, color="green", s=3, label=r"$x(t)$ observed")
plt.scatter(t_train, y_train, color="blue", s=3, label=r"$y(t)$ observed")
plt.legend()
plt.title("Lotka-Volterra observed data")
plt.xlabel(r"$t$")
plt.show()
```

Since we are now solving an inverse problem, we will assume the real values of α, β, γ and δ are not known and the goal of this module is to learn how to estimate these parameters. Let's start defining them in a way that our code knows they have to be learned.

```
# Pick some initial guess
alpha = dde.Variable(0.0)
beta = dde.Variable(0.0)
gamma = dde.Variable(0.0)
delta = dde.Variable(0.0)
```

Now we have to define the residuals and initial conditions in the same way as the forward approximation (previous module).

```
def ode(t, Y):
```

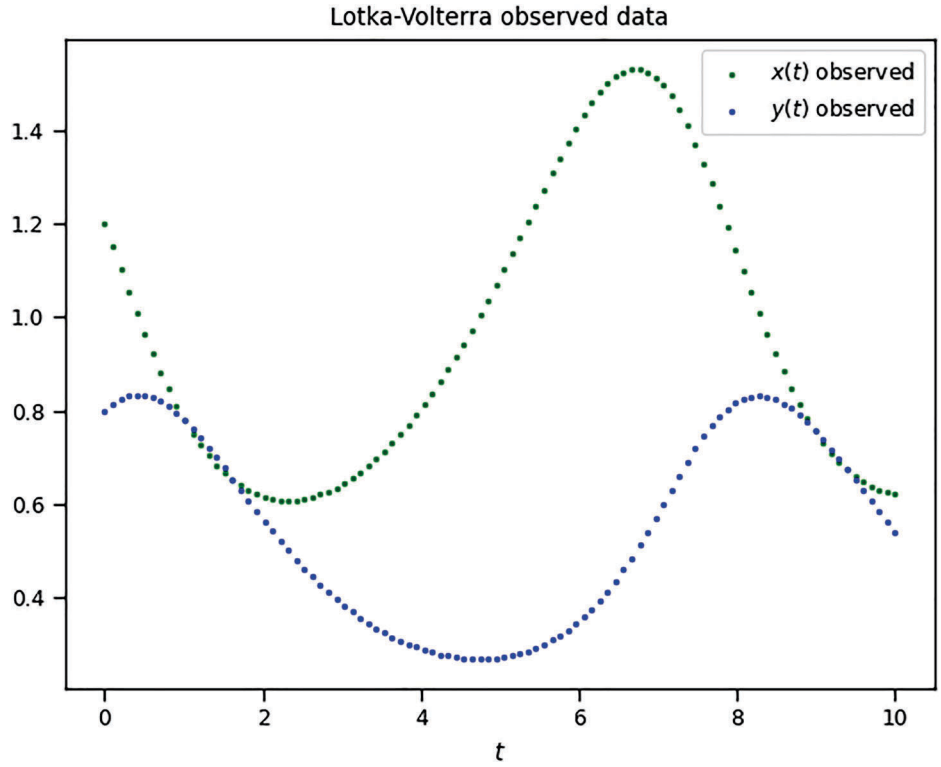

Figure 8. Synthetic data generated for system (2).

```
    x = Y[:, 0:1]
    y = Y[:, 1:2]
    dx_dt = dde.grad.jacobian(Y, t, i=0)
    dy_dt = dde.grad.jacobian(Y, t, i=1)

    return [
         dx_dt - alpha * x + beta * x * y,
         dy_dt + gamma * y  - delta * x * y
    ]
```

And also we need to define geometry and initial conditions as follows:

```
geom = dde.geometry.TimeDomain(t_initial, t_final)

def boundary(_, on_initial):
    return on_initial

ic_x = dde.icbc.IC(geom, lambda x: x0, boundary, component=0)
ic_y = dde.icbc.IC(geom, lambda x: y0, boundary, component=1)
```

We can use the observed data `t_train` and `Y_train`, which are equivalent to $x_{observed} = (x(t_1), x(t_2), \ldots, x(t_n))$ and $y_{observed} = (y(t_1), y(t_2), \ldots, y(t_n))$ for learning the values of our parameters. So we need to declare a new object and then include it in our model.

```
observe_x = dde.icbc.PointSetBC(t_train.reshape(-1, 1), Y_train[:, 0:1],
    component=0)
observe_y = dde.icbc.PointSetBC(t_train.reshape(-1, 1), Y_train[:, 1:2],
    component=1)
```

It is important for students to recall that `component=0` is associated with the variable x every time, defining the gradients, initial conditions, and observed data. Same for `component=1` which corresponds to the variable y. The data object is similar but we include the observed data as well in the list of conditions. Also note that we can add the observed data as training points with the argument `anchors`.

