



HELMUT SCHMIDT  
UNIVERSITÄT

Universität der Bundeswehr Hamburg



# Physics-Informed Neural Networks

Tutorial@MAGNET4Cardiac7T

# Availability of the Materials

---

All material can be found at

[github.com/CeciliaCoelho/PINNsTutorialMAGNET4Cardiac7T](https://github.com/CeciliaCoelho/PINNsTutorialMAGNET4Cardiac7T), including the slides, hands-on exercises and a jupyter notebook with the coding examples and hands-on that will be used.

# Introduction to Differential Equations

---

- What are Differential Equations
- Solving Differential Equations
- Differential Equations in Real life

# What are Differential Equations

---

Let's look at a simple, classic example of building a mathematical model to describe the evolution over time of a certain population (human, animal species, bacterial, etc.). Suppose that the number of individuals in a certain population at a given time  $t$  is given by

$$P(t) \tag{1}$$

, that is,  $P(t)$  represents a function of time (the population as a function of time). To develop the model, let's fix a certain time interval

$$[t, t + \Delta t]. \tag{2}$$

For example, if  $t = 1\text{ hour}$  and  $\Delta t = 4\text{ hours}$ , the interval would be  $[1; 5]$ . We will try to find a relationship between the population at time  $t$  ( $P(t)$ ) and the population at time  $t + \Delta t$  ( $P(t + \Delta t)$ ).

# What are Differential Equations

What happens during this time interval is that there was a certain number of deaths ( $M$ ) and a certain number of births ( $N$ ).

Our common sense suggests that:

- The larger the population, the higher the number of deaths and births (e.g., if in a population of 1000 individuals we have 10 deaths, then in a population of 2000 individuals we would expect 20 deaths). In other words, the number of deaths ( $M$ ) and births ( $N$ ) is proportional to the number of individuals ( $P(t)$ ):

$$N = \alpha P(t), \quad M = \beta P(t) \tag{3}$$

where  $\alpha$  and  $\beta$  ( $\in \mathbb{R}^+$ ) are the proportionality constants, which may vary from population to population.

- It is also expected that  $N$  and  $M$  depend on the time interval  $\Delta t$ . That is, the longer the time interval, the greater the number of births and deaths.

# What are Differential Equations

---

This double dependency of  $N$  and  $M$  on  $P(t)$  and  $\Delta t$  can then be expressed as:

$$N = \alpha P(t) \Delta t, \quad M = \beta P(t) \Delta t. \quad (4)$$

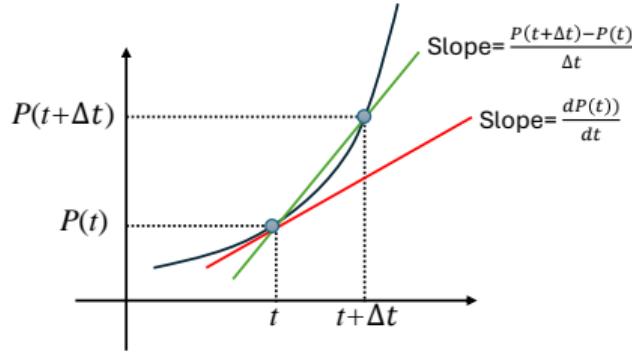
Consequently, the change in population over the time interval  $[t, t + \Delta t]$  is given by  $P(t + \Delta t) - P(t) = N - M$ , that is:

$$P(t + \Delta t) - P(t) = (\alpha - \beta) P(t) \Delta t. \quad (5)$$

Dividing both sides by  $\Delta t$ , we obtain the following equation:

$$\frac{P(t + \Delta t) - P(t)}{\Delta t} = (\alpha - \beta) P(t). \quad (6)$$

# What are Differential Equations



Taking the limit as  $\Delta t \rightarrow 0$ , we obtain the following differential equation:

$$\frac{dP}{dt} = (\alpha - \beta)P(t), \quad (7)$$

which is the well-known Malthusian equation, describing the *expected* variation (model) of population growth ( $\alpha > \beta$ ) or decline ( $\alpha < \beta$ ). Often, the notation  $P'(t)$  is used instead of  $\frac{dP}{dt}$ .

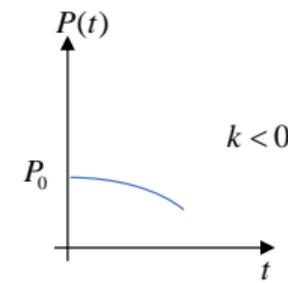
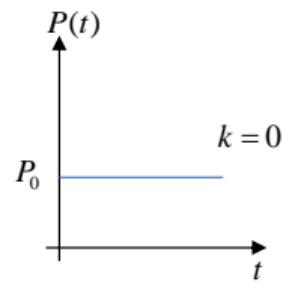
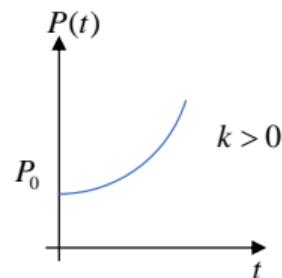
# What are Differential Equations

Let  $k = \alpha - \beta$ , then  $ce^{kt}$  with  $c \in \mathbb{R}$  an arbitrary constant, is the solution of our problem:

$$\frac{dP}{dt} = kP(t), \quad (8)$$

If we denote by  $P_0$  the number of individuals at the beginning of the study ( $t_0 = 0$ ),  $P(0) = P_0$ , then the solution to the model is given by (the arbitrary constant become a specific value:  $P_0$ ):

$$P(t) = P_0 e^{kt} \quad (9)$$



# What are Differential Equations

---

The differential equation  $\frac{dP}{dt} = kP(t)$  together with the initial condition  $P(0) = P_0$  is known as an **Initial Value Problem**.

This differential equation, which represents the Malthusian model, can also be written in the form:

$$\frac{dy(x)}{dx} = ky(x) \quad \text{or} \quad y' = ky$$

where the typical notation of  $x$  and  $y$  is used. From now on, we will preferably use this notation, where  $y$  is a function of  $x$ , with  $x$  being the independent variable.

# What are Differential Equations

A differential equation is an equality that involves a function of one or more variables and its derivatives up to a certain order.

## Example

$$x^2 \frac{d^2y}{dx^2} - xy \left( \frac{dy}{dx} \right)^4 = 0 \quad \text{or} \quad x^2 y'' - xy(y')^4 = 0 \quad (10)$$

$$y' = y \quad (11)$$

$$y'' + 2yy' = 3x \quad (12)$$

$$\frac{d^3v}{dt^3} + 5v \frac{dv}{dt} = \cos t \quad \text{or} \quad v''' + 5vv' = \cos t \quad (13)$$

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + 2 \frac{\partial^2 u}{\partial z^2} = 0 \quad (14)$$

# Solving Differential Equations

- The theory of differential equations began in the late 17th century, influenced by the works of G.W. Leibniz, I. Barrow, I. Newton, and the Bernoulli brothers, who solved simple first and second-order equations in mechanics and geometry. A notable milestone occurred on November 11, 1675, when Leibniz solved the equation  $\frac{dy}{dx} = x$ , using the integral symbol ( $\int$ ).
- The Malthus model is a simple ODE known as a separable variable differential equation. Its solution can be easily derived, as described in the following equations. Assuming that  $P \neq 0$ , we start with:

$$\frac{dP}{dt} = kP \Leftrightarrow \frac{1}{P} dP = k dt, \quad (15)$$

Next, we apply the integral to both sides of the equation:

$$\int \frac{1}{P} dP = \int k dt \Rightarrow \ln |P| = kt + c_1 \Rightarrow P(t) = c_2 e^{kt}, \quad (16)$$

where  $c_1$  and  $c_2$  are constants, with  $c_2 \in \mathbb{R}_0^+$ .

