Neural Networks and Differential Equations: From Infinite Layers to Continuous Modelling

by

Cecília Coelho[1], **Luís Ferrás**[2,3], **Andrzej Dulny**[1]

[1] Institute for Artificial Intelligence, Helmut Schmidt University, Hamburg, Germany
[2] CEFT and DEMec (Section of Mathematics) - FEUP, University of Porto, Portugal
[3] Centre of Mathematics (CMAT), University of Minho, Portugal
cecilia.coelho@hsu-hh.de, lferras@fe.up.pt

Outline of the Tutorial

- Introduction to Differential Equations
  - What are Differential Equations
  - Solving Differential Equations
  - Differential Equations in Real life
- Neural Networks for Solving Differential Equations
  - Challenges of Numerical Methods
  - Physics-Informed Neural Networks
  - Hands-on with the DeepXDE Library
- Neural Networks for Modelling Differential Equations
  - Challenges on Differential Equations Formulation for Describing Real-world Systems
  - Neural Ordinary Differential Equations
  - Hands-on with the Torchdiffeq Library
- Wrap-up
  - Physics-Informed Neural Networks versus Neural Ordinary Differential Equations
  - What's Next? (Neural Operators, Neural Laplace, etc.)

Availability of the Materials

All materials can be found at the official tutorial's website, including jupyter notebooks with the coding examples and additional exercises.
https://ceciliacoelho.github.io/tutorialNN4DEs/

Introduction to Differential Equations

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

https://github.com/nnde-ecai

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 $t = 1 hour$ and $\Delta t = 4\ hours$, 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 α and β ($\in \mathbb{R}^+$) are the proportionality constants, which may vary from population to population.

- It is also expected that N and M depend on the time interval Δt. 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 Δt can then be expressed as:

$$N = \alpha P(t)\Delta t, \qquad M = \beta P(t)\Delta t. \tag{4}$$

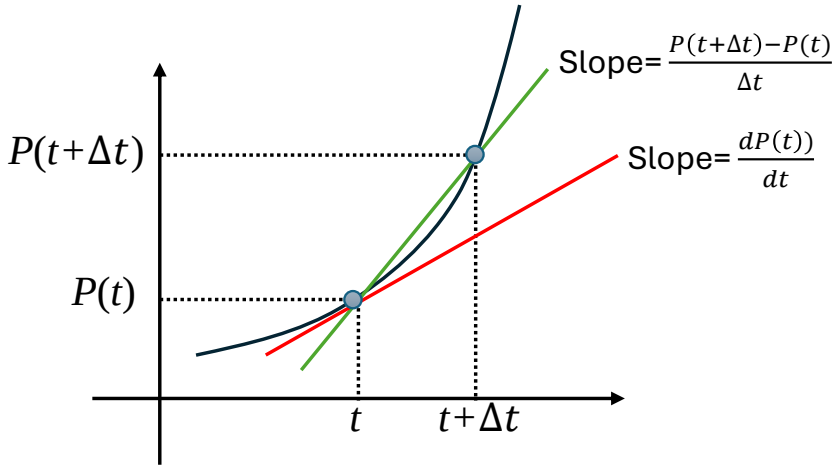
Consequently, the change in population over the time interval $[t,\ t + \Delta t]$ is given by $P(t + \Delta t) - P(t) = N - M$, that is:

$$P(t + \Delta t) - P(t) = (\alpha - \beta)P(t)\Delta t. \tag{5}$$

Dividing both sides by Δt, we obtain the following equation:

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

What are Differential Equations



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

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

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 $\frac{dP}{dt}$.

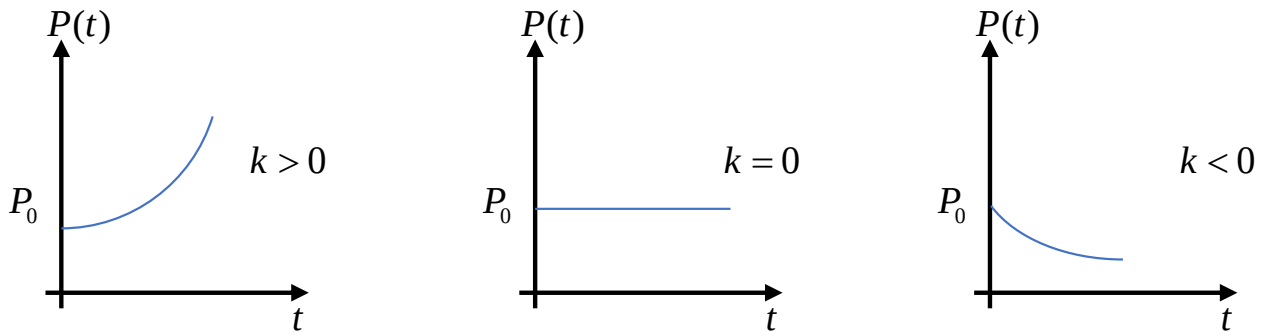
What are Differential Equations

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

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

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):

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



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

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

$$y' = y \tag{11}$$

$$y'' + 2yy' = 3x \tag{12}$$

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

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

What are Differential Equations

Definition (Ordinary Differential Equation (ODE))

A differential equation involving derivatives of one or more dependent variables with respect to an independent variable is called an ordinary differential equation (ODE).

Definition (Partial Differential Equation (PDE))

is a differential equation that involves partial derivatives of a multivariable function. A **PDE** involves one or more dependent variables and their partial derivatives with respect to two or more independent variables.

Example: For a function $u(x, y)$, a PDE might look like:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

This is known as the **Laplace equation**, which appears in heat conduction, fluid flow, and electrostatics.

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:

$$\frac{dP}{dt} = kP \Leftrightarrow \frac{1}{P}\, dP = k\, dt, \tag{15}$$

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

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

    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* ϕ *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

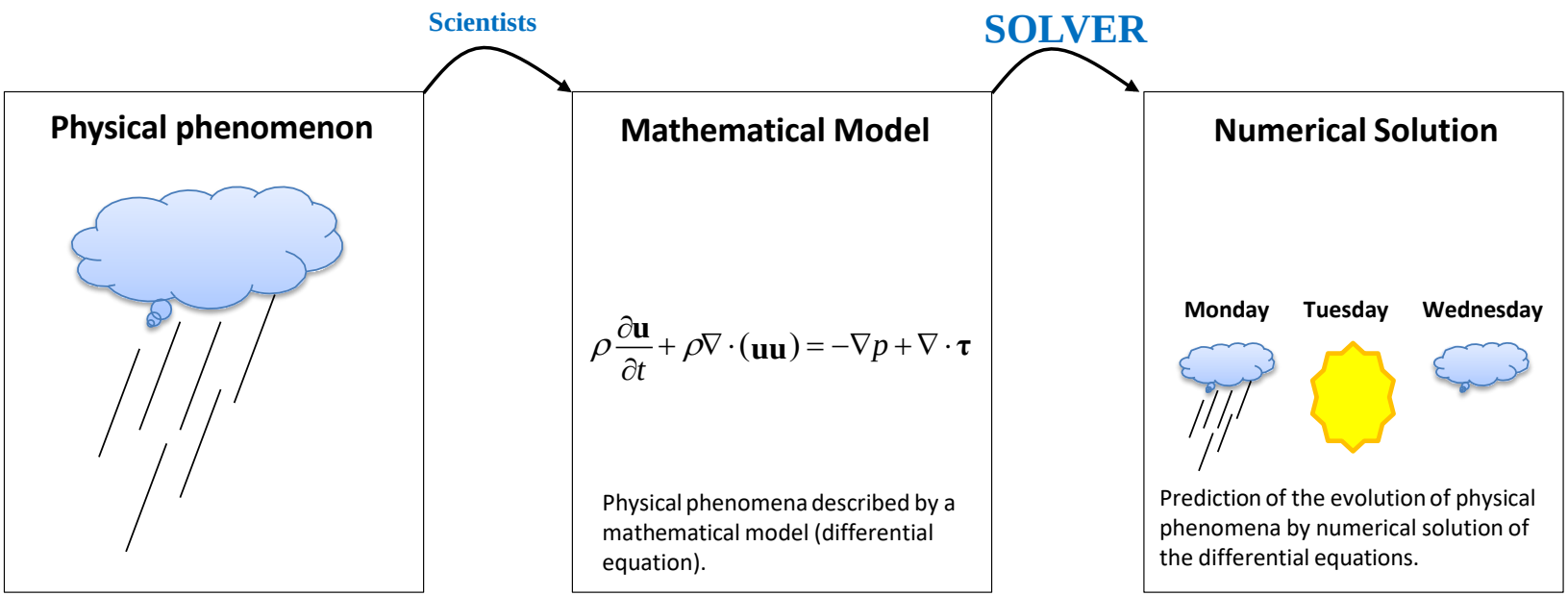
We previously discussed how to find the analytical solution to the equation

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

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

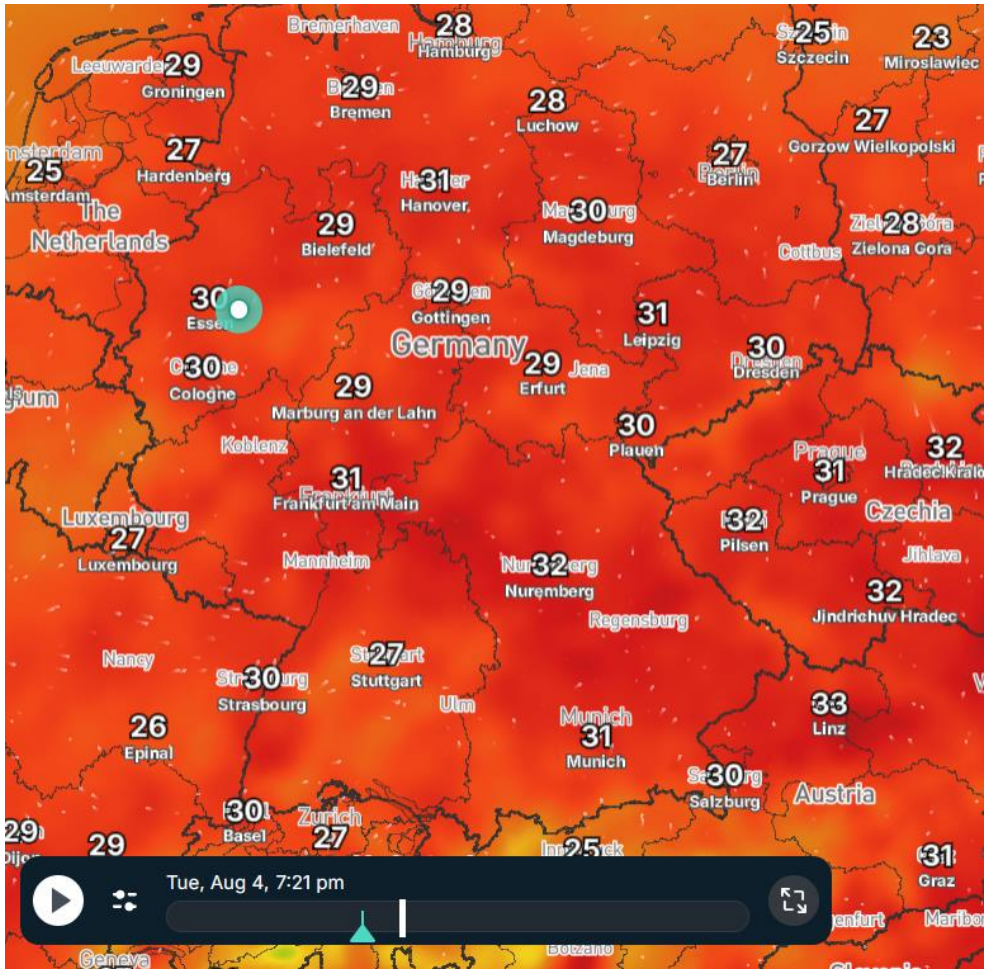
However, not all differential equations are this straightforward. In many cases, we must rely on **numerical solutions** to solve more complex equations!