```
data = dde.data.PDE(
    geom,
    ode,
    [ic_x, ic_y, observe_x, observe_y],
    num_domain=512,
    num_boundary=2,
    anchors=t_train,
)
```

Next, we can define our neural network as follows:

```
neurons = 64
layers = 6
activation = "tanh"
\DIFdelbegin\DIFdel{initializer } \DIFdelend \DIFaddbegin \DIFadd{initialiser
     }\DIFaddend = "Glorot normal"
net = dde.nn.FNN([1] + [neurons] * layers + [2], activation, \DIFdelbegin\
    DIFdel{initializer} \DIFdelend \DIFaddbegin \DIFadd{initialiser}\
    DIFaddend)
```

The next step is also similar as before, except we need to tell our model it has to learn external variables (α, β, γ and δ) using `external_trainable_variables`.

```
model = dde.Model(data, net)
model.compile(
    "adam",
    lr=0.001,
    external_trainable_variables=[alpha, beta, gamma, delta]
)
```

In order to study the convergence of the learning process related to the parameters we want to estimate, we need a separate file where we can store these estimations. The file `variables.dat` will store our estimations of α, β, γ and δ every 100 iterations.

```
variable = dde.callbacks.VariableValue(
    [alpha, beta, gamma, delta],
    period=100,
    filename="variables.dat"
)
```

In the training process one must also remember to add a variable using 'callbacks=[variable]'. Figure 9 shows the loss history of PINNs approach when trying to estimate the parameters in the system.

```
losshistory, train_state = model.train(iterations=30000, display_every=5000,
    callbacks=[variable])
dde.utils.external.plot_loss_history(losshistory)
```

We can also plot the data as well, but it is not the main point of this module. This can now be seen in Figure 10.

```
plt.scatter(t_train, x_train, color="green", s=5, label="x_observed")
plt.scatter(t_train, y_train, color="blue", s=5, label="y_observed")

sol_pred = model.predict(t_train.reshape(-1, 1))
x_pred = sol_pred[:, 0:1]
```

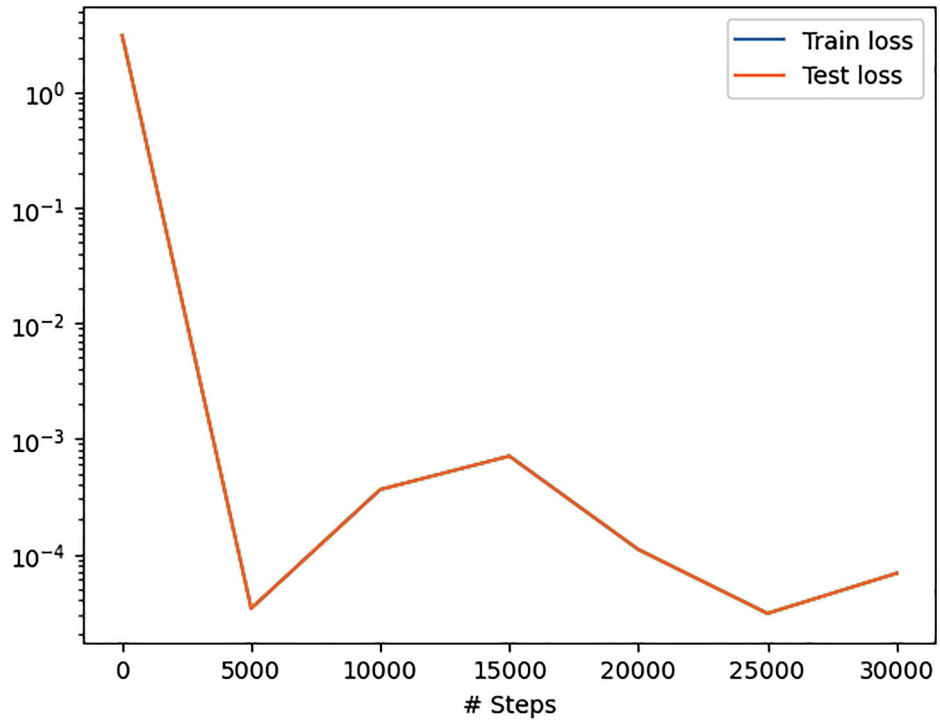

Figure 9. Loss history of PINNs approach for system (2) trying to estimate α, β, γ and δ.

```
y_pred = sol_pred[:, 1:2]

plt.plot(t_train, x_pred, color="red", linestyle="dashed", label=r"$x(t)$
    predicted")
plt.plot(t_train, y_pred, color="orange", linestyle="dashed", label=r"$y(t)$
    predicted")
plt.legend()
plt.title("Lotka-Volterra predicted and observed data")
plt.xlabel(r"$t$")
plt.show()
```

Next, we open the file `variables.dat` and create arrays for each parameter. The students need not worry about this next block of code as it will only return a dictionary where keys are the name of the parameters and values are their learning history (Figure 11).