# Solving Differential Equations

---

- Newton's equation of motion was pivotal in developing calculus, categorizing first-order equations into forms such as  $\frac{dy}{dx} = f(x, y)$ . Leibniz introduced the notation for derivatives and integral, and he developed the theory of **separable differential equations** and discovered methods for solving **linear first-order equations**.
- The 18th century saw significant advancements in the theory of differential equations, with contributions from Jacob and Johann Bernoulli, as well as prominent mathematicians like Clairaut, D'Alembert, and Euler. Euler established conditions for **exact first-order equations** and developed the theory of **integrating factors**.
- In the 19th century, Dirichlet proved the convergence of Fourier series, and Cauchy rigorously defined convergence concepts. Liouville established the limitations of solving differential equations using elementary functions, while the works of Picard and Peano addressed the existence and uniqueness of solutions for initial value problems.
- The second half of the 20th century witnessed advancements in **computational methods for solving differential equations**, thanks to the contributions of Carl Runge and Martin Kutta.

# Solving Differential Equations

## Engineering

Engineers apply scientific and mathematical principles to design, build and analyze systems. Their work is mostly practical and oriented towards solving real-world problems.

## Applied Mathematics

Applied mathematicians use mathematical techniques and theories to solve practical problems in various fields. They focus on modeling real-world phenomena and finding numerical solutions.

## Pure Mathematics

Pure mathematicians study mathematical concepts for their intrinsic value, without necessarily considering practical applications. They are more concerned with exploring theoretical aspects of mathematics.

The three fields are all essential because they approach problems differently.

- Pure mathematicians focus on determining whether a mathematical solution exists and under what conditions it can be applied.
- In contrast, engineers assume that a solution exists and immediately begin working on finding an exact or approximate solution to the problem.

# Solving Differential Equations

For example, mathematicians are concerned with the following type of results for differential equations:

## Theorem (Existence and Uniqueness of Solution)

*Consider the differential equation*

$$\frac{dy}{dx} = f(x, y)$$

*where 1-The function  $f$  is continuous in a domain  $D$  of the  $xy$ -plane;*

*2-The partial derivative  $\frac{\partial f}{\partial y}$  is also continuous in  $D$ .*

*Let  $(x_0, y_0)$  be a point in  $D$ . Then the differential equation has a unique solution  $\phi$  in the interval  $]x_0 - h, x_0 + h[$  (or  $|x - x_0| < h$ ), for sufficiently small  $h$ , which satisfies the condition*

$$\phi(x_0) = y_0.$$

# Solving Differential Equations

---

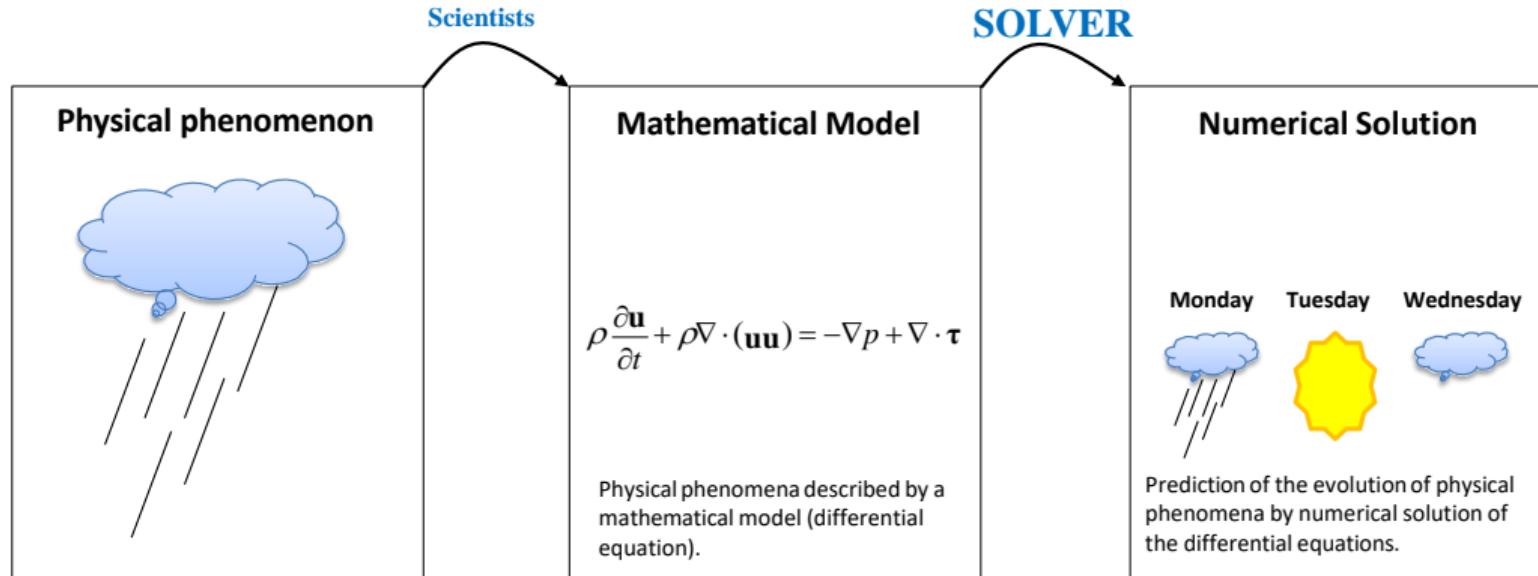
We previously discussed how to find the analytical solution to the equation

$$\frac{dP}{dt} = kP(t)$$

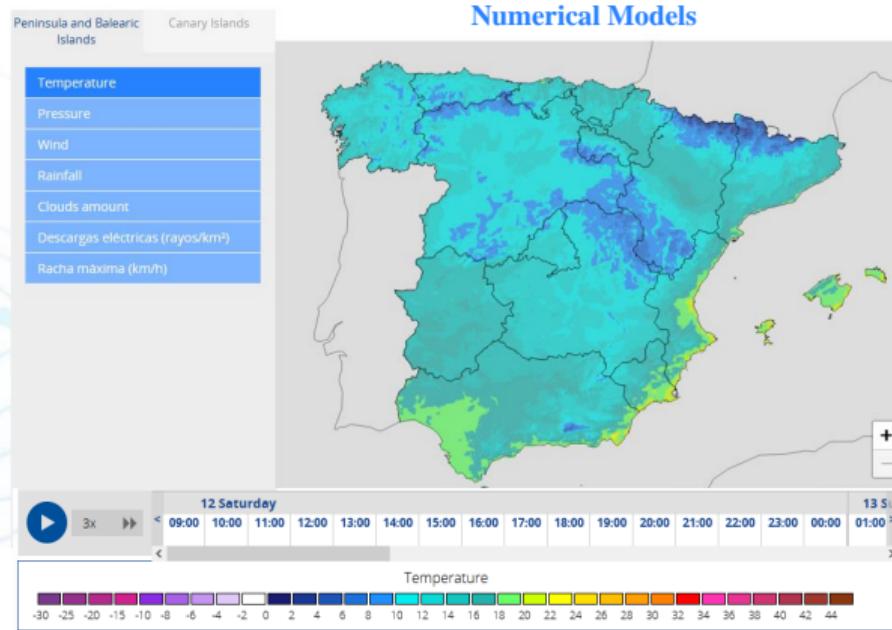
with the initial condition  $P(0) = P_0$ .

However, not all differential equations are this straightforward. In many cases, we must rely on **numerical solutions** to solve more complex equations!

# Solving Differential Equations



# Solving Differential Equations



Please note that while we can obtain weather forecasts for every hour, we cannot provide a forecast for a specific time, such as 10:36.

# Solving Differential Equations

---

Consider the simple differential equation,

$$\frac{dP}{dt} = kP(t)$$

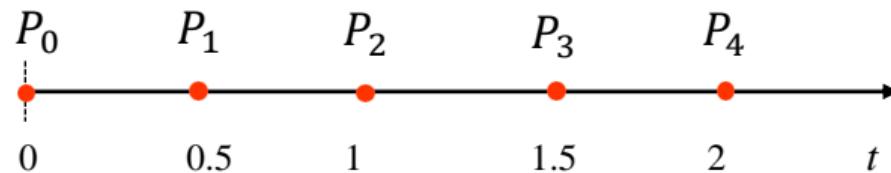
with the initial condition  $P(0) = P_0$ . We will now obtain its numerical solution!