Solving Differential Equations



Solving Differential Equations

Numerical Models



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,

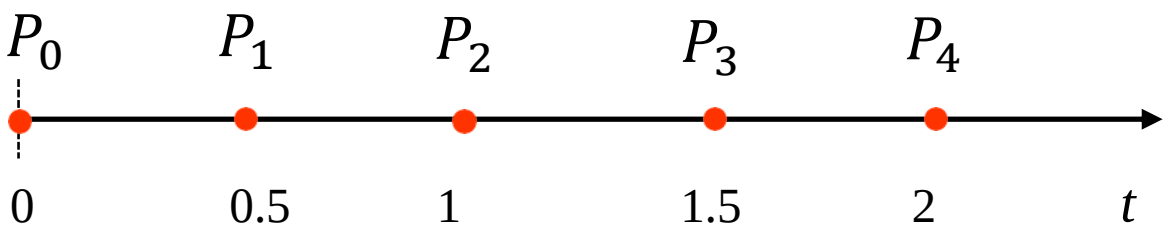
$$\frac{dP}{dt} = kP(t)$$

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

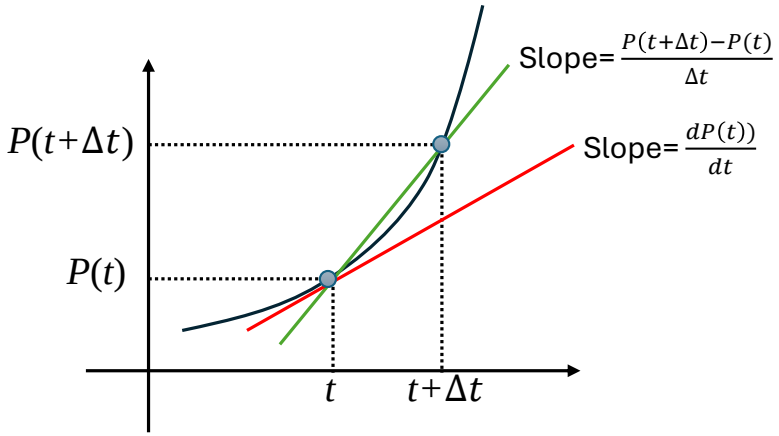
- Define an interval: $[0, 2]$
- Create a mesh: we will consider 5 mesh elements (4 intervals of 0.5) $\Delta t = 0.5$



- 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 $t_1 = 0.5,\ t_2 = 1,\ t_3 = 1.5,\ t_4 = 2$



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

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

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

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

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

Solving Differential Equations - FD

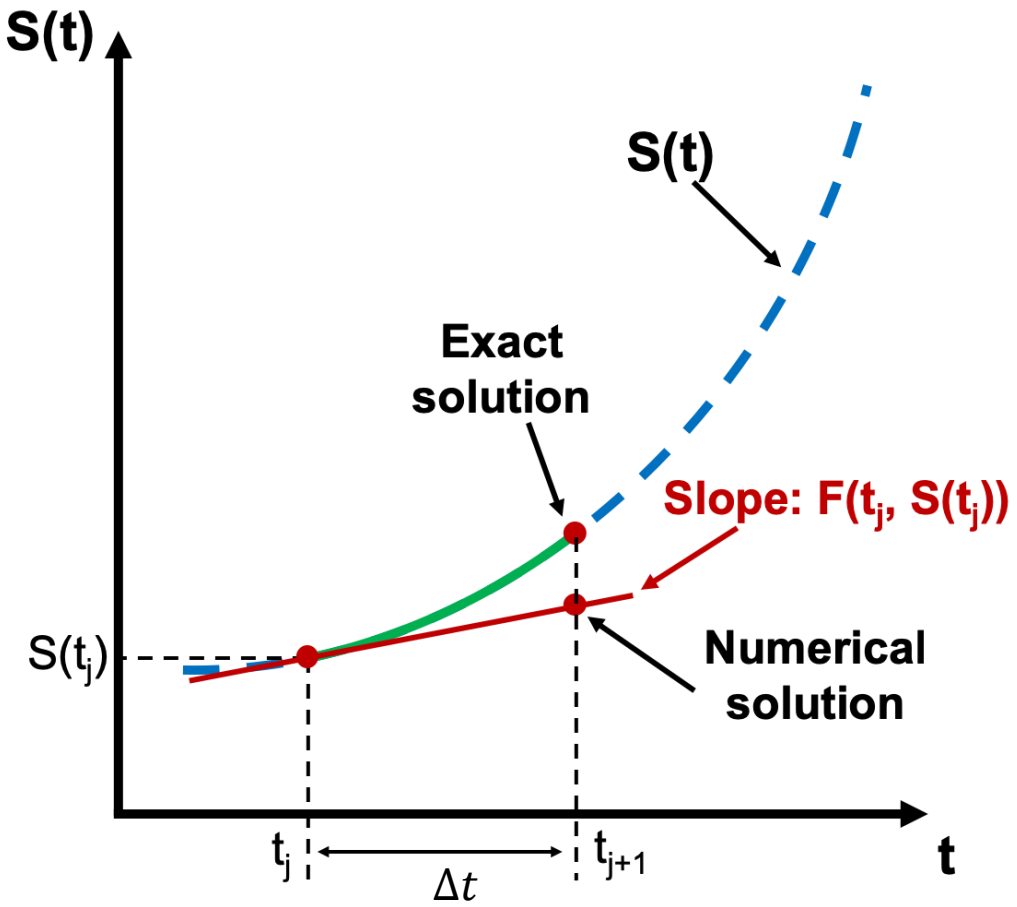
Let $\frac{dS(t)}{dt} = F(t, S(t))$ be an explicitly defined first order ODE. That is, F is a function that returns the derivative, or change, of a state given a time and state value. Also, let t_i be a numerical grid point of the interval $[t_0, t_f]$ with spacing Δt. Without loss of generality, we 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:

$$\begin{aligned}
\frac{P_1 - P_0}{\Delta t} &= kP_1 \\
\frac{P_2 - P_1}{\Delta t} &= kP_2 \\
\frac{P_3 - P_2}{\Delta t} &= kP_3 \\
\frac{P_4 - P_3}{\Delta t} &= kP_4
\end{aligned}$$

Solving Differential Equations - FD

- The system can be rewritten as:

$$(1 - k\Delta t)P_1 = P_0,$$

$$-P_1 + (1 - k\Delta t)P_2 = 0,$$
$$-P_2 + (1 - k\Delta t)P_3 = 0,$$
$$-P_3 + (1 - k\Delta t)P_4 = 0.$$

- In matrix form, for the unknowns $\mathbf{P} = [P_1, P_2, P_3, P_4]^T$, this becomes:

$$\begin{bmatrix} 1 - k\Delta t & 0 & 0 & 0 \\ -1 & 1 - k\Delta t & 0 & 0 \\ 0 & -1 & 1 - k\Delta t & 0 \\ 0 & 0 & -1 & 1 - k\Delta t \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix} = \begin{bmatrix} P_0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Solving Differential Equations - Hands On

Its 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**.
https://github.com/NNsDEsTutorial/NNsDEsTutorial.github.io

Solving Differential Equations - Hands On

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

```python
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

Its 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)
```

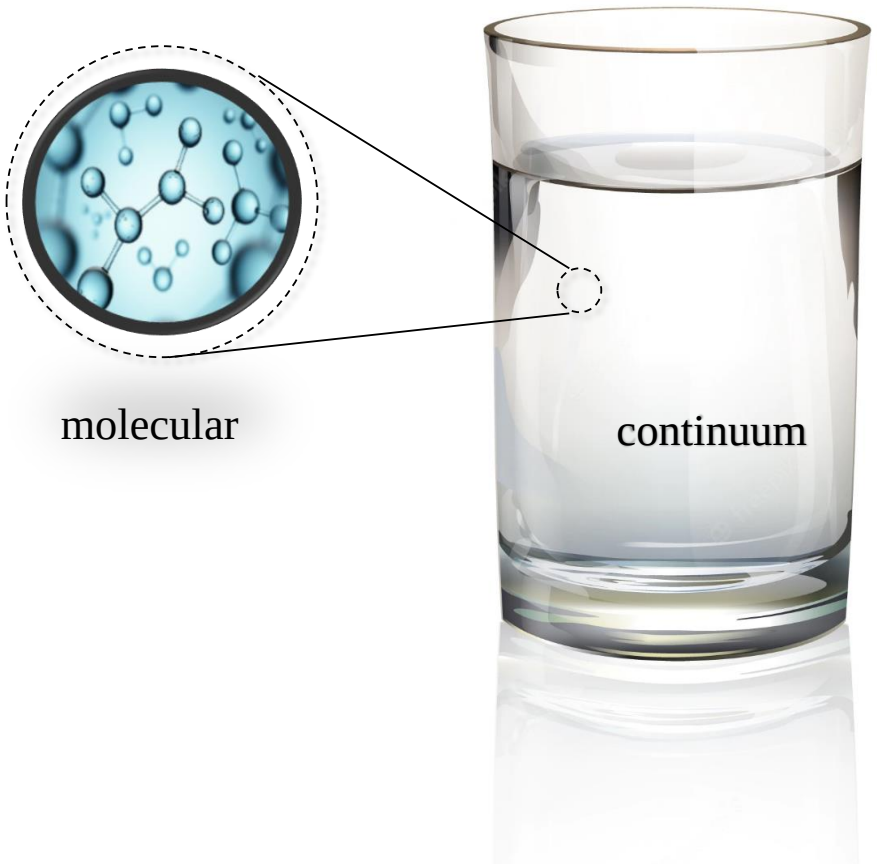
Consider a population of organisms that follows a logistic growth. The population size $P(t)$ at time t is governed by the following differential equation:

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

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**.