```
lines = open("variables.dat", "r").readlines()
raw_parameters_pred_history = np.array(
    [
          np.fromstring(
             min(re.findall(re.escape("[") + "(.*?)" + re.escape("]"), line),
     key=len),
             sep=",",
         )
         for line in lines
    ]
)
iterations = [int(re.findall("^[0-9]+", line)[0]) for line in lines]

parameters_pred_history = {
    name: raw_parameters_pred_history[:, i]
    for i, name in enumerate(parameters_real.keys())
```

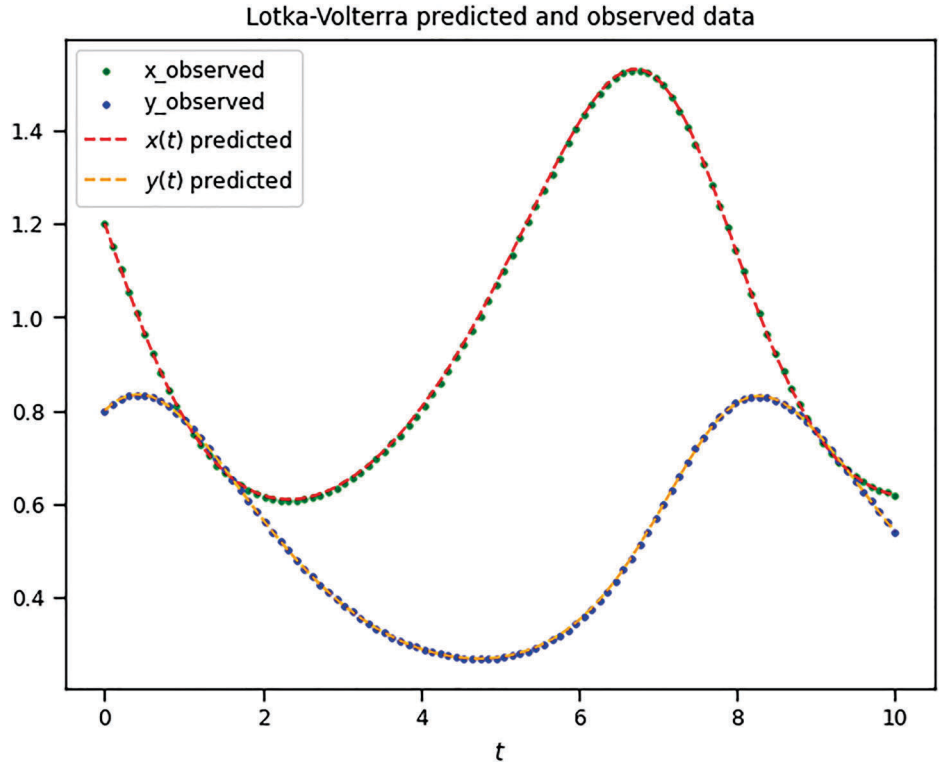

Figure 10. Observed data and PINNs prediction for system (2) trying to estimate α, β, γ and δ.

```
}
n_callbacis, n_variables = raw_parameters_pred_history.shape
fig, axes = plt.subplots(nrows=n_variables, sharex=True)
for ax, (parameter, parameter_value) in zip(axes, parameters_real.items()):
    ax.plot(iterations, parameters_pred_history[parameter] , "-")
    ax.plot(iterations, np.ones_like(iterations) * parameter_value, "--")
    ax.set_ylabel(parameter)
ax.set_xlabel("Iterations")
fig.suptitle("Parameter estimation")
fig.tight_layout()
```

Note that within 5000 iterations one can get a really good approximation for each parameter. Next, we can calculate the relative error for each parameter as well.

```
alpha_pred, beta_pred, gamma_pred, delta_pred = variable.value
print(f"alpha - real: {alpha_real:4f} - predicted: {alpha_pred:4f} - relative
     error: {np.abs((alpha_real - alpha_pred) / alpha_real):4f}")
print(f"beta - real: {beta_real:4f} - predicted: {beta_pred:4f} - relative
    error: {np.abs((beta_real - beta_pred) / beta_real):4f}")
print(f"gamma - real: {gamma_real:4f} - predicted: {gamma_pred:4f} - relative
     error: {np.abs((gamma_real - gamma_pred) / gamma_real):4f}")
print(f"delta - real: {delta_real:4f} - predicted: {delta_pred:4f} - relative
     error: {np.abs((delta_real - delta_pred) / delta_real):4f}")
```