For that we need to define the kind of approximation we will use, that is, the numerical method:

- Finite Difference (FD)
- Finite Volume (FV)
- Finite Elements (FE)
- other methods ...

# Solving Differential Equations - FD

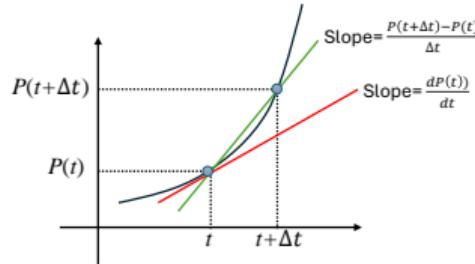
- Define an interval:  $[0, 2]$
- Create a mesh: we will consider 5 mesh elements (4 intervals of 0.5)  $\Delta t = 0.5$



- Initial condition:  $P(0) = P_0 = 2$
- Let:  $k = 1$
- **Objective:** determine the discrete solution  $P_1, P_2, P_3, P_4$

# Solving Differential Equations - FD

- Obtain approximations for  $\frac{dP}{dt}$  at  $t_1 = 0.5, t_2 = 1, t_3 = 1.5, t_4 = 2$



- $\frac{dP(t_i)}{dt} \approx \frac{P_i - P_{i-1}}{\Delta t}, i = 1, \dots, 4$
- Solve the system of equations (**Explicit Euler Method**):

$$\frac{P_1 - P_0}{\Delta t} = P_0 \Leftrightarrow P_1 = P_0 + \Delta t P_0$$

$$P_2 = P_1 + \Delta t P_1$$

$$P_3 = P_2 + \Delta t P_2$$

$$P_4 = P_3 + \Delta t P_3$$

# Solving Differential Equations - FD

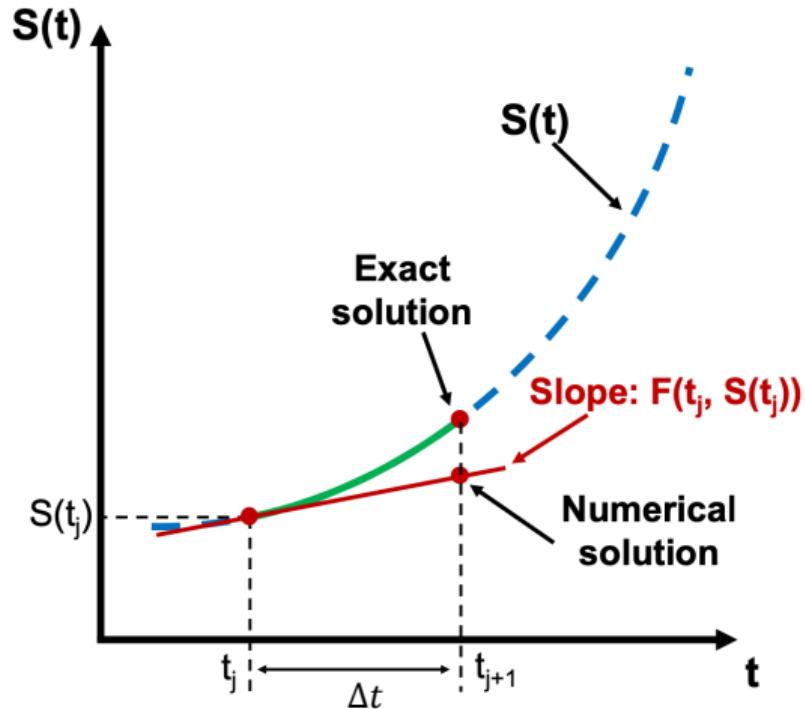
Let  $\frac{dS(t)}{dt} = F(t, S(t))$  be an explicitly defined first order ODE. That is,  $F$  is a function that returns the derivative, or change, of a state given a time and state value. Also, let  $t_j$  be a numerical grid point of the interval  $[t_0, t_f]$  with spacing  $\Delta t$ . Without loss of generality, we assume that  $t_0 = 0$ , and that  $t_f = N\Delta t$  for some positive integer,  $N$ . We then approximate the solution at  $S(t_{j+1})$  by:

$$S(t_{j+1}) = S(t_j) + (t_{j+1} - t_j) \frac{dS(t_j)}{dt}$$

which can also be written as

$$S(t_{j+1}) = S(t_j) + \Delta t \ F(t_j, S(t_j))$$

# Solving Differential Equations - FD



<https://pythonnumericalmethods.studentorg.berkeley.edu>

# Solving Differential Equations - FD

## Implicit Euler Method:

$$S(t_{j+1}) = S(t_j) + \Delta t \ F(t_{j+1}, S(t_{j+1}))$$

- for our previous example, we obtain the following system of equations:

$$\frac{P_1 - P_0}{\Delta t} = kP_1$$

$$\frac{P_2 - P_1}{\Delta t} = kP_2$$

$$\frac{P_3 - P_2}{\Delta t} = kP_3$$

$$\frac{P_4 - P_3}{\Delta t} = kP_4$$

# Solving Differential Equations - Hands On

It's now time for you to do it by yourselves (Part I)!

The differential equation  $\frac{df(t)}{dt} = e^{-t}$  with initial condition  $f(0) = -1$  has the exact solution  $f(t) = -e^{-t}$ . Approximate the solution to this initial value problem between 0 and 1 in increments of 0.1 using the **Explicit Euler method**. Plot the difference between the approximated solution and the exact solution.

Play with the **solver**, **model parameters** and the **number of mesh elements**.  
[link](#)

# Solving Differential Equations - Hands On

---

Its now time for you to do it by yourselves (Part I)!

---

```
import numpy as np
import matplotlib.pyplot as plt

# Define parameters
f = lambda t, s: np.exp(-t) # ODE
h = 0.3 # Step size
t = np.arange(0, 1 + h, h) # Numerical grid
s0 = -1 # Initial Condition

# Explicit Euler Method
s = np.zeros(len(t))
s[0] = s0

for i in range(0, len(t) - 1):
    s[i + 1] = s[i] + h*f(t[i], s[i])
```

---

# Solving Differential Equations - Hands On

---

Cont.

---

```
plt.figure(figsize = (6, 5))
plt.plot(t, s, 'bo--', label='Approximate')
plt.plot(t, -np.exp(-t), 'g', label='Exact')
plt.title('Approximate and Exact Solution \
for Simple ODE')
plt.xlabel('t')
plt.ylabel('f(t)')
plt.grid()
plt.legend(loc='lower right')
plt.show()
```

---

# Solving Differential Equations - Hands On

It's now time for you to do it by yourselves (Part II)! You do not need to implement the numerical solver, Python can easily deal with that:

---

```
sol = solve_ivp(ODE, [0, 5], y0, t_eval=t)
```

---

Consider a population of organisms that follows a logistic growth. The population size  $P(t)$  at time  $t$  is governed by the following differential equation:

$$P'(t) = rP(t) \left(1 - \frac{P(t)}{K}\right), \quad P(t_0) = 100, \quad (17)$$

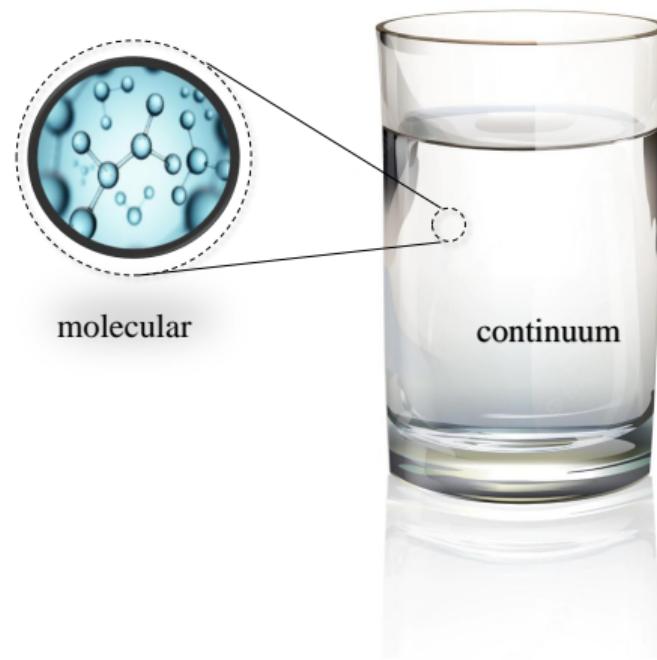
where  $r$  is the growth rate, and  $K$  is the carrying capacity of the environment.