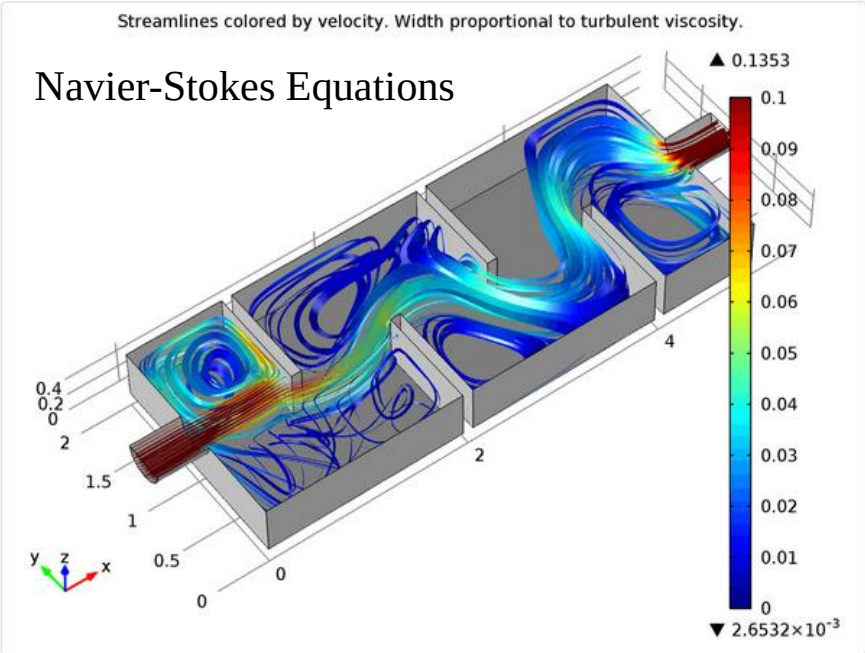
Differential Equations in Real Life

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

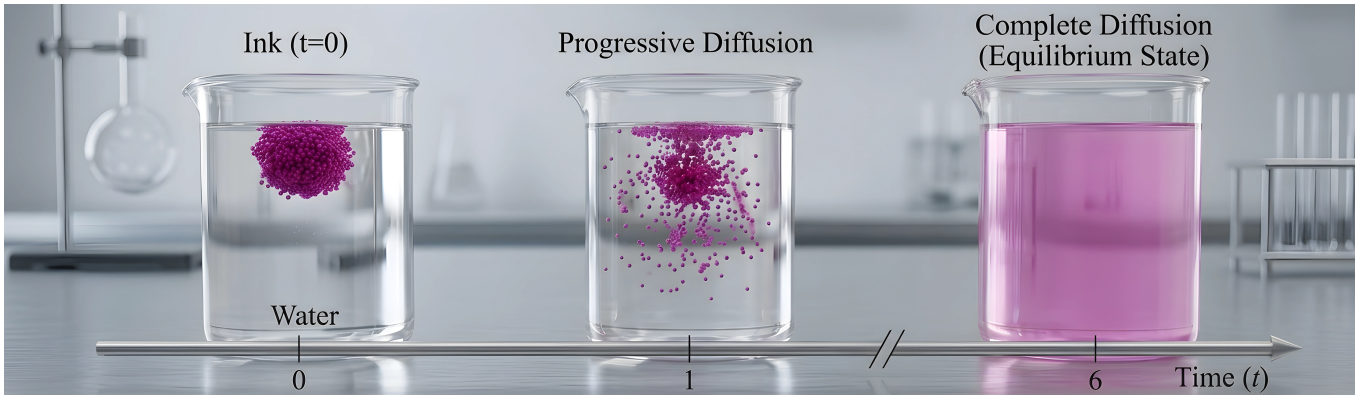
$$\nabla \cdot \overline{u} = 0$$

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

Navier-Stokes equations

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).



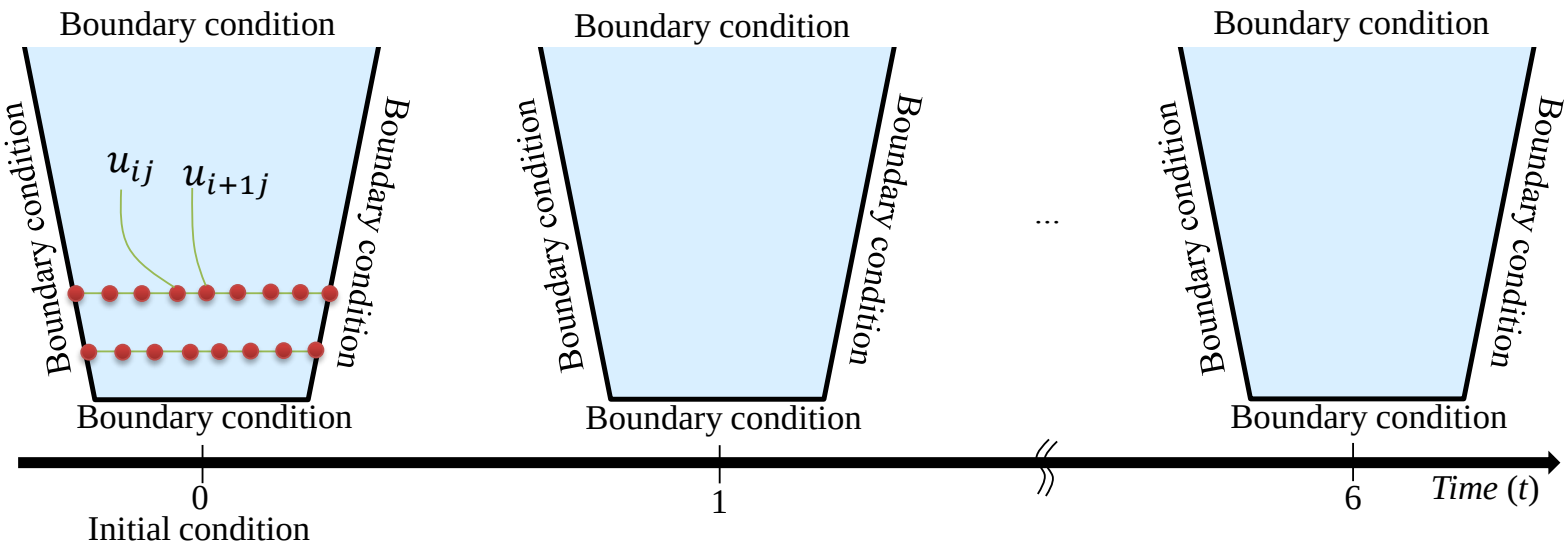
$$\frac{\partial u(t,x)}{\partial t} = D\frac{\partial^2 u(t,x)}{\partial x^2} \quad (1D) \tag{18}$$

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

Neural Networks for Solving DE

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

Challenges of Numerical Methods



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

Explicit

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

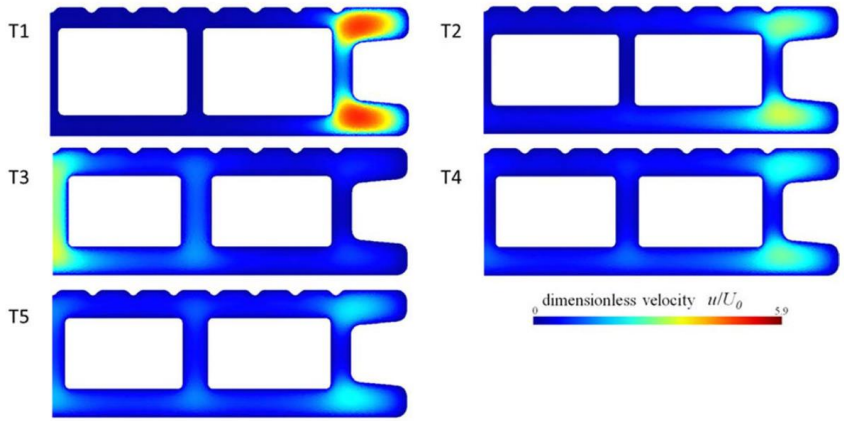
Implicit

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

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.



N. Gonçalves, P. Teixeira, L.L. Ferrás, A. Afonso, Innovative approach to the design of profile extrusion dies, Society of Plastic Engineers – Plastic Research Online (2014) 10.2417/spepro.005733.

Two Paradigms

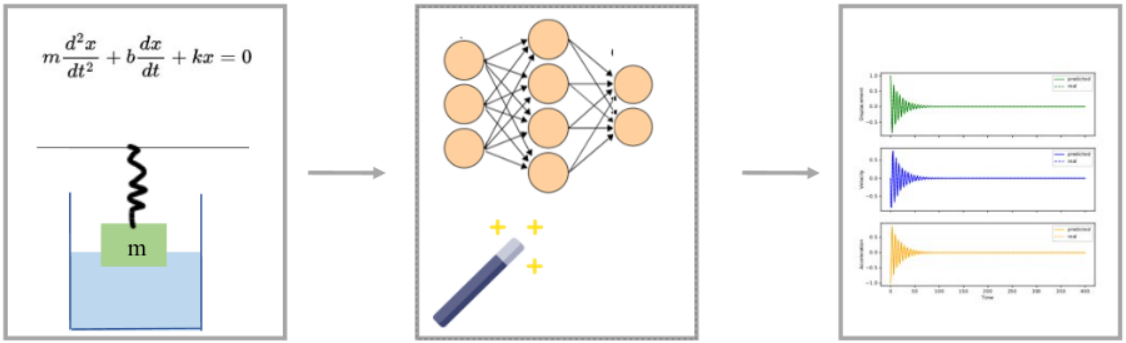
Solving differential equations

- 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 behavior is well-defined mathematically.

Modelling differential equations

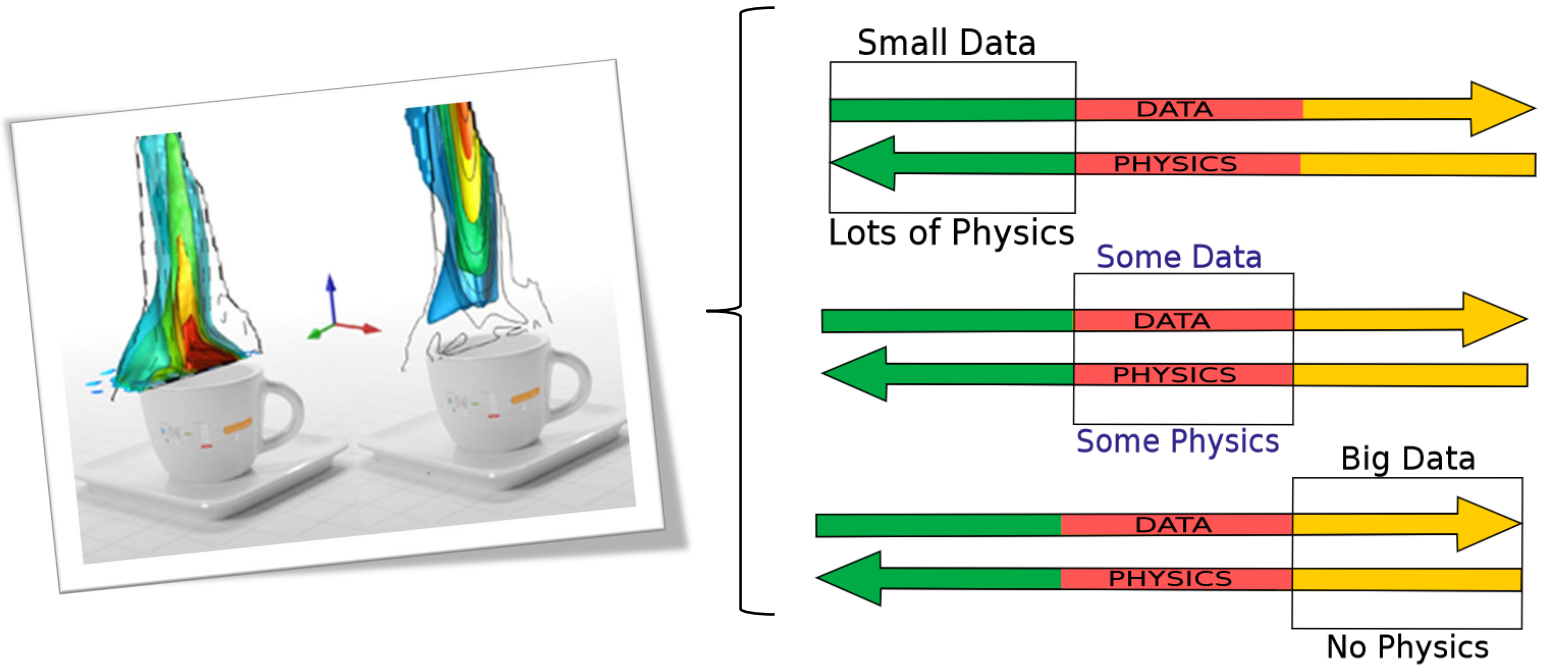
- Fully or partially unknown governing equations;
- Requires observed data to adjust the parameters or functions to match the data;
- Involves traditional optimization methods;
- Used for modeling systems in epidemiology, ecology, finance, and others.