```
alpha - real: 0.666667 - predicted: 0.665105 - relative error: 0.002342
beta - real: 1.333333 - predicted: 1.330125 - relative error: 0.002406
gamma - real: 1.000000 - predicted: 0.999372 - relative error: 0.000628
delta - real: 1.000000 - predicted: 0.998206 - relative error: 0.001794
```

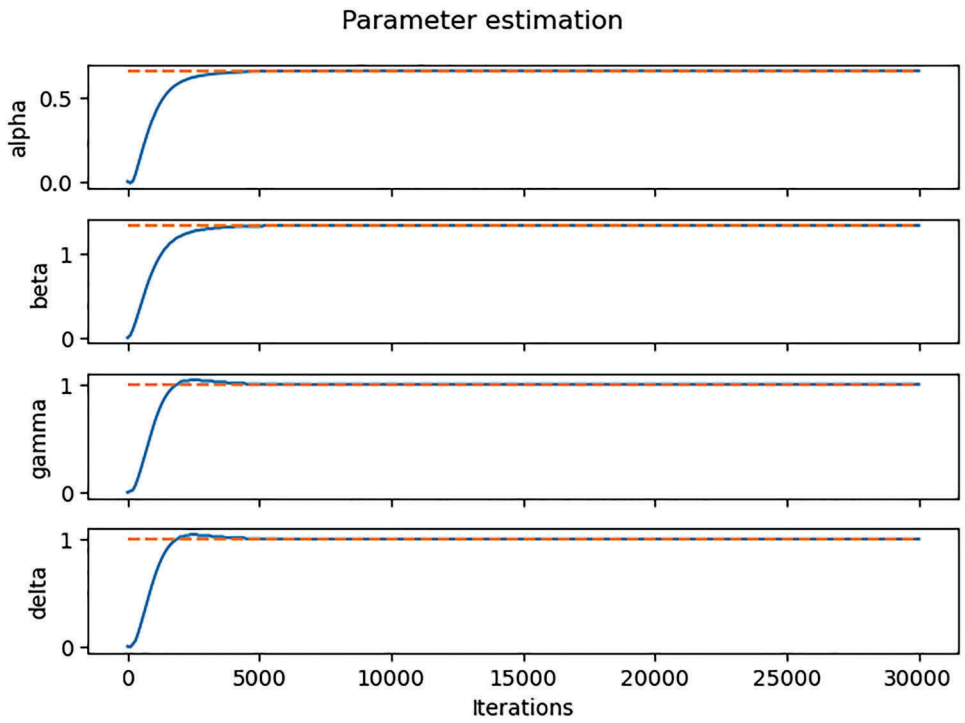

Figure 11. Learning history of parameter estimation of system (2) using PINNs.

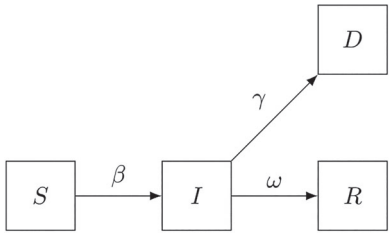

Figure 12. SIRD compartmental model.

5. Module-3: application to disease dynamics

In this section, students have an opportunity to work with datasets obtained for understanding disease dynamics. Specifically, we consider the following differential equation system (for simplicity) described by the SIRD system (Anastassopoulou et al., 2020). SIRD compartmental models are mathematical models used to describe the spread of infectious diseases in a population. The name *SIRD* stands for *Susceptible*, *Infected*, *Recovered*, and *Dead*, which are the four compartments (see Figure 12) that individuals can be classified into based on their disease status. In this model, individuals can move from one compartment to another over time, based on the parameters of the model. Specifically, individuals can move from the susceptible compartment to the infected compartment if

they are exposed to the disease, from the infected compartment to the recovered compartment if they recover from the disease, and from the infected compartment to the dead compartment if they die from the disease. The SIRD model assumes that once an individual recovers from the disease, they are immune to future infections, and that the rate of transmission of the disease is proportional to the number of susceptible individuals and the number of infected individuals in the population. These models are used to predict the spread and impact of infectious diseases and can be used to evaluate different intervention strategies, such as vaccination programmes and social distancing measures, to control the spread of the disease. This model leads to the following equations

$$\begin{aligned} \frac{\mathrm{d}S}{\mathrm{d}t} &= -\frac{\beta S}{N} I \\ \frac{\mathrm{d}I}{\mathrm{d}t} &= \frac{\beta S}{N} I - \omega I - \gamma I \\ \frac{\mathrm{d}R}{\mathrm{d}t} &= \omega I \\ \frac{\mathrm{d}D}{\mathrm{d}t} &= \gamma I \end{aligned} \tag{3}$$

where $S(t)$, $I(t)$, $R(t)$, $D(t)$ are the number of *Susceptible*, *Infected*, *Recovered*, and *Dead* individuals, respectively. Here β, ω and γ are the rates of transmission, recovery, and mortality, respectively. In order to get a good insight into the spread of an infectious disease, it is important to be able to estimate these dynamic parameters using data.

As a first step, we import all the libraries needed to employ numerical methods for solving differential equations and for using PINNs as before.

```
import re
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import deepxde as dde

from deepxde.backend import tf
from scipy.integrate import solve_ivp

sns.set_theme(style="darkgrid")
```

Again, to validate the code, we will manufacture a solution first for a prescribed set of parameters which will serve as synthetic data. Consider a population of 10.000.000 people and only one person infected.

```
N = 1e7
S_0 = N - 1
I_0 = 1
R_0 = 0
D_0 = 0
y0 = [S_0, I_0, R_0, D_0]  # Initial conditions vector

beta = 0.5
omega = 1 / 14
gamma = 0.1 / 14

parameters_real = {
    "beta": beta,
```

```
    "omega": omega,
    "gamma": gamma,
}
```

We will use the same approach as the last module using a numerical solver in order to create the data.

```
def generate_data(
    t_array,
    y0,
):
    def func(t, y):
         S, I, R, D = y
         dS_dt = - beta * S / N * I
         dI_dt = beta * S / N * I - omega * I - gamma * I
         dR_dt = omega * I
         dD_dt = gamma * I

         return np.array([dS_dt, dI_dt, dR_dt, dD_dt])

    t_span = (t_array[0], t_array[-1])
    sol = solve_ivp(func, t_span, y0, t_eval=t_array)
    return sol.y.T
```

For illustration purposes, for this example, we will take a time span of four months, which is approximately 120 days. The output is an array of 4 columns and 120 rows.