Consider  $r = 0.1$ ,  $K = 1000$ . Also, consider a mesh of 200 points.

Play with the **solver**, **model parameters** and the **number of mesh elements**.

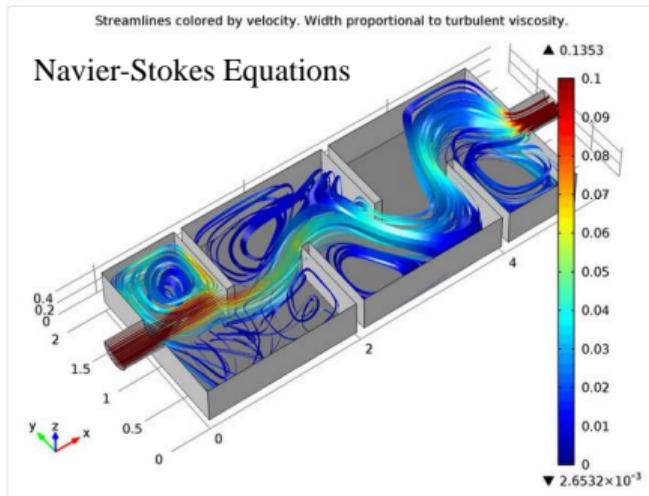
# Differential Equations in Real Life

Most interesting real life applications are modelled by the **Partial Differential Equations**



# Differential Equations in Real Life

Most interesting real life applications are modelled by the **Partial Differential Equations**



They are used by engineers and physicists all over the world in many fields, which go from aircraft design to blood circulation. They are also very complicated to solve and that is why they are one of the seven [Millennium Prize Problem](#). Solving one of these problems will win you a million dollars

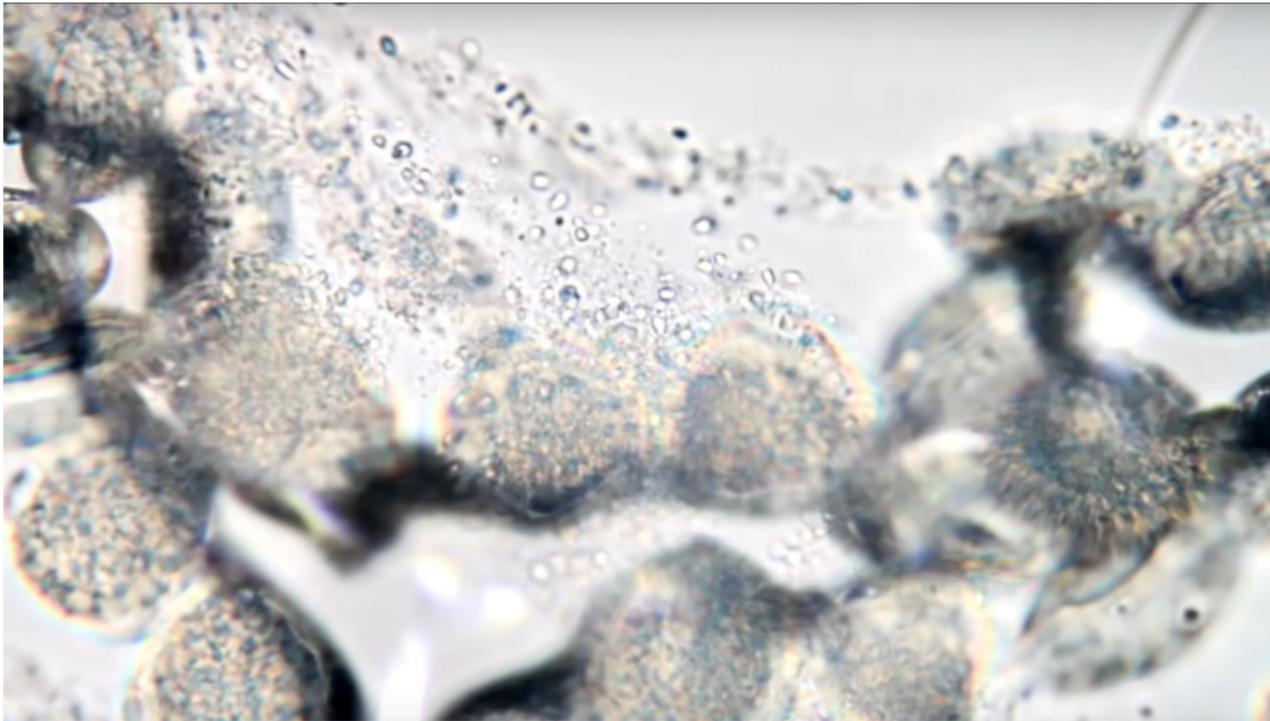
$$\nabla \cdot \bar{u} = 0$$

$$\rho \frac{D\bar{u}}{Dt} = -\nabla p + \mu \nabla^2 \bar{u} + \rho \bar{F}$$

Navier-Stokes equations

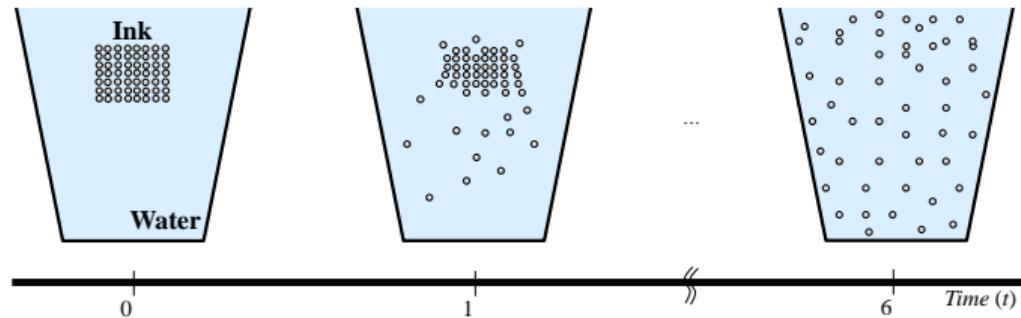
# Differential Equations in Real Life

---



# Differential Equations in Real Life

Diffusion can be seen as a transport phenomena where distribution, mixing or transport of mass/particles occurs without requiring bulk motion (the spreading of something more widely).



$$\frac{\partial u(t, x)}{\partial t} = D \frac{\partial^2 u(t, x)}{\partial x^2} \quad (1D) \quad (18)$$

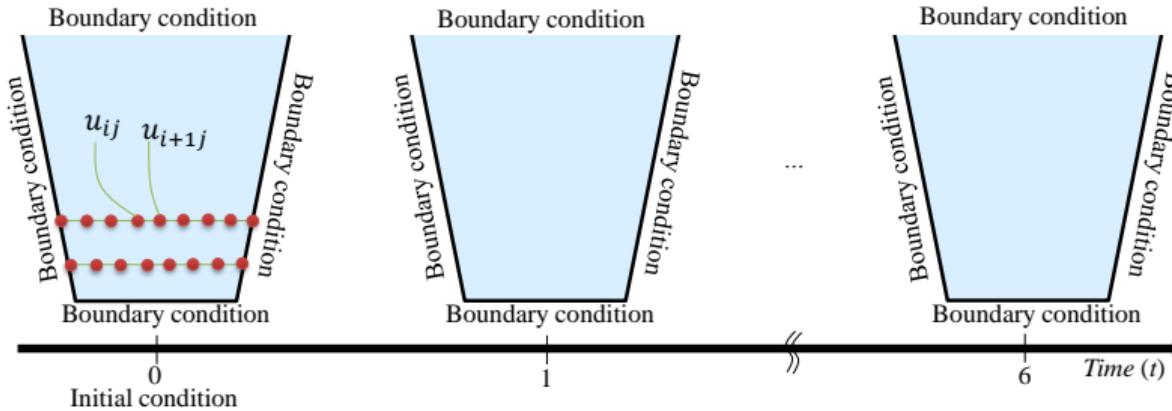
$$\frac{\partial u(t, x, y)}{\partial t} = D \left( \frac{\partial^2 u(t, x, y)}{\partial x^2} + \frac{\partial^2 u(t, x, y)}{\partial y^2} \right) \quad (2D) \quad (19)$$

# Neural Networks for Solving DEs

---

- Challenges of Numerical Methods;
- Physics-Informed Neural Networks;
- Hands-on with the DeepXDE Library.