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 [14];
- Physics-Informed Neural Networks (PINNs) use a neural network to approximate the curve of solutions [14];
- PINNs offer a faster and more cost-effective alternative to numerical methods when doing predictions [14].

Physics-Informed Neural Networks



Raissi, 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.

Data-driven Solution of PDEs

Given a known set of model parameters $\boldsymbol{\lambda}$, the forward problem asks:

How can we determine the unknown solution $\boldsymbol{u}(\boldsymbol{t}, \boldsymbol{x})$*?*

Classical numerical methods may become computationally demanding for:

- high-dimensional problems;
- complex geometries and multiphysics systems;
- real-time simulation and repeated evaluation;
- problems where mesh generation is difficult.

Physics-Informed Neural Networks (PINNs) provide a mesh-free, data-driven alternative by embedding the governing equations into the training loss.

PINN Approximation

Let the governing PDE be written as

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

PINNs approximate the unknown solution using a neural network

$$\widehat{u}(t,x;\boldsymbol{\theta}) \approx u(t,x), \tag{21}$$

where $\boldsymbol{\theta}$ denotes the trainable network parameters.
The network must simultaneously:

1. fit available observations or boundary data;
2. satisfy the governing differential equation.

Constrained PINN Formulation

For fixed model parameters $\boldsymbol{\lambda}$, the data-fitting problem can be formulated as

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

The objective minimizes the discrepancy between the network prediction and known data, subject to satisfaction of the PDE throughout the domain.

PDE Residual

Define the PDE residual by

$$r_{\mathrm{PDE}}(t, x; \boldsymbol{\theta}) = \frac{\partial \widehat{u}(t, x; \boldsymbol{\theta})}{\partial t} + \mathcal{N}[\widehat{u}(t, x; \boldsymbol{\theta}); \boldsymbol{\lambda}]. \tag{23}$$

A perfect solution satisfies

$$r_{\mathrm{PDE}}(t, x; \boldsymbol{\theta}) = 0$$

at every point in the domain.
In practice, the residual is evaluated at a finite set of collocation points

$$\{(t_i, x_i)\}_{i=1}^{N_{\mathrm{PDE}}}.$$

Unconstrained PINN Loss

The constrained problem is transformed into an unconstrained optimization problem using a penalty formulation:

$$\mathcal{L}(\boldsymbol{\theta}) = \mu_u \mathcal{L}_u(\boldsymbol{\theta}) + \mu_{\text{PDE}} \mathcal{L}_{\text{PDE}}(\boldsymbol{\theta}), \tag{24}$$

where

$$\mathcal{L}_u = \frac{1}{N_u} \sum_{n=1}^{N_u} \left(\widehat{u}_n - u_n\right)^2, \tag{25}$$

and

$$\mathcal{L}_{\text{PDE}} = \frac{1}{N_{\text{PDE}}} \sum_{i=1}^{N_{\text{PDE}}} r_{\text{PDE}}(t_i, x_i; \boldsymbol{\theta})^2. \tag{26}$$

Balancing the Loss Terms

The penalty parameters μ_u and μ_{PDE} control the relative contribution of the loss terms.

- If μ_u is too large, the network may focus primarily on fitting the data.
- If μ_{PDE} is too large, the network may satisfy the PDE while fitting the data poorly.
- The magnitudes of the loss terms may change during training.

Possible strategies include:

- manually selected fixed weights;
- hyperparameter optimization, for example with Optuna;
- adaptive penalty methods;
- two-stage training;
- filter-based optimization methods.

Example: Steady-state Heat Conduction

Consider steady-state heat conduction in a two-dimensional plate. The temperature field satisfies Laplace's equation:

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0, \qquad (x, y) \in \Omega. \tag{27}$$

The objective is to approximate $u(x, y)$ with a neural network

$$\widehat{u}(x, y; \boldsymbol{\theta})$$

while enforcing the Laplace equation throughout the domain.

Heat-Conduction Loss Function

For the heat-conduction problem, the PDE residual is

$$r_{\mathrm{PDE}}(x, y; \boldsymbol{\theta}) = \frac{\partial^2 \widehat{u}}{\partial x^2} + \frac{\partial^2 \widehat{u}}{\partial y^2}. \tag{28}$$

The PINN loss is

$$\mathcal{L}(\boldsymbol{\theta}) = \mu_u \mathcal{L}_u(\boldsymbol{\theta}) + \mu_{\mathrm{PDE}} \frac{1}{N_{\mathrm{PDE}}} \sum_{n=1}^{N_{\mathrm{PDE}}} r_{\mathrm{PDE}}(x_n, y_n; \boldsymbol{\theta})^2. \tag{29}$$

The first term enforces the boundary conditions, while the second term enforces the PDE inside the plate.

Boundary Conditions

Let

$$\Omega = [0, 1] \times [0, 1].$$

Consider the Dirichlet boundary conditions

$$\begin{aligned} u(0, y) &= 0, & u(1, y) &= 0, \\ u(x, 0) &= 0, & u(x, 1) &= \sin(\pi x). \end{aligned}$$

The analytical solution is

$$u(x, y) = \sin(\pi x) \frac{\sinh(\pi y)}{\sinh(\pi)}. \tag{30}$$

This solution satisfies both the Laplace equation and the prescribed boundary conditions.

Training Points

PINNs use two principal types of training points.

- **Collocation points:** sampled inside Ω to enforce the PDE residual.
- **Boundary points:** sampled on $\partial\Omega$ to enforce the prescribed boundary values.

At each collocation point,

$$r_{\mathrm{PDE}}(x_i, y_i; \boldsymbol{\theta})$$

is evaluated.
At each boundary point, the prediction is compared with the known value:

$$r_{\mathrm{BC}}(x_j, y_j; \boldsymbol{\theta}) = \widehat{u}(x_j, y_j; \boldsymbol{\theta}) - u_{\mathrm{BC}}(x_j, y_j).$$

Automatic Differentiation

The PDE residual requires derivatives of the neural network with respect to its inputs.
For example, the heat-conduction problem requires

$$\frac{\partial^2 \widehat{u}}{\partial x^2}, \qquad \frac{\partial^2 \widehat{u}}{\partial y^2}.$$

PINNs compute these derivatives using automatic differentiation:

- the network is represented as a computational graph;
- the chain rule is applied systematically;
- derivatives are computed with respect to the network inputs;
- higher-order derivatives are obtained by differentiating repeatedly.

This avoids finite-difference approximations of the network derivatives.

PINN Training Algorithm

Algorithm 1 Training a PINN

Require: Domain Ω, boundary $\partial\Omega$, N_{PDE}, N_u, neural network $\widehat{u}$

1: Initialize $\boldsymbol{\theta}$
2: **repeat**
3: Sample collocation points in Ω
4: Sample boundary points on $\partial\Omega$
5: Evaluate $\widehat{u}$ at all sampled points
6: Compute r_{PDE} using automatic differentiation
7: Compute $\mathcal{L}_{\text{PDE}}$ and $\mathcal{L}_u$
8: Compute

$$\mathcal{L} = \mu_{\text{PDE}}\mathcal{L}_{\text{PDE}} + \mu_u\mathcal{L}_u$$

9: Update $\boldsymbol{\theta}$ using gradient descent
10: **until** convergence or maximum number of iterations

Ensure: $\widehat{u}(\boldsymbol{\theta}^*; x, y)$

Implementation with DeepXDE

We implement the steady-state heat-conduction problem using DeepXDE with a PyTorch backend.

```
import os
os.environ["DDE_BACKEND"] = "pytorch"

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

Defining the Geometry

The plate is represented by the square domain

$$\Omega = [0, 1] \times [0, 1].$$

```
geom = dde.geometry.Rectangle(
    xmin=[0, 0],
    xmax=[1, 1]
)
```

The geometry is used to sample:

- collocation points in the interior;
- boundary points on the four edges.

Encoding the PDE

Laplace's equation is implemented through its residual:

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

    du_yy = dde.grad.hessian(
        u, x, i=1, j=1
    )

    return du_xx + du_yy
```

The function returns

$$r_{\text{PDE}} = \frac{\partial^2 \widehat{u}}{\partial x^2} + \frac{\partial^2 \widehat{u}}{\partial y^2}.$$

Encoding the Boundary Conditions

```python
def boundary_left(x, on_boundary):
    return on_boundary and np.isclose(x[0], 0.0)

def boundary_right(x, on_boundary):
    return on_boundary and np.isclose(x[0], 1.0)

def boundary_bottom(x, on_boundary):
    return on_boundary and np.isclose(x[1], 0.0)

def boundary_top(x, on_boundary):
    return on_boundary and np.isclose(x[1], 1.0)

bc_left = dde.icbc.DirichletBC(
    geom, lambda x: 0.0, boundary_left
)
bc_right = dde.icbc.DirichletBC(
    geom, lambda x: 0.0, boundary_right
)
bc_bottom = dde.icbc.DirichletBC(
    geom, lambda x: 0.0, boundary_bottom
)
bc_top = dde.icbc.DirichletBC(
    geom,
    lambda x: np.sin(np.pi * x[:, 0:1]),
    boundary_top
)

bcs = [bc_left, bc_right, bc_bottom, bc_top]
```

Constructing the PDE Data Object

```python
def u_exact(x):
    return (
        np.sin(np.pi * x[:, 0:1])
        * np.sinh(np.pi * x[:, 1:2])
        / np.sinh(np.pi)
    )

data = dde.data.PDE(
    geom,
    pde,
    bcs,
    num_domain=2000,
    num_boundary=400,
    num_test=1000,
    solution=u_exact
)
```

Here:

- `num_domain` specifies N_{PDE};
- `num_boundary` specifies N_u;
- `solution` is used only for evaluation.

Defining the Neural Network

We use a fully connected feed-forward network with:

- two input neurons for (x, y);
- four hidden layers with 50 neurons each;
- one scalar output neuron for $\widehat{u}$.

```
layer_sizes = [2] + [50] * 4 + [1]

net = dde.nn.FNN(
    layer_sizes,
    activation="tanh",
    kernel_initializer="Glorot uniform"
)
```

The $\tanh$ activation is smooth and differentiable, which is important when computing second-order derivatives.

Training the PINN

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

model.compile(
    "adam",
    lr=1e-3,
    metrics=["l2 relative error"]
)

model.train(iterations=10000)
```

Adam is effective for making rapid initial progress in the non-convex optimization landscape.

Fine-tuning with L-BFGS

After the Adam phase, the network can be fine-tuned using L-BFGS:

```
model.compile("L-BFGS")
model.train()
```

The two-stage strategy combines:

- the robust initial optimization of Adam;
- the fine-grained convergence of a quasi-Newton method.

The choice of optimiser and training schedule remains problem-dependent.

Evaluating the Trained Network

The trained PINN can be evaluated at arbitrary points.