```
n_days = 120  # 4 months
t_train = np.arange(0, n_days, 1)[:, np.newaxis]
y_train = generate_data(np.ravel(t_train), y0)
y_train.shape
```

Now we can explore the simulated data. For this we will include some Python libraries including `pandas` (`pd`) that are often used for working with data sets as needed for analysing, cleaning, exploring, and manipulating data. Similarly we also include `seaborn` (`sns`) for getting better plots, respectively.

Note that `pandas` allows us to get a more general structure for data analysis, `pd.DataFrame`, for instance, we can give each column a name, *S, I, R, D*, respectively. And `seaborn` is a plotting tool built on top of `matplotlib`. It works better with 'pandas' and one can get really good results with a few lines of code.

```
model_name = "SIRD"
populations_names = list(model_name)
data_real = (
         pd.DataFrame(y_train, columns=populations_names)
         .assign(time=t_train)
         .melt(id_vars="time", var_name="status", value_name="population")
)

fig, ax = plt.subplots(figsize=(10, 4))
sns.lineplot(
    data=data_real,
    x="time",
    y="population",
    hue="status",
    legend=True,
    linestyle="dashed",
    ax=ax
)

ax.set_title(f"{model_name} model - Training Data")
fig.show()
```

Figure 13. Synthetic data generated for system (3).

Figure 13 plots the dynamics of each of the variables in the SIRD model which reflects the typical behaviour that is often observed.

5.1. Diseases informed neural networks

Now that the students have been exposed to PINNs we help them to explore the application of PINNs to understand disease dynamics by estimating parameters in the SIRD model. The students can refer back to the predator-prey model parameter estimation approach and extend that for the SIRD model.

As a first step, we start with parameter prediction and we define an initial guess for those parameters we want to estimate.

```
_beta = dde.Variable(0.0)
_omega = dde.Variable(0.0)
_gamma = dde.Variable(0.0)
```

The next step is to describe the residuals for the SIRD system which are given by:

$$\mathcal{L}_{\mathcal{F}_S} = \frac{\mathrm{d}S}{\mathrm{d}t} - \left(-\frac{\beta S}{N} I\right)$$

$$\mathcal{L}_{\mathcal{F}_I} = \frac{\mathrm{d}I}{\mathrm{d}t} - \left(\frac{\beta S}{N} I - \omega I - \gamma I\right)$$

$$\mathcal{L}_{\mathcal{F}_R} = \frac{\mathrm{d}R}{\mathrm{d}t} - \omega I$$

$$\mathcal{L}_{\mathcal{F}_D} = \frac{\mathrm{d}D}{\mathrm{d}t} - \gamma I$$

```
def ode(t, y):

    S = y[:, 0:1]
    I = y[:, 1:2]
    R = y[:, 2:3]
    D = y[:, 3:4]
```

```
    dS_dt = dde.grad.jacobian(y, t, i=0)
    dI_dt = dde.grad.jacobian(y, t, i=1)
    dR_dt = dde.grad.jacobian(y, t, i=2)
    dD_dt = dde.grad.jacobian(y, t, i=3)

    return [
         dS_dt - ( - _beta * S / N * I ),
         dI_dt - ( _beta * S / N * I - _omega * I - _gamma * I  ),
         dR_dt - ( _omega * I ),
         dD_dt - ( _gamma * I )
    ]
```

Next, we introduce the relevant geometry, boundary conditions (if any), and initial conditions.

```
# Geometry
geom = dde.geometry.TimeDomain(t_train[0, 0], t_train[-1, 0])


# Boundaries
def boundary(_, on_initial):
    return on_initial

# Initial conditions
S_0, I_0, R_0, D_0 = y_train[0, :]
ic_S = dde.icbc.IC(geom, lambda x: S_0, boundary, component=0)
ic_I = dde.icbc.IC(geom, lambda x: I_0, boundary, component=1)
ic_R = dde.icbc.IC(geom, lambda x: R_0, boundary, component=2)
ic_D = dde.icbc.IC(geom, lambda x: D_0, boundary, component=3)
```

The next step is to record the observed data for all four variables in the SIRD model.

```
observed_S = dde.icbc.PointSetBC(t_train, y_train[:, 0:1], component=0)
observed_I = dde.icbc.PointSetBC(t_train, y_train[:, 1:2], component=1)
observed_R = dde.icbc.PointSetBC(t_train, y_train[:, 2:3], component=2)
observed_D = dde.icbc.PointSetBC(t_train, y_train[:, 3:4], component=3)
```

Next, we describe the data for the model.

```
data = dde.data.PDE(
    geom,
    ode,
    [
         ic_S,
         ic_I,
         ic_R,
         ic_D,
         observed_S,
         observed_I,
         observed_R,
         observed_D,
    ],
    num_domain=256,
    num_boundary=2,
    anchors=t_train,
)
```

Now we define the neural network as the next step. Here we need to define the structure of the network. Once again, there is just one input (time) but there are four neurons in the last layer since we are working on a system of four equations. A neural network architecture is shown in Figure 14.