# Challenges of Numerical Methods



**Explicit**

$$\frac{u_{ij}^{n+1} - u_{ij}^n}{\Delta t} = D \left( \frac{u_{i+1j}^n - 2u_{ij}^n + u_{i-1j}^n}{\Delta x^2} + \frac{u_{ij+1}^n - 2u_{ij}^n + u_{ij-1}^n}{\Delta y^2} \right)$$

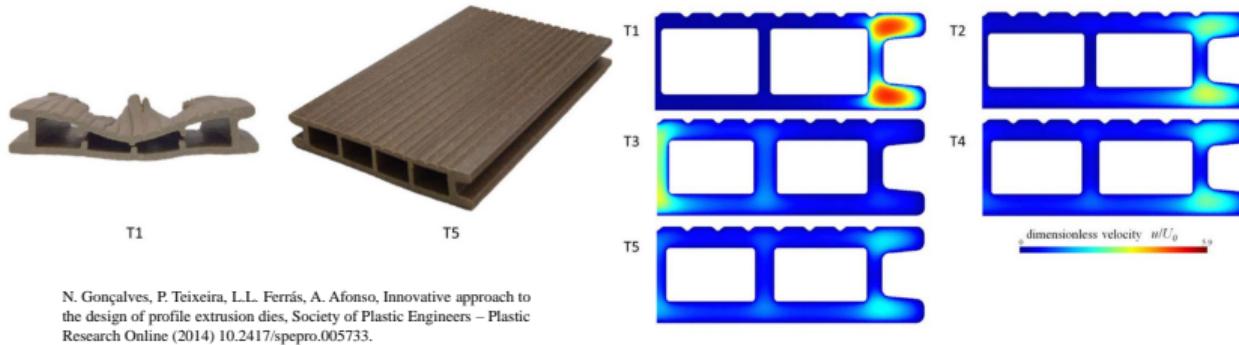
**Implicit**

$$\frac{u_{ij}^{n+1} - u_{ij}^n}{\Delta t} = D \left( \frac{u_{i+1j}^{n+1} - 2u_{ij}^{n+1} + u_{i-1j}^{n+1}}{\Delta x^2} + \frac{u_{ij+1}^{n+1} - 2u_{ij}^{n+1} + u_{ij-1}^{n+1}}{\Delta y^2} \right)$$

Solve a system of equations for each time step

# Challenges of Numerical Methods

There are several numerical methods and different solvers for solving differential equations, but nearly all of them share one major drawback — **they are time and memory-consuming!** Simulations can sometimes take months to complete.



# Two Paradigms

---

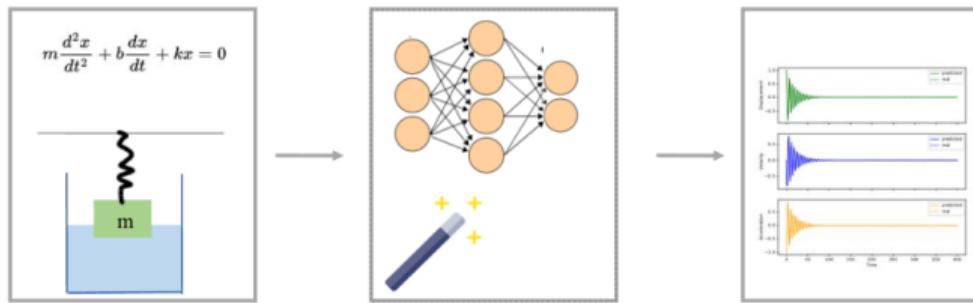
## Solving differential equations

- Complete knowledge of the system;
- Exact solutions can be found but not often;
- Numerical solutions involve numerical methods;
- Used for finding the exact solutions for ODEs and PDEs in physics and engineering, where the system's behaviour is well-defined mathematically.

## Modelling differential equations

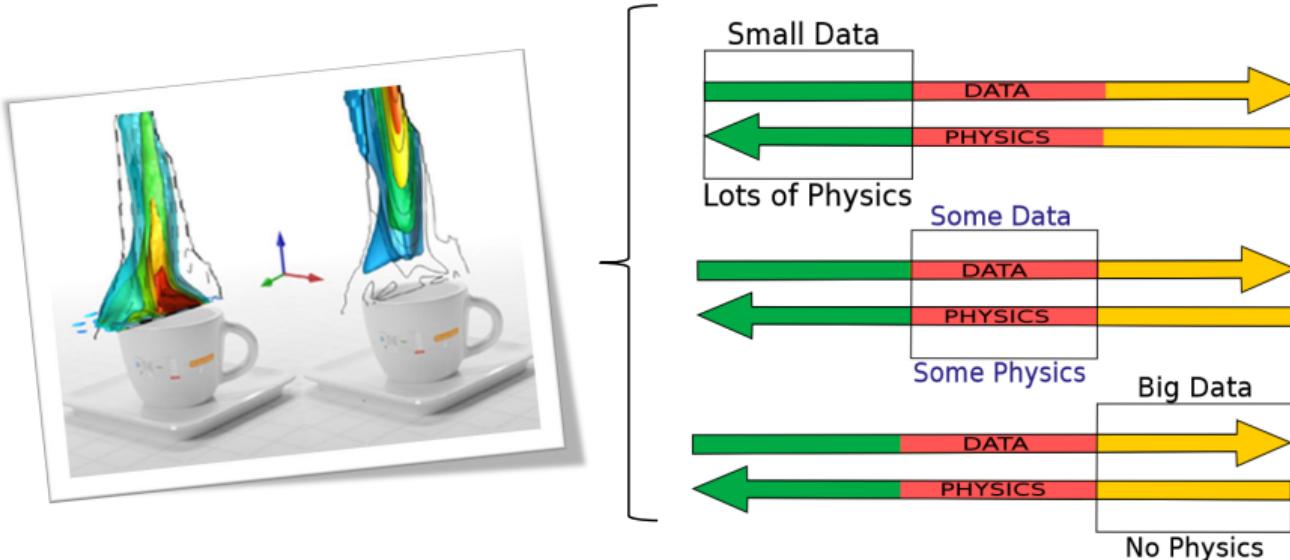
- Fully or partially unknown governing equations;
- Requires observed data to adjust the parameters or functions to match the data;
- Involves traditional optimization methods;
- Used for modelling systems in epidemiology, ecology, finance, and others.

# Neural Networks for Solving DEs



- Real systems in physics, chemistry and engineering are typically described by differential equations;
- When differential equations are known, numerically solving them can be computationally prohibitive [3];
- Physics-Informed Neural Networks (PINNs) use a neural network to approximate the curve of solutions [3];
- PINNs offer a faster and more cost-effective alternative to numerical methods when doing predictions [3].

# Physics-Informed Neural Networks



Raiissi, M., Perdikaris, P., Ahmadi, N., & Karniadakis, G. E. (2024). Physics-informed neural networks and extensions. *arXiv preprint arXiv:2408.16806*.

Cai, S., Wang, Z., Fuest, F., Jeon, Y. J., Gray, C., & Karniadakis, G. E. (2021). Flow over an espresso cup: inferring 3-D velocity and pressure fields from tomographic background oriented Schlieren via physics-informed neural networks. *Journal of Fluid Mechanics*, 915, A102.

# Physics-Informed Neural Networks

Consider a parameterised and nonlinear partial differential equation (PDE) of the general form:

$$\frac{\partial u(t, x)}{\partial t} + \mathcal{N}[u; \lambda] = 0, \quad x \in \Omega, t \in [0, T], \quad (20)$$

where  $u(t, x)$  is the solution,  $\mathcal{N}[u; \lambda]$  is a differential operator and  $\Omega$  the domain of  $x$ . This setup encapsulates a wide range of problems in mathematical physics including conservation laws, diffusion processes, advection–diffusion–reaction systems, and kinetic equations.