```
nx, ny = 100, 100

x = np.linspace(0, 1, nx)
y = np.linspace(0, 1, ny)

X, Y = np.meshgrid(x, y)
XY = np.vstack(
    (X.flatten(), Y.flatten())
).T

u_pred = model.predict(XY)
u_true = u_exact(XY)

relative_l2_error = (
    np.linalg.norm(u_pred - u_true)
    / np.linalg.norm(u_true)
)
```

The analytical solution is used only to assess the prediction error.

Visualizing the Results

```
u_pred = u_pred.reshape(ny, nx)
u_true = u_true.reshape(ny, nx)
error = np.abs(u_pred - u_true)

fig, axs = plt.subplots(1, 3, figsize=(16, 4.5))

plots = [
    (u_pred, r"PINN prediction $\widehat{u}(x,y)$"),
    (u_true, r"Analytical solution $u(x,y)$"),
    (error, "Absolute error")
]

for ax, (field, title) in zip(axs, plots):
    image = ax.pcolormesh(
        X, Y, field,
        shading="auto",
        cmap="jet"
    )
    ax.set_title(title)
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_aspect("equal")
    fig.colorbar(image, ax=ax)

plt.tight_layout()
plt.show()
```

Visualizing the Results

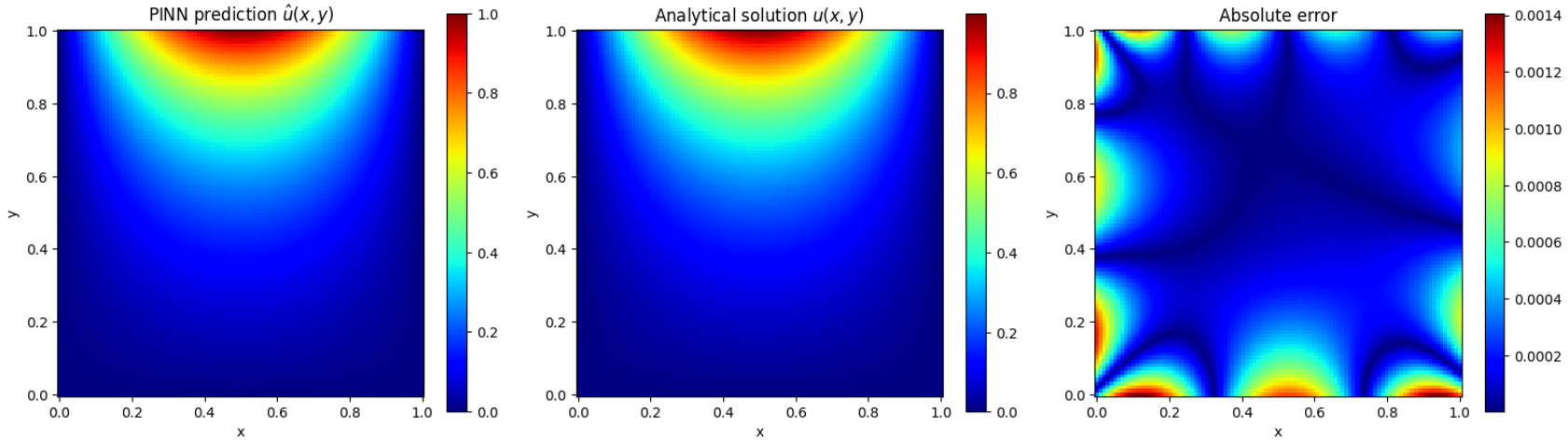


Figure 1: Output plots of training a PINN for the steady-state heat conduction in 2D, composed of: the PINN's prediction; the ground-truth analytical solution; their absolute difference.

Hands-On: Boundary Sampling

Consider the results in Figure 1. The trained PINN prediction, analytical solution, and absolute error are compared over the plate. The error is particularly high near the two bottom corners, forming ring-like contours in the rightmost panel.

1. Retrain the model after increasing the number of boundary points:

$$\texttt{num\_boundary} : 400 \longrightarrow 800.$$

   Keep all other settings fixed and regenerate Figure 1.

2. Compare the new error panel with the original one.
3. Does the error near the bottom corners decrease at the same rate as the overall relative L_2 error, or is one affected more strongly?

Data-driven Discovery of PDEs

In the forward problem, the governing equation and its parameters are known, and the objective is to approximate the solution.
In the inverse problem, we ask:

Given measurements of the system, how can we identify the unknown model parameters $\boldsymbol{\lambda}$?

The unknown parameters may represent:

- physical coefficients;
- source terms;
- boundary values;
- material properties;
- reaction or transport parameters.

Unknown PDE Parameters

Suppose the governing equation is

$$\mathcal{N}[u(t,x);\boldsymbol{\lambda}] = 0, \qquad (t,x) \in [0,T] \times \Omega, \tag{31}$$

where the parameter vector $\boldsymbol{\lambda}$ is unknown.
The solution is approximated by

$$\widehat{u}(t,x;\boldsymbol{\theta}) \approx u(t,x). \tag{32}$$

In an inverse PINN, both sets of variables are trainable:

$$\boldsymbol{\theta} \quad \text{and} \quad \boldsymbol{\lambda}.$$

Constrained Inverse PINN

The inverse problem can be formulated as

$$\begin{aligned} \underset{\boldsymbol{\theta} \in \mathbb{R}^{n_\theta},\, \boldsymbol{\lambda} \in \mathbb{R}^{n_\lambda}}{\text{minimize}} \quad & \mathcal{L}_u(\boldsymbol{\theta}) = \frac{1}{N_u} \sum_{n=1}^{N_u} \left(\widehat{u}(t_n, x_n; \boldsymbol{\theta}) - u_n\right)^2 \\ \text{subject to} \quad & \mathcal{N}[\widehat{u}(t, x; \boldsymbol{\theta}); \boldsymbol{\lambda}] = 0, \qquad (t, x) \in [0, T] \times \Omega. \end{aligned} \tag{33}$$

The objective minimizes the discrepancy between the predicted and observed solution while enforcing the PDE throughout the domain.

Information Used by an Inverse PINN

The available information is divided into three sets:

1. **Observed solution data:**

$$\{(t_n, x_n, u_n)\}_{n=1}^{N_{\text{observed}}}.$$

2. **Boundary and initial-condition data:**

$$\{(t_m, x_m, u_m)\}_{m=1}^{N_{\text{BC/IC}}}.$$

3. **PDE collocation points:**

$$\{(t_k, x_k)\}_{k=1}^{N_{\text{PDE}}}.$$

The observed data constrain the solution, while the PDE residual provides information about the unknown parameters.

Unconstrained Inverse PINN Loss

Using a penalty formulation, the constrained problem becomes

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\lambda}) = \mathcal{L}_u(\boldsymbol{\theta}) + \mu \mathcal{L}_{\text{PDE}}(\boldsymbol{\theta}, \boldsymbol{\lambda}), \tag{34}$$

where $\mathcal{L}_u = \mathcal{L}_{\text{observed}} + \mathcal{L}_{\text{BC/IC}}$.
The individual terms are

$$\mathcal{L}_{\text{observed}} = \frac{1}{N_{\text{observed}}} \sum_{n=1}^{N_{\text{observed}}} \left(\widehat{u}_n - u_n\right)^2,$$

$$\mathcal{L}_{\text{BC/IC}} = \frac{1}{N_{\text{BC/IC}}} \sum_{m=1}^{N_{\text{BC/IC}}} \left(\widehat{u}_m - u_m\right)^2.$$

Inverse PINN Training

The PDE residual is

$$r_{\text{PDE}} = \mathcal{N}[\widehat{u}(t, x; \boldsymbol{\theta}); \boldsymbol{\lambda}]. \tag{35}$$

The optimization updates both the network parameters and the physical parameters:

$$(\boldsymbol{\theta}^*, \boldsymbol{\lambda}^*) = \arg\min_{\boldsymbol{\theta}, \boldsymbol{\lambda}} \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\lambda}). \tag{36}$$

Thus, the PDE residual carries information about $\boldsymbol{\lambda}$ into the gradient-based optimization.

Example: Identifying Thermal Diffusivity

Consider the two-dimensional heat equation

$$\frac{\partial u}{\partial t} - \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = 0, \tag{37}$$

where the thermal diffusivity α is unknown.
The PINN approximates the temperature field by

$$\widehat{u}(t, x, y; \boldsymbol{\theta}),$$

while α is treated as an additional trainable variable.

Thermal Diffusivity: PDE Residual

For this problem, the PDE residual is

$$r_{\text{PDE}}(t, x, y; \boldsymbol{\theta}, \alpha) = \frac{\partial \widehat{u}}{\partial t} - \alpha \left(\frac{\partial^2 \widehat{u}}{\partial x^2} + \frac{\partial^2 \widehat{u}}{\partial y^2} \right). \tag{38}$$

The inverse PINN loss becomes

$$\mathcal{L}(\boldsymbol{\theta}, \alpha) = \mathcal{L}_{\text{observed}} + \mathcal{L}_{\text{BC/IC}} + \mu \mathcal{L}_{\text{PDE}}. \tag{39}$$

Both $\boldsymbol{\theta}$ and α are optimized simultaneously.

Heat Equation: Boundary and Initial Data

Consider the unit square

$$\Omega = [0, 1] \times [0, 1].$$

We impose homogeneous Dirichlet boundary conditions:

$$u(t, x, y) = 0, \qquad (x, y) \in \partial\Omega,$$

and the initial condition

$$u(0, x, y) = \sin(\pi x) \sin(\pi y). \tag{40}$$

The observed solution data are

$$\{(t_n, x_n, y_n, u_n)\}_{n=1}^{N_{\text{observed}}}.$$

Synthetic Observations

For testing purposes, synthetic measurements can be generated using

$$u(t, x, y) = e^{-2\pi^2 \alpha_{\text{true}} t} \sin(\pi x) \sin(\pi y). \tag{41}$$

The true diffusivity is used only to generate the observations:

$$\alpha_{\text{true}} = 0.4.$$

During training, the PINN does not receive α_{true}. It must recover the parameter from the measurements and the governing PDE.

Inverse PINN Training Algorithm

Algorithm 2 Training an inverse PINN

Require: N_{PDE}, observed data, BC/IC data, domain Ω, network $\widehat{u}(\boldsymbol{\theta})$, unknown parameters $\boldsymbol{\lambda}$

1: Initialize $\boldsymbol{\theta}$ and $\boldsymbol{\lambda}$
2: **repeat**
3: Sample collocation points in the domain
4: Draw batches of observed and BC/IC data
5: Evaluate $\widehat{u}$ at all points
6: Compute r_{PDE} using current $\boldsymbol{\lambda}$
7: Compute $\mathcal{L}_{\text{observed}}$
8: Compute $\mathcal{L}_{\text{BC/IC}}$
9: Compute $\mathcal{L}_{\text{PDE}}$
10: Set

$$\mathcal{L} = \mathcal{L}_{\text{observed}} + \mathcal{L}_{\text{BC/IC}} + \mu\mathcal{L}_{\text{PDE}}$$

11: Update $\boldsymbol{\theta}$ and $\boldsymbol{\lambda}$
12: **until** convergence

Ensure: $\widehat{u}(\boldsymbol{\theta}^*)$ and $\boldsymbol{\lambda}^*$

Implementation with DeepXDE

We implement the thermal-diffusivity identification problem using DeepXDE and a PyTorch backend.

```
import os
os.environ["DDE_BACKEND"] = "pytorch"

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

The inverse-problem implementation extends the forward-problem implementation by adding:

- an unknown trainable variable;
- observed solution data;
- joint optimization of network and physical parameters.

Defining the Space-Time Domain

The spatial domain is a unit square and the time interval is $[0, 1]$.

```
geom = dde.geometry.Rectangle(
    xmin=[0, 0],
    xmax=[1, 1]
)

timedomain = dde.geometry.TimeDomain(0, 1)

geomtime = dde.geometry.GeometryXTime(
    geom,
    timedomain
)
```

DeepXDE represents the input points using the coordinate ordering

$$(x, y, t).$$

Declaring the Unknown Parameter

The thermal diffusivity is introduced as a trainable variable:

```
alpha = dde.Variable(0.5)
```

The value 0.5 is only an initial guess.
The parameter becomes identifiable only after it is:

1. included in the PDE residual;
2. passed to the optimiser as an external trainable variable.