```
neurons = 64
layers = 3
```

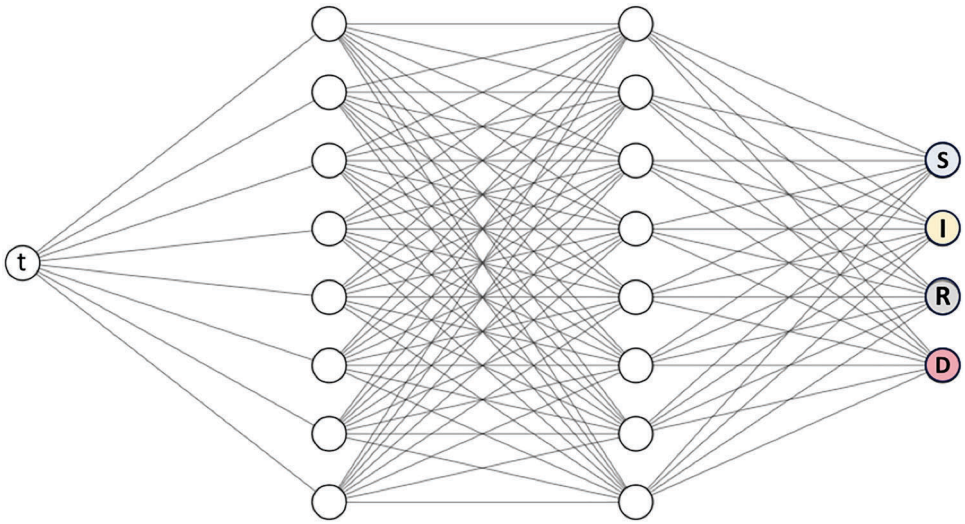

Figure 14. Neural Network architecture of SIRD system (3).

```
activation = "relu"
net = dde.nn.FNN([1] + [neurons] * layers + [4], activation, "Glorot uniform"
    )
```

The next step is the training part.

```
variable_filename = "sird_variables.dat"
variable = dde.callbacks.VariableValue(
    [_beta, _omega, _gamma],
    period=100,
    filename=variable_filename
)
```

```
model = dde.Model(data, net)
model.compile(
    "adam",
    lr=1e-3,
    external_trainable_variables=[_beta, _omega, _gamma]
)
losshistory, train_state = model.train(
    iterations=30000,
    display_every=5000,
    callbacks=[variable]
)
dde.saveplot(losshistory, train_state, issave=False, isplot=True)
```

We can plot training and predicted data together (see Figure 15). Notice it is almost a perfect fit, which makes sense, since we are predicting data for which we already know the parameter values.

```
t_pred =  np.arange(0, n_days, 1)[:, np.newaxis]
y_pred = model.predict(t_pred)
data_pred = (
    pd.DataFrame(y_pred, columns=populations_names, index=t_pred.ravel())
    .rename_axis("time")
    .reset_index()
    .melt(id_vars="time", var_name="status", value_name="population")
)
```

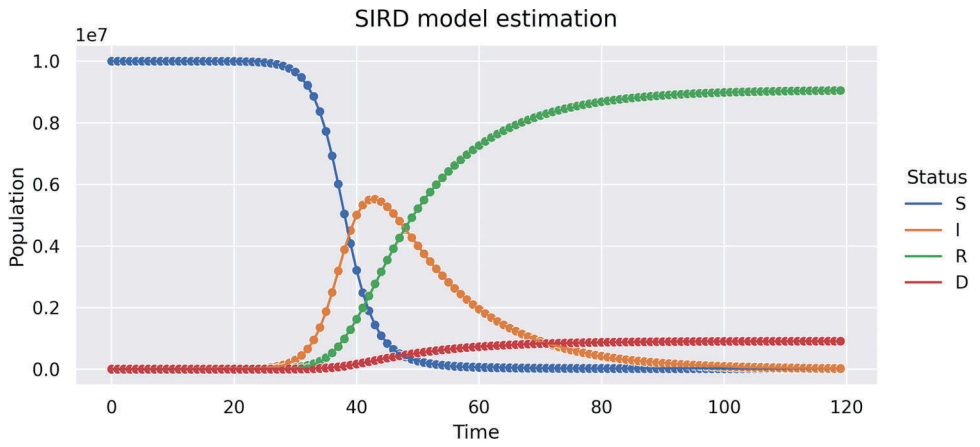

Figure 15. Observed data and PINNs prediction for system (3) for parameter estimation.

```
g = sns.relplot(
    data=data_pred,
    x="time",
    y="population",
    hue="status",
    kind="line",
    aspect=2,
    height=4
)

sns.scatterplot(
    data=data_real,
    x="time",
    y="population",
    hue="status",
    ax=g.ax,
    legend=False
)
(
    g.set_axis_labels("Time", "Population")
    .tight_layout(w_pad=1)
)

g._legend.set_title("Status")
g.fig.subplots_adjust(top=0.9)
g.fig.suptitle(f"SIRD model estimation")

plt.savefig("sird_prediction.png", dpi=300)
plt.show()
```

To find the robustness of the algorithm, we next study how well this neural network learned the parameters of our infectious disease system. The model predictions are illustrated in Figure 15.