# PINNs - 1D Burger's Equation

---

As a motivating example, take the 1D Burger's equation, corresponding to

$$\mathcal{N}[u; \lambda] = \lambda_1 u \frac{du}{dx} - \lambda_2 \frac{\partial^2 u}{\partial x^2}:$$

$$\frac{\partial u}{\partial t} + \lambda_1 u \frac{\partial u}{\partial x} - \lambda_2 \frac{\partial^2 u}{\partial x^2} = 0, \quad x \in \Omega, t \in [0, T]. \quad (21)$$

This equation arises in various areas of applied mathematics, including fluid mechanics, nonlinear acoustics, gas dynamics, and traffic flow. For small values of the viscosity parameters  $\lambda_1, \lambda_2$ , it is hard to solve by numerical methods.

Given noisy measurements of the system, we are interested in the solution of two distinct problems: data-driven solutions of PDEs, and data-driven discovery of PDEs [3].

# Data-driven Solutions of PDEs

**Problem:** Given fixed model parameters  $\lambda$ , what can be said about the unknown  $u(t, x)$ ?

PINNs approximate the solution  $u(t, x)$  using a neural network  $\hat{u}(t, x; \theta)$ , where  $\theta$  represents the network parameters. The PINN is trained to fit the data  $(u_n)$  and satisfy the PDE, given by Eq. (20) [3]. This can be formulated as a constrained optimisation problem:

$$\begin{aligned} & \underset{\theta \in \mathbb{R}^{n_\theta}}{\text{minimize}} \quad l(\theta) = \frac{1}{N} \sum_{n=1}^N (\hat{u}_n(\theta) - u_n)^2 \\ & \text{subject to} \quad \frac{\partial u(t, x)}{\partial t} + \mathcal{N}[u; \lambda] = 0, \quad t = [0, T], \end{aligned} \tag{22}$$

# Data-driven Solutions of PDEs

To do this, we can rewrite the constrained problem (22) into an unconstrained problem by using a penalty method [2]:

$$\underset{\boldsymbol{\theta} \in \mathbb{R}^{n_{\theta}}}{\text{minimize}} \quad l(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^N (\hat{u}_n(\boldsymbol{\theta}) - u_n)^2 + \mu \left( \frac{\partial u(t, x)}{\partial t} + \mathcal{N}[u; \lambda] \right) \quad (23)$$

Then, the loss function of the neural network  $\hat{u}(t, x; \boldsymbol{\theta})$  consists of two components: the error of the fit to the data, and the violation of the PDE constraint multiplied by a weighting factor  $\mu$  [3].

# Data-driven Solutions of PDEs

---

To train the neural network  $\hat{u}(x; \theta)$ , data is needed. Collocation points are crucial in training PINNs and these are taken from locations to enforce the PDE's loss term. Likewise, initial and boundary training data are used to fit the data:

- **Collocation points**,  $N_{PDE}$ : scattered points within the domain where the PDE is enforced;
- **Initial/boundary points**,  $N_u$ : points where the solution is known to enforce initial and boundary conditions [3].

## Example - 1D Burger's Equation

Consider the 1D Burger's Equation with Dirichlet boundary conditions and  $\lambda_1 = 1, \lambda_2 = \frac{0.01}{\pi}$  [3]:

$$\begin{aligned} \frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} - \frac{0.01}{\pi} \frac{\partial^2 u}{\partial x^2} &= 0, \quad x \in [-1, 1], t \in [0, 1], \\ u(0, x) &= -\sin(\pi x), \\ u(t, -1) &= u(t, 1) = 0. \end{aligned}$$

## Example - 1D Burger's Equation

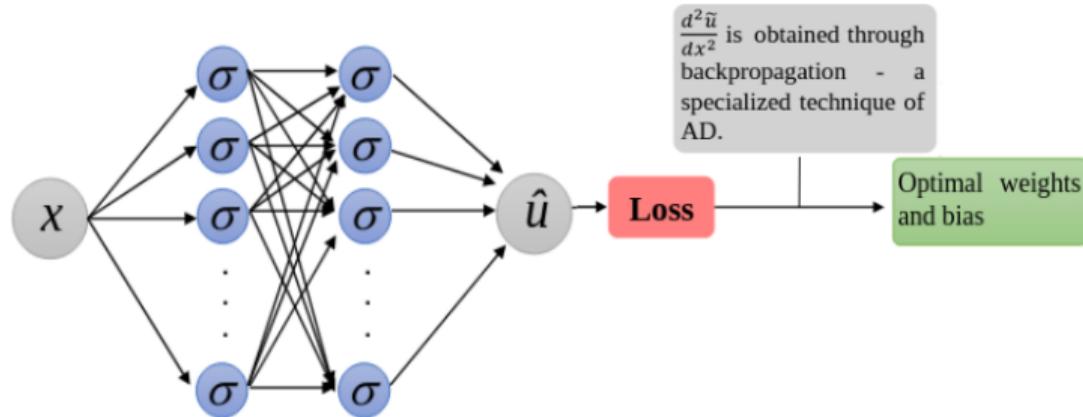
First, we define the penalty term as  $\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} - \frac{0.01}{\pi} \frac{\partial^2 u}{\partial x^2}$  and build the loss function:

$$\begin{aligned} & \underset{\boldsymbol{\theta} \in \mathbb{R}^{n_\theta}}{\text{minimize}} \quad l(\boldsymbol{\theta}) = \frac{1}{N_u} \sum_{n=1}^{N_u} (\hat{u}_n(\boldsymbol{\theta}) - u_n)^2 + \\ & \quad \frac{1}{N_{PDE}} \sum_{n=1}^{N_{PDE}} \left( \frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} - \frac{0.01}{\pi} \frac{\partial^2 u}{\partial x^2} \right) \end{aligned} \tag{24}$$

Then we proceed by approximating  $u(t, x)$  by a neural network [3].

# Example - 1D Burger's Equation

To compute the derivatives of  $u(t, x)$ , PINNs use backpropagation [3].



# Example - 1D Burger's Equation

---

First, the DeepXDE and TensorFlow modules are imported:

---

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

---

Then we start by defining the geometry and time domain of the problem using the built-in classes [3, 1]:

---

```
geom = dde.geometry.Interval(-1, 1)
timedomain = dde.geometry.TimeDomain(0, 1)
geomtime = dde.geometry.GeometryXTime(geom, timedomain)
```

---

# Example - 1D Burger's Equation

---

Next, we code the PDE:

---

```
def pde(x, u):
    du_x = dde.grad.jacobian(u, x, i=0, j=0)
    du_t = dde.grad.jacobian(u, x, i=0, j=1)
    du_xx = dde.grad.hessian(u, x, i=0, j=0)

    return du_t + u * du_x - 0.01 / np.pi * du_xx
```

---

The first argument  $x$  is a vector in which the first component is the  $x$ -coordinate and the second component is the  $t$ -coordinate. The second argument  $u$  is the output given by the neural network [3, 1].

## Example - 1D Burger's Equation

Then we define the initial and boundary conditions using the geometry and time domains previously defined:

---

```
bc = dde.icbc.DirichletBC(geomtime, lambda x: 0,
lambda _, on_boundary: on_boundary)
ic = dde.icbc.IC(geomtime, lambda x: -np.sin(np.pi *
x[:, 0:1]), lambda _, on_initial: on_initial)
```

---

The first argument  $x$  is a vector in which the first component is the  $x$ -coordinate and the second component is the  $t$ -coordinate. The second argument  $u$  is the output given by the neural network [3, 1].

## Example - 1D Burger's Equation

Now, we have specified the geometry, PDE, and initial and boundary conditions. Thus we define the time-dependent PDE problem using the built-in function:

---

```
data = dde.data.TimePDE(geomtime, pde, [bc, ic],  
    num_domain=2540, num_boundary=80, num_initial=160)
```

---

The number 2540 is the number of training points sampled inside the domain,  $N_{PDE}$ . The number 80 is the number of training points sampled on the boundary and 160 are the initial points for the initial conditions,  $N_u$  [3, 1].