Encoding the Heat Equation

The input columns are ordered as (x, y, t).

```
def pde(x, u):
    du_t = dde.grad.jacobian(
        u, x, i=0, j=2
    )

    du_xx = dde.grad.hessian(
        u, x, i=0, j=0
    )

    du_yy = dde.grad.hessian(
        u, x, i=1, j=1
    )

    return du_t - alpha * (du_xx + du_yy)
```

The residual uses the current estimate of α at every training iteration.

Boundary and Initial Conditions

We prescribe zero temperature on all boundaries:

```
def boundary(x, on_boundary):
    return on_boundary

bc = dde.icbc.DirichletBC(
    geomtime,
    lambda x: 0.0,
    boundary
)
```

The initial temperature profile is $u(0, x, y) = \sin(\pi x) \sin(\pi y)$.

```
def init_func(x):
    return (
        np.sin(np.pi * x[:, 0:1])
        * np.sin(np.pi * x[:, 1:2]))

ic = dde.icbc.IC(
    geomtime,
    init_func,
    lambda _, on_initial: on_initial)
```

Loading Observed Data

In a real application, measurements may be stored in a file with columns (x, y, t, u).

```
raw_data = np.loadtxt(
    "observed_data.csv",
    delimiter=",",
    skiprows=1
)

observe_x = raw_data[:, 0:3]
observe_u = raw_data[:, 3:4]

observe_bc = dde.icbc.PointSetBC(
    observe_x,
    observe_u,
    component=0
)
```

The observed data can be located anywhere in the space-time domain.

Generating Synthetic Observations

For a study case, we use the analytical solution
$u(t, x, y) = e^{-2\pi^2 \alpha_{\text{true}} t} \sin(\pi x) \sin(\pi y)$.

```
alpha_true = 0.4

def u_exact(x):
    xx = x[:, 0:1]
    yy = x[:, 1:2]
    tt = x[:, 2:3]

    return (
        np.exp( -2 * np.pi**2
            * alpha_true * tt)
        * np.sin(np.pi * xx)
        * np.sin(np.pi * yy) )

N_observed = 200
observe_x = geomtime.random_points(N_observed)
observe_u = u_exact(observe_x)

observe_bc = dde.icbc.PointSetBC(
    observe_x,
    observe_u,
    component=0)
```

The value α_{true} is not used during training.

Constructing the Inverse PINN Data

The observed data are included together with the boundary and initial conditions.

```
data = dde.data.TimePDE(
    geomtime,
    pde,
    [bc, ic, observe_bc],
    num_domain=2000,
    num_boundary=100,
    num_initial=100,
    anchors=observe_x,
    num_test=1000
)
```

The loss contains:

- PDE residual error from interior points;
- boundary and initial-condition error;
- observed-data error from `PointSetBC`.

The `anchors` argument ensures that the observation points are included in the training data.

Defining the Neural Network

The network now receives three input variables:

$$(x, y, t).$$

```
layer_sizes = [3] + [50] * 4 + [1]

net = dde.nn.FNN(
    layer_sizes,
    activation="tanh",
    kernel_initializer="Glorot uniform"
)
```

The output is the predicted temperature

$$\widehat{u}(x, y, t; \theta).$$

Joint Training of θ and α

The unknown diffusivity must be explicitly passed to the optimiser.

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

variable = dde.callbacks.VariableValue(
    alpha,
    period=1000,
    filename="alpha_history.dat")

model.compile(
    "adam",
    lr=1e-3,
    external_trainable_variables=[alpha])

model.train(
    iterations=20000,
    callbacks=[variable])
```

The optimiser updates both $\boldsymbol{\theta}$ and α.

Fine-tuning with L-BFGS

After the Adam phase, the inverse PINN can be fine-tuned using L-BFGS.

```
model.compile(
    "L-BFGS",
    external_trainable_variables=[alpha])

model.train(
    callbacks=[variable])
```

The callback records the evolution of the estimated diffusivity during training. Inverse problems may require more iterations than forward problems because the network must:

- fit the observations;
- satisfy the PDE;
- identify the unknown parameter.

Evaluating the Recovered Parameter

After training, the learnt diffusivity can be compared with the known value used to generate the synthetic data.

```
print(
    f"Estimated diffusivity: "
    f"{float(alpha):.4f}"
)

print(
    f"True diffusivity: "
    f"{alpha_true:.4f}"
)
```

The parameter-identification error is

$$e_\alpha = \left|\alpha - \alpha_{\text{true}}\right| . \tag{42}$$

Tracking Parameter Convergence

The `VariableValue` callback stores the parameter history.

```
import re

iterations = []
alpha_history = []

with open("alpha_history.dat") as f:
    for line in f:
        values = re.findall(r"[-+]?\d*\.?\d+" r"(?:[eE][-+]?\d+)?", line)

        iterations.append( int(float(values[0])))
        alpha_history.append( float(values[1]))

iterations = np.array(iterations)
alpha_history = np.array(alpha_history)

plt.figure(figsize=(6, 4))
plt.plot(iterations,
    alpha_history,
    label=r"Estimated $\alpha$" )
plt.axhline(alpha_true,
    color="black",
    linestyle="--",
    label=r"True $\alpha$" )

plt.xlabel("Iteration")
plt.ylabel(r"$\alpha$")
plt.legend()
plt.tight_layout()
plt.show()
```

Tracking Parameter Convergence

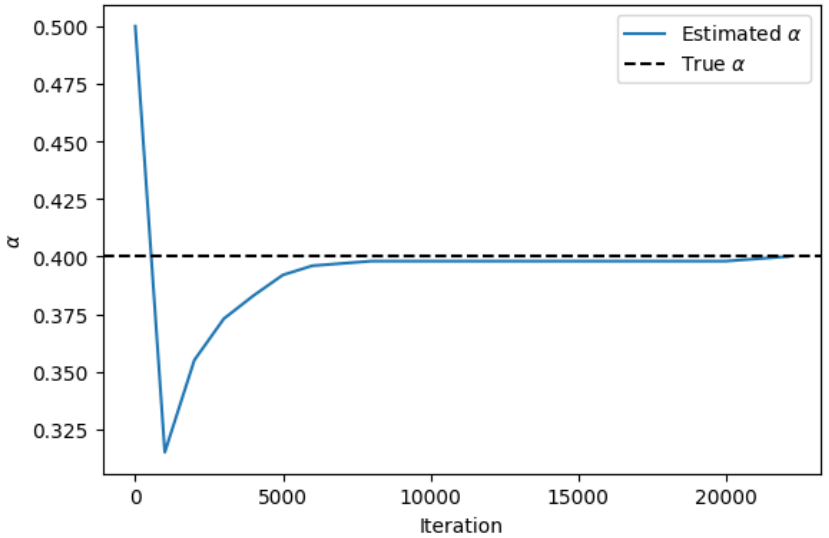


Figure 2: Plot of the evolution of the estimate for the diffusivity α over training (solid line) and it's true value (dashed line) for comparison.

Visualizing the Temperature Field

The trained network can be evaluated at a fixed time, for example $t = 0.5$.

```
nx, ny = 100, 100

x = np.linspace(0, 1, nx)
y = np.linspace(0, 1, ny)

X, Y = np.meshgrid(x, y)

t_fixed = 0.5 * np.ones((nx * ny, 1))

XYT = np.hstack((
    X.reshape(-1, 1),
    Y.reshape(-1, 1),
    t_fixed
))

u_pred = model.predict(XYT)
u_true = u_exact(XYT)

u_pred = u_pred.reshape(ny, nx)
u_true = u_true.reshape(ny, nx)

error = np.abs(u_pred - u_true)
```

Comparing the Reconstructed Solution

```
fig, axs = plt.subplots(
    1, 3,
    figsize=(16, 4.5))

fields = [( u_pred, r"PINN prediction $\widehat{u}(x,y,t=0.5)$" ),
    ( u_true, r"Analytical solution $u(x,y,t=0.5)$" ),
    ( error, "Absolute error" ) ]

for ax, (field, title) in zip(axs, fields):
    image = ax.pcolormesh(
        X, Y, field,
        shading="auto",
        cmap="jet" )

    ax.set_title(title)
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_aspect("equal")
    fig.colorbar(image, ax=ax)