```
lines = open(variable_filename, "r").readlines()
raw_parameters_pred_history = np.array(
    [
          np.fromstring(
             min(re.findall(re.escape("[") + "(.*?)" + re.escape("]"), line),
     key=len),
             sep=",",
         )
```

Table 2. Parameter predictions and relative errors for SIRD model using DINNs.

	Real	Predicted	Rel. Error
β	.5000	.4995	.0010
ω	.0714	.0711	.0047
γ	.0071	.0071	.0092

```
          for line in lines
    ]
)

iterations = [int(re.findall("^[0-9]+", line)[0]) for line in lines]

parameters_pred_history = {
    name: raw_parameters_pred_history[:, i]
    for i, (name, nominal) in enumerate(parameters_real.items())
}

n_callbacis, n_variables = raw_parameters_pred_history.shape
fig, axes = plt.subplots(nrows=n_variables, sharex=True, figsize=(6, 5),
    layout="constrained")
for ax, (parameter, parameter_value) in zip(axes, parameters_real.items()):
    ax.plot(iterations, parameters_pred_history[parameter] , "-")
    ax.plot(iterations, np.ones_like(iterations) * parameter_value, "--")
    ax.set_ylabel(parameter)
ax.set_xlabel("Iterations")
fig.suptitle("Parameter estimation")
fig.tight_layout()
fig.savefig("sird_parameter_estimation.png", dpi=300)
```

Finally, we can compute the relative errors, which are in Table 2.