## Example - 1D Burger's Equation

---

Next we define a neural network architecture. Here, we use a fully connected neural network of depth 4 and width 20:

---

```
net = dde.nn.FNN([2] + [20] * 3 + [1], "tanh",
    "Glorot normal")
```

---

Then we build the network and choose an optimiser [3, 1]:

---

```
model = dde.Model(data, net)
model.compile("adam", lr=1e-3)
```

---

## Example - 1D Burger's Equation

---

We then train the model for 15000 iterations:

---

```
losshistory, train_state = model.train(iterations=15000)
```

---

After we train the network using Adam, we continue to train the network using L-BFGS to achieve a smaller loss [3, 1]:

---

```
model.compile("L-BFGS-B")
losshistory, train_state = model.train()
```

---

# Example - 1D Burger's Equation

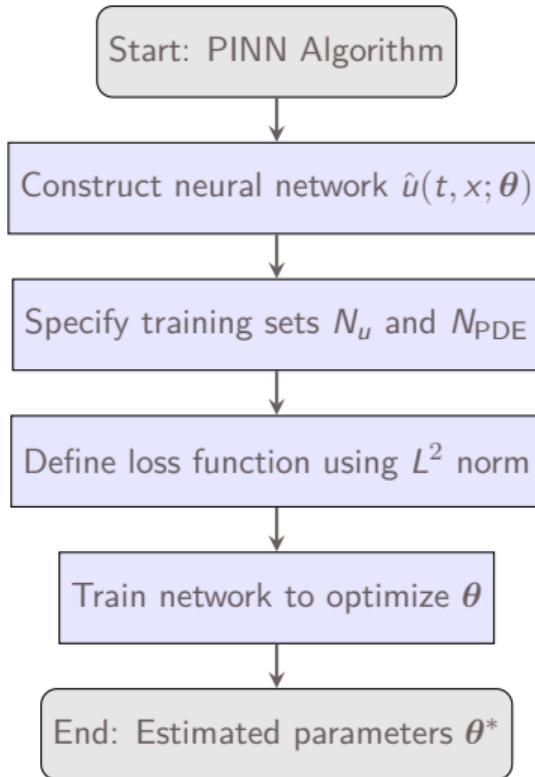
---

```
def gentestdata():
    data = np.load("Burgers.npz")
    t, x, exact = data["t"], data["x"], data["usol"].T
    xx, tt = np.meshgrid(x, t)
    X = np.vstack((np.ravel(xx), np.ravel(tt))).T
    y = exact.flatten()[:, None]
    return X, y

X, y_true = gentestdata()
y_pred = model.predict(X)
f = model.predict(X, operator=pde)
print("Mean residual:", np.mean(np.absolute(f)))
print("L2 relative error:",
      dde.metrics.l2_relative_error(y_true, y_pred))
np.savetxt("test.dat", np.hstack((X, y_true, y_pred)))
```

---

# Review: Data-Driven Solution



# Data-driven Discovery of PDEs

**Problem:** *What are the parameters  $\lambda$  that best describe the observed data?*

Again, PINNs approximate the solution  $u(x)$  using a neural network  $\hat{u}(t, x; \theta)$ , where  $\theta$  represents the network parameters. The PINN is trained to fit the data and satisfy the PDE:

$$\begin{aligned} & \underset{\theta, \lambda \in (\mathbb{R}^{n_\theta}, \mathbb{R}^{n_\lambda})}{\text{minimize}} && I(\theta, \lambda) = \frac{1}{N} \sum_{n=1}^N (\hat{u}_n(\theta) - u_n)^2 \\ & \text{subject to} && \frac{\partial u(t, x)}{\partial t} + \mathcal{N}[u; \lambda] = 0, \quad t = [0, T], \end{aligned} \tag{25}$$

**Additionally, the parameters  $\lambda$  turn into parameters of the PINN [4].**

# Data-driven Discovery of PDEs

The loss function of the neural network  $\hat{u}(t, x; \boldsymbol{\theta})$  consists of the two components: the error of the fit to the data, and the violation of the PDE constraint multiplied by a weighting factor  $\mu$ .

$$\underset{\boldsymbol{\theta}, \boldsymbol{\lambda} \in (\mathbb{R}^{n_\theta}, \mathbb{R}^{n_\lambda})}{\text{minimize}} \quad l(\boldsymbol{\theta}, \boldsymbol{\lambda}) = \frac{1}{N} \sum_{n=1}^N (\hat{u}_n(\boldsymbol{\theta}) - u_n)^2 + \textcolor{red}{\mu} \left( \frac{\partial u}{\partial t} + \mathcal{N}[u; \boldsymbol{\lambda}] \right) \quad (26)$$

where  $N$  is composed of initial/boundary points  $N_u$  and experimental data  $N_{\text{data}}$ ,  
 $N = \{N_u, N_{\text{data}}\}$

**The parameters  $\lambda$  are learnt by being optimised along the neural network parameters [4].**

# Data-driven Discovery of PDEs

---

To train the neural network  $\hat{u}(t, x; \theta)$  and discover the parameters  $\lambda$ , data is needed:

- **Collocation points**,  $N_{PDE}$ : scattered points within the domain where the PDE is enforced;
  - **Initial/boundary points**,  $N_u$ : points where the solution is known to enforce initial and boundary conditions, as well as to discover the unknown parameters  $\lambda$ .
- Unlike the problem of "data-driven solutions", in this case some training data from the solution is needed, which can be experimental data  $N_{\text{data}}$  [4].**

## Example - Diffusion Equation

---

Consider a diffusion equation with an unknown parameter  $C$  and Dirichlet boundary conditions [4]:

$$\frac{\partial u}{\partial t} = C \frac{\partial^2 u}{\partial x^2} - e^{-t} (\sin(\pi x) - \pi^2 \sin(\pi x)), \quad x \in [-1, 1], t \in [0, 1],$$
$$u(0, x) = \sin(\pi x),$$
$$u(t, -1) = u(t, 1) = 0.$$

## Example - Diffusion Equation

First, we define the penalty term by rewriting the PDE and build the loss function [4]:

$$\begin{aligned} \underset{\boldsymbol{\theta} \in \mathbb{R}^{n_{\theta}}}{\text{minimize}} \quad & I(\boldsymbol{\theta}) = \frac{1}{N_u} \sum_{n=1}^{N_u} (\hat{u}_n(\boldsymbol{\theta}) - u_n)^2 + \\ & \frac{1}{N_{PDE}} \sum_{n=1}^{N_{PDE}} \left( \frac{\partial u}{\partial t} - C \frac{\partial^2 u}{\partial x^2} + e^{-t} (\sin(\pi x) + \pi^2 \sin(\pi x)) \right) + \frac{1}{N_{\text{data}}} \sum_{n=1}^{N_{\text{data}}} (\hat{u}_n(\boldsymbol{\theta}) - u_n)^2 \end{aligned} \quad (27)$$

Then we proceed by approximating  $u(t, x)$  by a neural network [4].

## Example - Diffusion Equation

---

Then we start by defining the geometry and time domain of the problem using the built-in classes [4, 1]:

---

```
geom = dde.geometry.Interval(-1, 1)
timedomain = dde.geometry.TimeDomain(0, 1)
geomtime = dde.geometry.GeometryXTime(geom, timedomain)
```

---

# Example - Diffusion Equation

---

Next, we code the PDE [4, 1]:

---

```
def pde(x, u):
    du_t = dde.grad.jacobian(u, x, i=0, j=1)
    du_xx = dde.grad.hessian(u, x, i=0, j=0)

    return (dy_t - C * dy_xx + tf.exp(-x[:, 1:])
           * (tf.sin(np.pi * x[:, 0:1]) - np.pi ** 2
           * tf.sin(np.pi * x[:, 0:1])))
```

---

The first argument  $x$  is a vector in which the first component is the  $x$ -coordinate and the second component is the  $t$ -coordinate. The second argument  $u$  is the output given by the neural network.  $C$  is an unknown parameter that will be initialised as 2.0 [4, 1].