plt.tight_layout()
plt.show()
```

Comparing the Reconstructed Solution

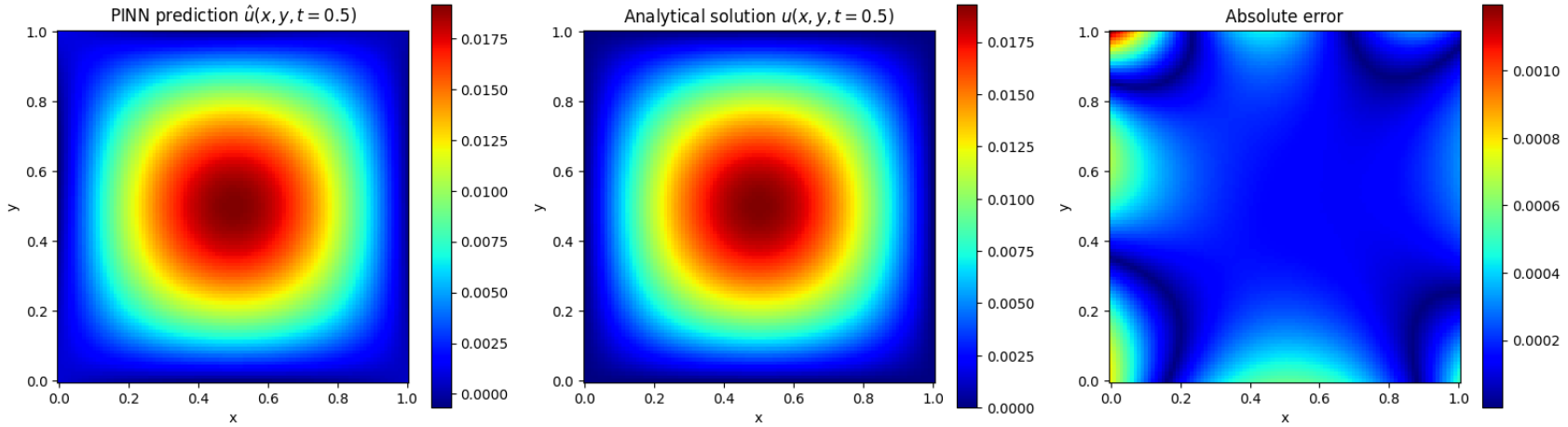
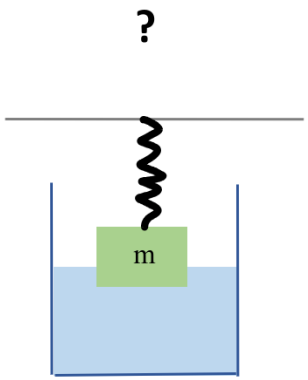


Figure 3: Output of training a PINN for the thermal diffusivity identification problem, composed of the PINN's prediction, the analytical solution, and their absolute difference at $t = 0.5$.

Neural Networks for Modelling DE

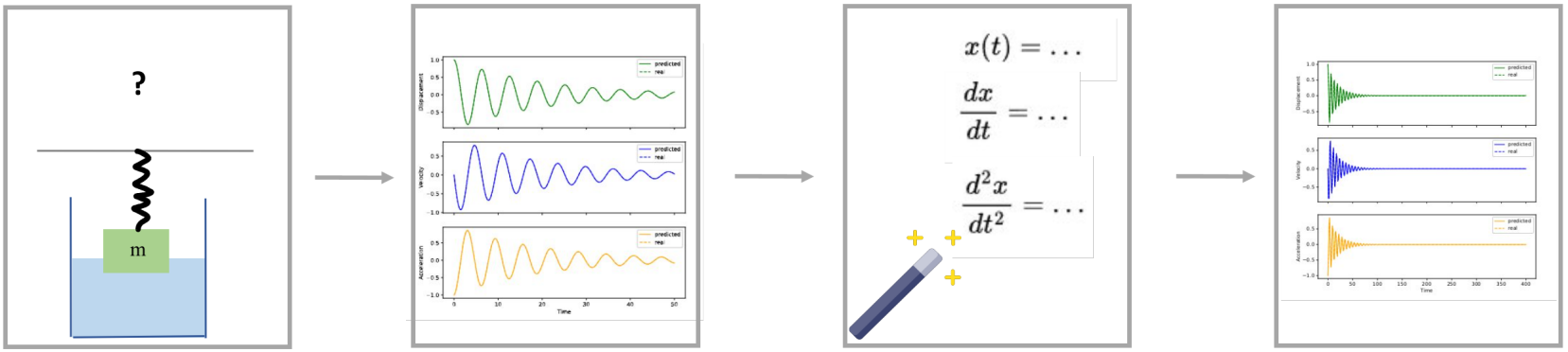
- Challenges on Differential Equations Formulation for Describing Real-world Systems
- Neural Ordinary Differential Equations
- Hands-on with the Torchdiffeq Library

Challenges on Formulating DEs



- Often the equations that model systems are unknown;
- When experimental data is available, mathematical models can be fitted;
- Traditional techniques require expert-knowledge and are a ”trial and error” process.

Neural Networks for Modelling DEs



- Neural networks are universal approximators [7];
- Experimental data can be used to train a neural network to fit the data;
- However, they fit time-independent functions to data [4];
- Data has to be regularly-sampled for training [4];
- Neural Ordinary Differential Equations adjust a time-dependent function to data, an ODE [4].

Neural Ordinary Differential Equations

In Residual Networks, we consider the following transformation of a hidden state from layer t to layer $t+1$:

$$\boldsymbol{h}_{t+1} = \boldsymbol{h}_t + \boldsymbol{f}_t(\boldsymbol{h}_t, \boldsymbol{\theta}_t), \tag{43}$$

where $\boldsymbol{h}_t \in \mathbb{R}^d$ is the hidden state at layer t, $\boldsymbol{\theta}_t$ represents the parameters of the network determined by the learning process (the weights and biases), and $\boldsymbol{f}_t : \mathbb{R}^d \to \mathbb{R}^d$ is a differentiable function. Assuming we have a finite number of layers, for the residual forward problem presented in (43) to be stable, it is recommended to control the variation of $\boldsymbol{f}_t(\boldsymbol{h}_t, \boldsymbol{\theta}_t)$ across iterations. Therefore, introducing a positive constant δ, one can control the variation of $\boldsymbol{f}_t$:

$$\boldsymbol{h}_{t+1} = \boldsymbol{h}_t + \delta \boldsymbol{f}_t(\boldsymbol{h}_t, \boldsymbol{\theta}_t). \tag{44}$$

Neural Ordinary Differential Equations

This equation can be rewritten as:

$$\frac{\boldsymbol{h}_{t+1} - \boldsymbol{h}_t}{\delta} = \boldsymbol{f}_t(\boldsymbol{h}_t, \boldsymbol{\theta}_t), \tag{45}$$

and taking the limit $\delta \to 0$, we obtain the following ODE,

$$\frac{d\boldsymbol{h}(t)}{dt} = \boldsymbol{f}(t, \boldsymbol{h}(t), \boldsymbol{\theta}(t)). \tag{46}$$

defined over a certain time interval $t \in [t_0, T]$ with $T > t_0$. Note that while it is indeed true that the parameter δ provides a means to regulate the stability of the forward problem (44), the stability is not solely determined by δ but also influenced by the variations in the weights.

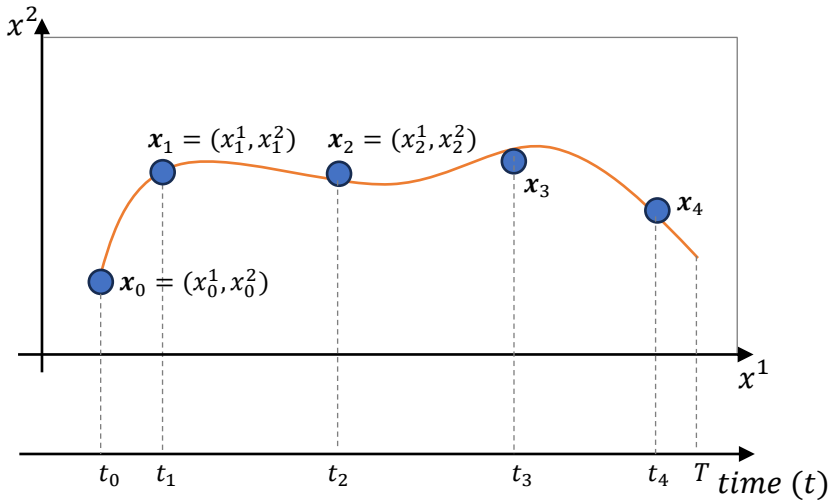
Eq. (46) extends the discrete nature of the Residual Network, Eq. (44), to a continuous model.

Neural Ordinary Differential Equations

Assume we have a collection of ordered data ($N+1$ ordered observations) $\boldsymbol{x} = \{\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$, which represent the state of some dynamical system at discrete instants t_i over the time interval $[t_0, T]$ (with $t_N = T$). Each $\boldsymbol{x}_i = (x_i^1, x_i^2, \ldots, x_i^d) \in \mathbb{R}^d$, $i = 0, \ldots, N$ is associated with instant t_i.
We assume that the given data can be modelled by the initial value problem in Eq. (47). This allows us to determine the behaviour of the dynamical system at any instant within the interval $[t_0, T]$ [4, 5].

$$\begin{cases} \dfrac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{f}(t, \boldsymbol{x}(t)) \\ \boldsymbol{x}(t_0) = \boldsymbol{x}_0, \quad t \in [t_0, T]. \end{cases} \tag{47}$$

Neural Ordinary Differential Equations



Data obtained experimentally or a priori from a certain dynamical system

Fit to the data obtained by solving the initial value problem:

$$\begin{cases} \dfrac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{f}(t, \boldsymbol{x}(t)) \\ \boldsymbol{x}(t_0) = \boldsymbol{x}_0, \qquad t \in [t_0, T] \end{cases}$$

Figure 4: Fit of an ODE to data $\{\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_4\}$ obtained experimentally or provided by a dynamical system. The blue symbols represent the data points, while the orange line represents the fit obtained from the initial value problem shown on the right. Each vector $\boldsymbol{x}_i$ corresponds to a specific instant t_i. The initial value problem allows us to determine the behaviour of the dynamical system at any instant within the interval $[t_0, T]$ [4, 5].

Neural Ordinary Differential Equations

The problem is that neither the solution $\boldsymbol{x}(t) \in \mathbb{R}^d$ nor the function $\boldsymbol{f}(t, \boldsymbol{x}(t)) : \mathbb{R} \times \mathbb{R}^d \to \mathbb{R}^d$ are known (the ODE may be linear, nonlinear, etc).

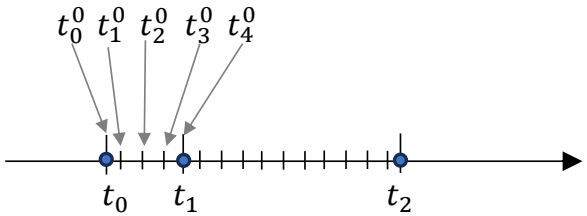
Neural ODEs provide a viable solution to approximate the initial value problem (47) using only the data $\boldsymbol{x} = \{\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ (the ground truth in our Neural ODE).
Let $\boldsymbol{h}(t)$ be an approximation of $\boldsymbol{x}(t)$.

$$\begin{cases} \dfrac{d\boldsymbol{h}(t)}{dt} = \boldsymbol{f}_{\boldsymbol{\theta}}(t, \boldsymbol{h}(t)), \\ \boldsymbol{h}(t_0) = \boldsymbol{x}_0 \quad t \in [t_0, T]. \end{cases} \tag{48}$$

The right-hand side's analytical expression $(\boldsymbol{f}(t, \boldsymbol{x}(t)))$ is replaced by a NN (denoted by $\boldsymbol{f}_{\boldsymbol{\theta}}(t, \boldsymbol{h}(t))$), where $\boldsymbol{\theta}$ represents the weights and biases of the network [4, 5].

Neural Ordinary Differential Equations

To illustrate the Neural ODE, we assume that the numerical method used to solve the initial value problem (48) is the explicit Euler method. A mesh is defined for each interval $[t_i, t_{i+1}]$, $i = 0, \ldots, N-1$. Therefore, given the initial condition $\boldsymbol{h}(t_0) = \boldsymbol{x}_0$, a (uniform) mesh $\{t_m^i = m\Delta t_i : m = 0, 1, ..., M_i\}$ on an interval $[t_i, t_{i+1}]$ with some integer M_i and $\Delta t := (t_{i+1} - t_i)/M_i$, we compute the numerical solution as (for the interval $[t_0, t_1]$) [4, 5],



$$\hat{\boldsymbol{h}}(t_1^0) = \boldsymbol{x}_0 + \Delta t \boldsymbol{f}_{\boldsymbol{\theta}}(t_0^0, \boldsymbol{x}_0)$$
$$\hat{\boldsymbol{h}}(t_2^0) = \hat{\boldsymbol{h}}(t_1^0) + \Delta t \boldsymbol{f}_{\boldsymbol{\theta}}(t_1^0, \hat{\boldsymbol{h}}(t_1^0))$$
$$\vdots$$
$$\hat{\boldsymbol{h}}(t_{M_0}^0) = \hat{\boldsymbol{h}}(t_{M-1}) + \Delta t \boldsymbol{f}_{\boldsymbol{\theta}}(t_{M-1}, \hat{\boldsymbol{h}}(t_{M-1}).$$

Neural Ordinary Differential Equations

We can say that the Neural ODE consists of two main components: a numerical ODE solver, and the neural network $\boldsymbol{f_\theta}(t, \boldsymbol{h}(t))$ [4, 5].

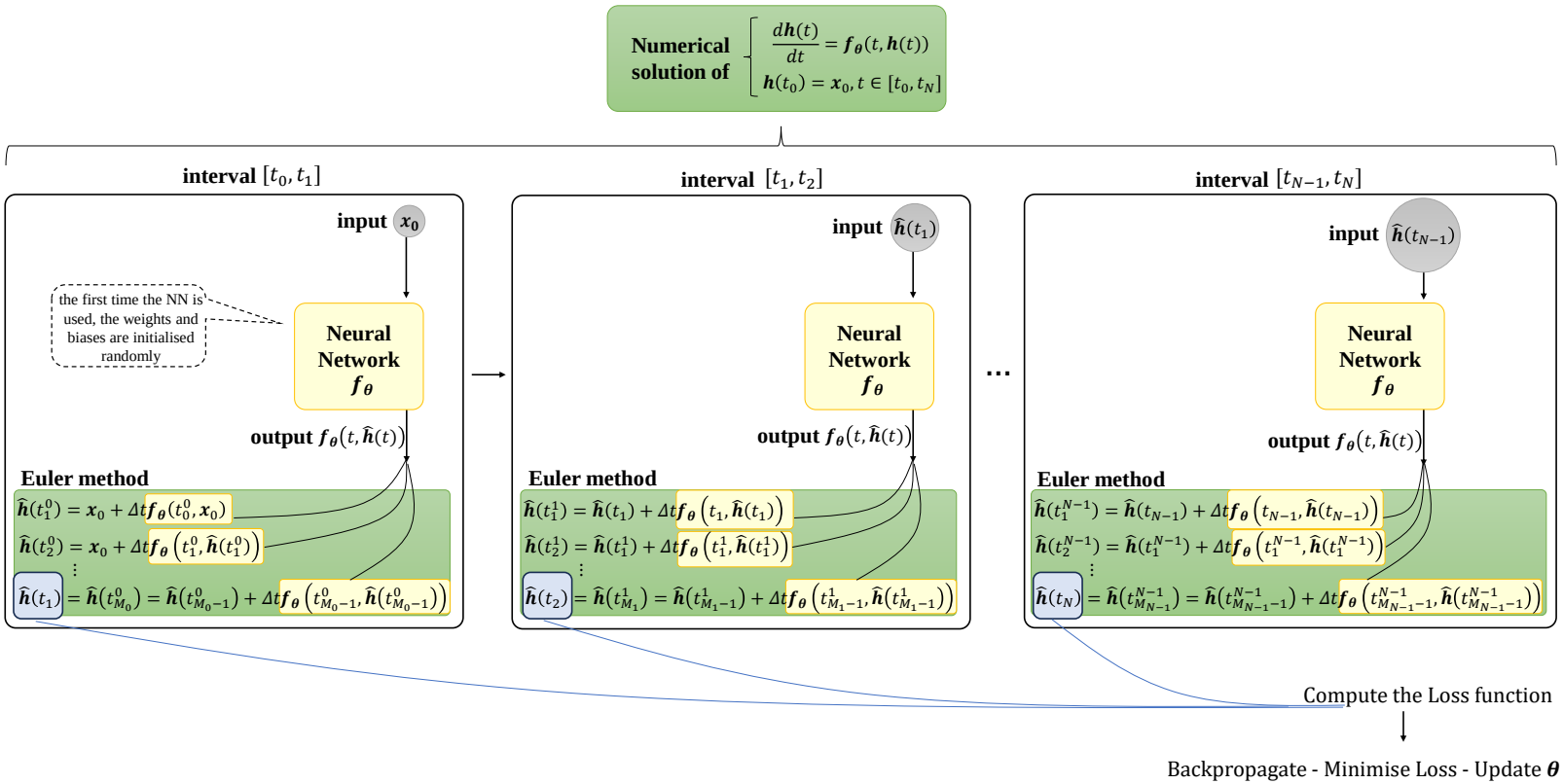


Figure 5: Schematic of a Neural ODE iteration. Note that along the sequence of figures (left to right) the NN $\boldsymbol{f_\theta}(t, \boldsymbol{h}(t))$ doesn't change.

Neural Ordinary Differential Equations

We can use both adaptive and fixed-step ODE solvers. When the step size is explicitly specified Δt, the discretization takes place for each sub-interval of observations $[t_i, t_{i+1}]$ with the specified step size yielding $(t_{i+1} - t_i)/\Delta t$ time steps [4, 5].

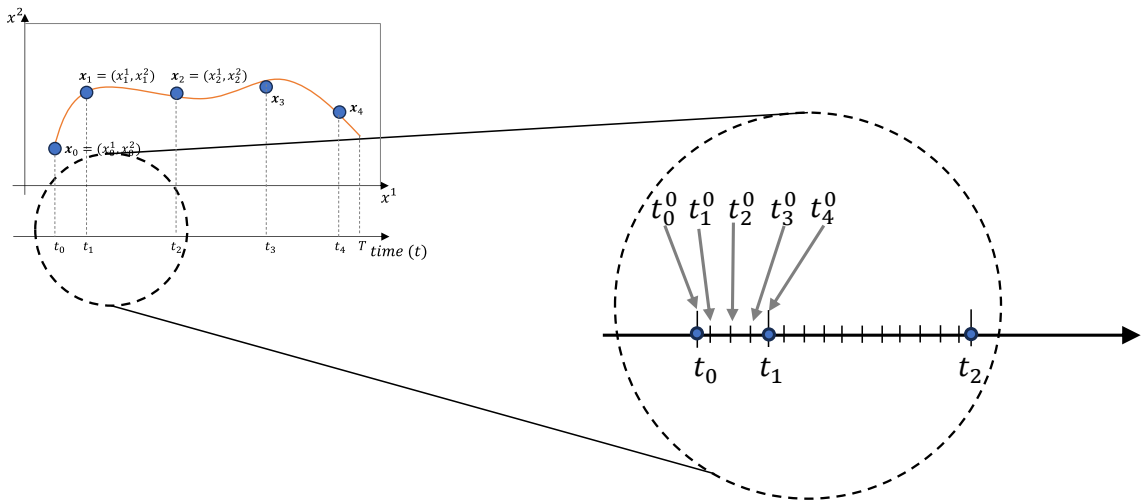


Figure 6: Example of a typical mesh used in the numerical solution of Eq. (48) for each interval $[t_i, t_{i+1}]$, $i = 0, \ldots, N-1$, where t_i is the time of observation $\boldsymbol{x}_i$ [5].

Neural Ordinary Differential Equations

Different numerical solvers can be used to obtain the numerical solution of (48), for the time being, we will refer to a numerical solver as ODESolve. Assuming $i = 1, \ldots, N$ and that that t_N corresponds to T, each state $\boldsymbol{h}(t_i)$ is then numerically given by [4, 5],

$$\hat{\boldsymbol{h}}(t_i) = \text{ODESolve}(\boldsymbol{f}_{\boldsymbol{\theta}}, \boldsymbol{x}_0, \{t_1, \ldots, t_N\}). \tag{49}$$

Example - Population Growth

Consider you have experimental data from a population growth. Here, we will use the synthetic data fabricated earlier. We want to model the dynamics using a Neural ODE [4, 5].
First, the torchdiffeq and torch modules are imported [5, 3]:

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchdiffeq import odeint
```