```
parameters_pred = {
    name: var for name, var in zip(parameters_real.keys(), variable.value)
}
error_df = (
    pd.DataFrame(
          {
              "Real": parameters_real,
              "Predicted": parameters_pred
          }
    )
    .assign(
          **{"Relative Error": lambda x: (x["Real"] - x["Predicted"]).abs() /
    x["Real"]}
    )
)
error_df
```

Also, students will be able to observe that within a few iterations (neural networks in industry can take millions and billions of iterations) we were able to obtain results within a 0.01 relative error. This predictions can be bettered by hyper-tuning the algorithm's parameters; however, that will require more advanced study (Figure 16).

6. Conclusion and future work

In this work, we introduce a literate programming style to teach the numerical solution to differential equations. With a goal to engage students with technology to learn more about applications of differential equations, we introduce neural networks as a potential approach to advance their learning. The paper also presents some of the important software tools

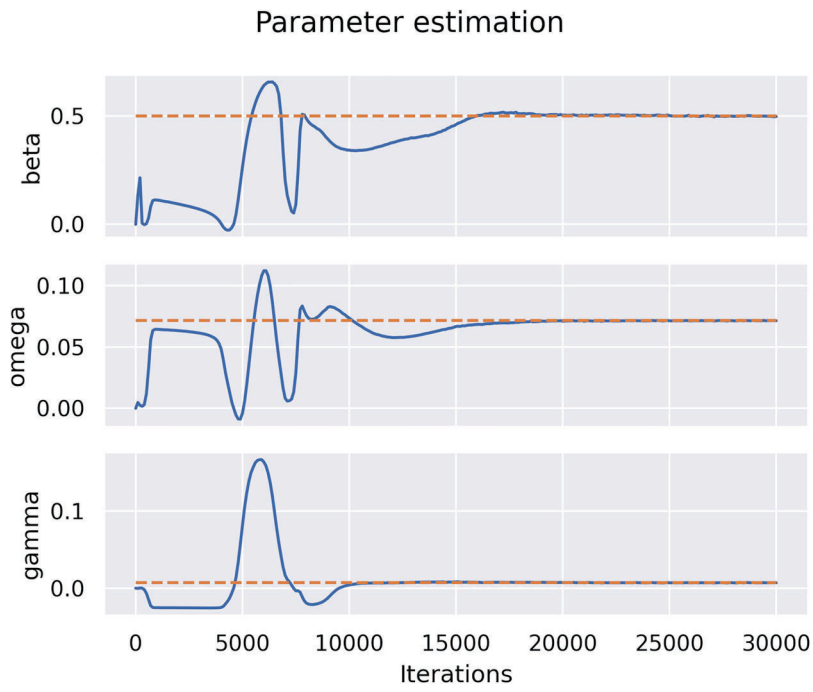

Figure 16. Learning history of parameter estimation of system (3) using PINNs.

including programming languages, notebooks and storage repositories that can be used in combination by instructors to enhance their own pedagogical practices when teaching differential equations.

Applications of the Physics Informed Neural Networks (PINNs) were introduced and shown as a potential alternative to traditional numerical approaches used to solve ODEs such as the classical Runge–Kutta methods. The power of PINNs to also be able to solve for the parameters in the models in an inverse fashion was also introduced and applied to some benchmark problems. It is also important to highlight some elements inherent to machine learning approaches, such as initial guesses or machine architectures. For example, using Graphical Processor Units (GPU) for PINNs will substantially increase the speed of the algorithm. On the other hand, it is very likely when students run their models they will notice prediction or estimation values are not exactly the same between each other, however, these differences will not be big enough for discarding this methodology. In fact, the results shown in the paper validate that PINNs are not only a viable candidate, but also are a robust and reliable algorithm for both solving differential equations numerically in both a forward and inverse fashion.

As a next step, we hope to study the impact of such instructional approaches on enhancing the learning of students as well as faculty mindset change in using such approaches to enhance their own pedagogical practices. Based on a few presentations and workshops the authors have given to faculty and students at various conferences and programmes, the response has been positive in accepting the approach presented in this work. We hope to do more rigorous quasi-experimental studies to see longitudinal impact of changes in instructional practices and student learning in a forthcoming paper. We are also planning

to use this material for bootcamps that are associated with projects related to modelling, analysis and simulation of infectious diseases.

The resources linked to this paper can be accessed from https://aoguedao.github.io/teaching-ml-diffeq. A separate supplemental resource file has also been added that includes a guide for instructors to access all the relevant Python codes.

Acknowledgments

The authors are also very grateful to the anonymous reviewers whose feedback was very useful.

Disclosure statement

No potential conflict of interest was reported by the authors.

Funding

This work is partially supported by the National Science Foundation [grant numbers DMS-2031029 and DMS-2230117].

References

Ackleh, A. S., Allen, E. J., Kearfott, R. B., & Seshaiyer, P. (2009). *Classical and modern numerical analysis: Theory, methods and practice*. CRC Press.

Anastassopoulou, C., Russo, L., Tsakris, A., Siettos, C., & Othumpangat, S. (2020). Data-based analysis, modelling and forecasting of the COVID-19 outbreak. *PloS One*, *15*(3), e0230405. https://doi.org/10.1371/journal.pone.0230405

Bock, S., & Weiß, M. (2019). A proof of local convergence for the Adam optimizer. In *2019 International Joint Conference on Neural Networks (IJCNN)* (pp. 1–8). IEEE.

Chen, Y., Lu, L., Karniadakis, G. E., & Negro, L. D. (2020, April). Physics-informed neural networks for inverse problems in nano-optics and metamaterials. *Optics Express*, *28*(8), 11618–11633. https://doi.org/10.1364/OE.384875

Ding, W., Florida, R., Summers, J., Nepal, P., & Burton, B. (2019). Experience and lessons learned from using SIMIODE modeling scenarios. *PRIMUS*, *29*(6), 571–583. https://doi.org/10.1080/10511970.2018.1488318

Fang, Z., & Zhan, J. (2020). Deep physical informed neural networks for metamaterial design. *IEEE Access*, *8*, 24506–24513. https://doi.org/10.1109/ACCESS.2019.2963375

Knuth, D. E. (1984). Literate programming. *The Computer Journal*, *27*(2), 97–111. https://doi.org/10.1093/comjnl/27.2.97

Mathews, A., Francisquez, M., Hughes, J. W., Hatch, D. R., Zhu, B., & Rogers, B. N. (2021, August). Uncovering turbulent plasma dynamics via deep learning from partial observations. *Physical Review E*, *104*(2), 025205. https://doi.org/10.1103/physreve.104.025205

McCarthy, C., Swanson, E., & Winkel, B. (2019). Special issue of primus: Modeling approach to teaching differential equations. *PRIMUS*, *29*(6), 503–508. Taylor & Francis. https://doi.org/10.1080/10511970.2019.1565789

Nedialkov, N. S. (2011). Implementing a rigorous ODE solver through literate programming. In *Modeling, design, and simulation of systems with uncertainties* (pp. 3–19). Springer.

Raissi, M., Perdikaris, P., & Karniadakis, G. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, *378*, 686–707. https://doi.org/10.1016/j.jcp.2018.10.045

Raissi, M., Ramezani, N., & Seshaiyer, P. (2019). On parameter estimation approaches for predicting disease transmission through optimization, deep learning and statistical inference methods. *Letters in Biomathematics*, *6*(2), 1–26. https://doi.org/10.30707/LiB

Raissi, M., Yazdani, A., & Karniadakis, G. E. (2020). Hidden fluid mechanics: Learning velocity and pressure fields from flow visualizations. *Science*, *367*(6481), 1026–1030. https://doi.org/10.1126/science.aaw4741

Seshaiyer, P. (2017). Leading undergraduate research projects in mathematical modeling. *PRIMUS*, *27*(4–5), 476–493. https://doi.org/10.1080/10511970.2016.1240732

Seshaiyer, P., & Solin, P. (2017). Enhancing student learning of differential equations through technology. *International Journal for Technology in Mathematics Education*, *24*(4), 207–215. https://doi.org/10.1564/tme_v24.4.05

Shaier, S., Raissi, M., & Seshaiyer, P. (2022). Data-driven approaches for predicting spread of infectious diseases through DINNs: Disease informed neural networks. *Letters in Biomathematics*, *9*(1), 71–105.

Simmons, G. F. (2016). *Differential equations with applications and historical notes*. CRC Press.

Smith, J. D., Ross, Z. E., Azizzadenesheli, K., & Muir, J. B. (202108). HypoSVI: Hypocentre inversion with Stein variational inference and physics informed neural networks. *Geophysical Journal International*, *228*(1), 698–710. https://doi.org/10.1093/gji/ggab309

Yucesan, Y. A., & Viana, F. A. (2021). Hybrid physics-informed neural networks for main bearing fatigue prognosis with visual grease inspection. *Computers in Industry*, *125*, 103386. https://doi.org/10.1016/j.compind.2020.103386