---

```
C = dde.Variable(2.0)
```

---

# Example - Diffusion Equation

---

Then we define the initial and boundary conditions using the geometry and time domains previously defined:

---

```
bc = dde.icbc.DirichletBC(geomtime, func,  
    lambda _, on_boundary: on_boundary)  
ic = dde.icbc.IC(geomtime, func, lambda _,  
    on_initial: on_initial)
```

---

The reference solution is [4, 1]:

---

```
def func(x):  
    return np.sin(np.pi * x[:, 0:1]) * np.exp(-x[:, 1:])
```

---

## Example - Diffusion Equation

---

In this problem, we provide extra information on some training points so the parameter  $C$  can be identified from these observations. We generate 10 equally-spaced input points  $(x, t)$ , with  $x \in [-1, 1]$  and  $t = 1$ , using the corresponding exact solution [4].

---

```
observe_x = np.vstack((np.linspace(-1, 1, num=10),
                      np.full((10), 1))).T
observe_y = dde.icbc.PointSetBC(observe_x,
                                func(observe_x), component=0)
```

---

## Example - Diffusion Equation

---

Now, we have specified the geometry, PDE, initial and boundary conditions, and extra observations. Thus we define the time-dependent PDE problem using the built-in function [4, 1]:

---

```
data = dde.data.TimePDE(geomtime, pde,  
[bc, ic, observe_y],  
num_domain=40, num_boundary=20, num_initial=10,  
anchors=observe_x, solution=func, num_test=10000)
```

---

The number 40 is the number of training points sampled inside the domain,  $N_{PDE}$ . The number 20 is the number of training points sampled on the boundary, 10 are the initial points for the initial conditions, furthermore *anchors* is the additional training points  $N_u$  [4, 1].

## Example - Diffusion Equation

---

Next we define a neural network architecture. Here, we use a fully connected neural network of depth 4 and width 32 [4, 1]:

---

```
net = dde.nn.FNN([2] + [32] * 3 + [1], "tanh",
    "Glorot normal")
```

---

Then we build the network, choose an optimiser and pass the unknown parameter  $C$  as a trainable variable [4, 1]:

---

```
model = dde.Model(data, net)
model.compile("adam", lr=1e-3,
metrics=["l2 relative error"],
external_trainable_variables=C)
```

---

## Example - Diffusion Equation

---

We then train the model for 50000 iterations and output  $C$  every 1000 iterations:

---

```
variable = dde.callbacks.VariableValue(C, period=1000)
losshistory, train_state = model.train(iterations=50000,
callbacks=[variable])
```

---

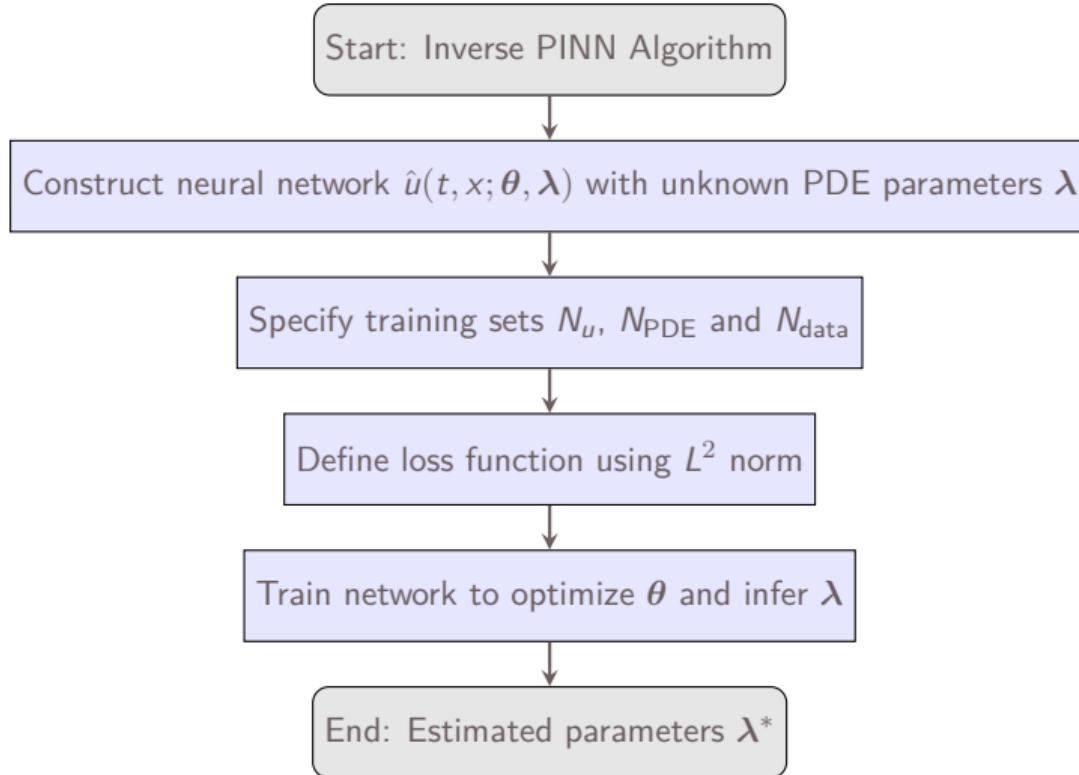
We also save and plot the best trained result and loss history [4, 1]:

---

```
dde.saveplot(losshistory, train_state, issave=True
, isplot=True)
```

---

# Review: Data-driven Discovery



## PINNs

- Mesh-independent;
- Suitable for problems in irregular or complex domains;
- Generalise to unseen data;
- Computes at arbitrary times directly;
- Data-hungry;
- Sensitive to hyperparameter choices and NN architectures;
- Computationally expensive to train;
- Lack transparency and interpretability.

## Numerical Methods

- Some methods rely on structured grids;
- Computationally expensive in high dimensions;
- Iterative method;
- Work with limited data;
- Established guidelines;
- Computationally efficient for some problems;
- Well-understood algorithms.

# What's next?

---

- Learning families of solutions with Neural Operators: Kovachki, Nikola, et al. "Neural operator: Learning maps between function spaces with applications to pdes." *Journal of Machine Learning Research* 24.89 (2023): 1-97.
- Modelling Differential Equations to Data with Neural ODEs: Chen, Ricky TQ, et al. "Neural ordinary differential equations." *Advances in neural information processing systems* 31 (2018).
- Physics-informed Machine Learning Playlist by Steve Brunton: [youtube.com/-playlist?list=PLMrJAhleNNQ0BaKuBKY43k4xMo6NSbBasi=T3F69J7eJjNMvGDL](https://youtube.com/-playlist?list=PLMrJAhleNNQ0BaKuBKY43k4xMo6NSbBasi=T3F69J7eJjNMvGDL)
- PINNs for medical imaging: Banerjee, Chayan, et al. "Pinns for medical image analysis: A survey." *arXiv preprint arXiv:2408.01026* (2024).
- Modelling blood flow: Kissas, Georgios, et al. "Machine learning in cardiovascular flows modeling: Predicting arterial blood pressure from non-invasive 4D flow MRI data using physics-informed neural networks." *Computer Methods in Applied Mechanics and Engineering* 358 (2020): 112623.

- [1] Lu Lu et al. "DeepXDE: A deep learning library for solving differential equations". In: *SIAM review* 63.1 (2021), pp. 208–228.
- [2] Jorge Nocedal and Stephen J Wright. *Numerical optimization*. Springer, 1999.
- [3] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational physics* 378 (2019), pp. 686–707.
- [4] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. "Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations". In: *arXiv preprint arXiv:1711.10561* (2017).



Cecília Coelho  
Professorship Computer Science in Mechanical Engineering  
Institute for Artificial Intelligence  
Helmut Schmidt University / University of the Federal Armed Forces Hamburg  
Holstenhofweg 85, 22043 Hamburg  
tel.: 040 / 6541-2440  
email: [cecilia.coelho@hsu-hh.de](mailto:cecilia.coelho@hsu-hh.de)