With this Neural ODE we will be able to predict the population P at each time instant given the initial condition:

```
true_y0 = torch.tensor([2.518629])
```

Example - Population Growth

Next, we define a neural network to approximate the right-hand side of the ODE [5, 3]:

```
class ODEFunc(nn.Module):
    def __init__(self):
        super(ODEFunc, self).__init__()

        self.net = nn.Sequential(nn.Linear(1, 50),
            nn.Tanh(), nn.Linear(50,50),
            nn.ELU(), nn.Linear(50, 1))

        for m in self.net.modules():
            if isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, mean=0, std=0.1)
                nn.init.constant_(m.bias, val=0)

    def forward(self, t, y):
        return self.net(y)
```

Example - Population Growth

Then we compile the network and choose an optimiser:

```
func = ODEFunc()
optimiser = optim.Adam(func.parameters(), lr=1e-5)
```

We define the time instants in which we want to know the solutions [5, 3]:

```
t = torch.linspace(0., 1, 500)
```

Example - Population Growth

Now, we have specified the network, optimiser and we have training data available. Then we define the training loop:

```
        for itr in range(1, 2000):
            pred_y = odeint(func, true_y0, t, method='rk4')
            loss = nn.MSELoss()(pred_y, true_y)

            optimiser.zero_grad()
            loss.backward()
            optimiser.step()
            for itr % 100 == 0:
                print('Iter {:04d}|Loss {:.6f}'.format(itr, loss))
```

Here we specify the numerical method to be a fixed-step one and print the loss value every 100 iterations [5, 3].

Neural Operators

Operator Learning Framework

Fourier Neural Operators

Applications

One model per problem

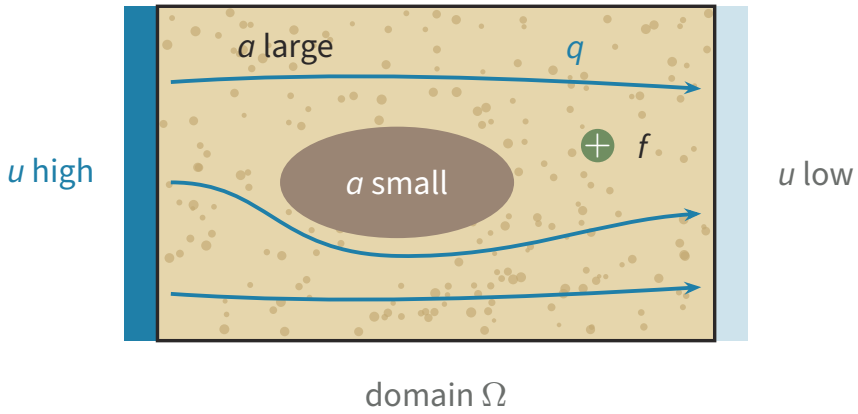
So far, every model we trained solves **one** problem instance.

- A PINN fits *one* solution u for *one* coefficient, boundary condition and forcing.
- Change the material, the load, or the initial condition $\Rightarrow$ **re-train from scratch**.
- Change the grid or resolution $\Rightarrow$ re-train again.
- A trained MLP/CNN is a map $\mathbb{R}^n \to \mathbb{R}^m$ — its size is *baked into the training grid*.

*We keep re-solving problems that only differ in their inputs. Can we learn the **solver** once, for a whole family?*

Example: Darcy Flow

Steady flow through a **porous medium** — groundwater in soil, oil in rock. We will come back to it for the rest of this part.



$a(x)$ **permeability** — the medium. Large in sand, tiny in clay.

$u(x)$ **pressure** — what we solve for. Fixed on the boundary.

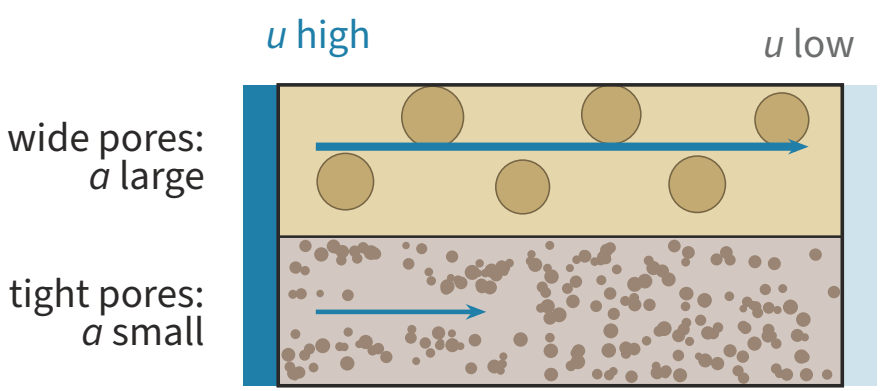
$q(x)$ **flux** — how much fluid moves, and where to.

$f(x)$ **source** — injection or rainfall.

A *different medium a* gives a *different pressure u* — and classically, each one costs a fresh solve.

Deriving the Darcy equation

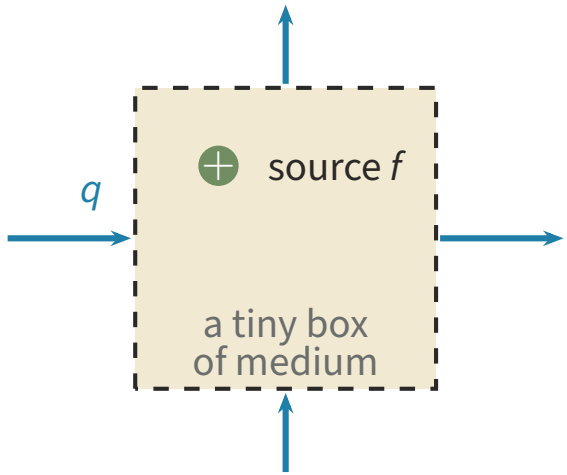
1. Darcy's experiment (1856)



$$q = -a(x)\,\nabla u(x)$$

same pressure drop, more permeable medium $\Rightarrow$ more flow

2. nothing gets lost



$$\nabla \cdot q(x) = f(x)$$

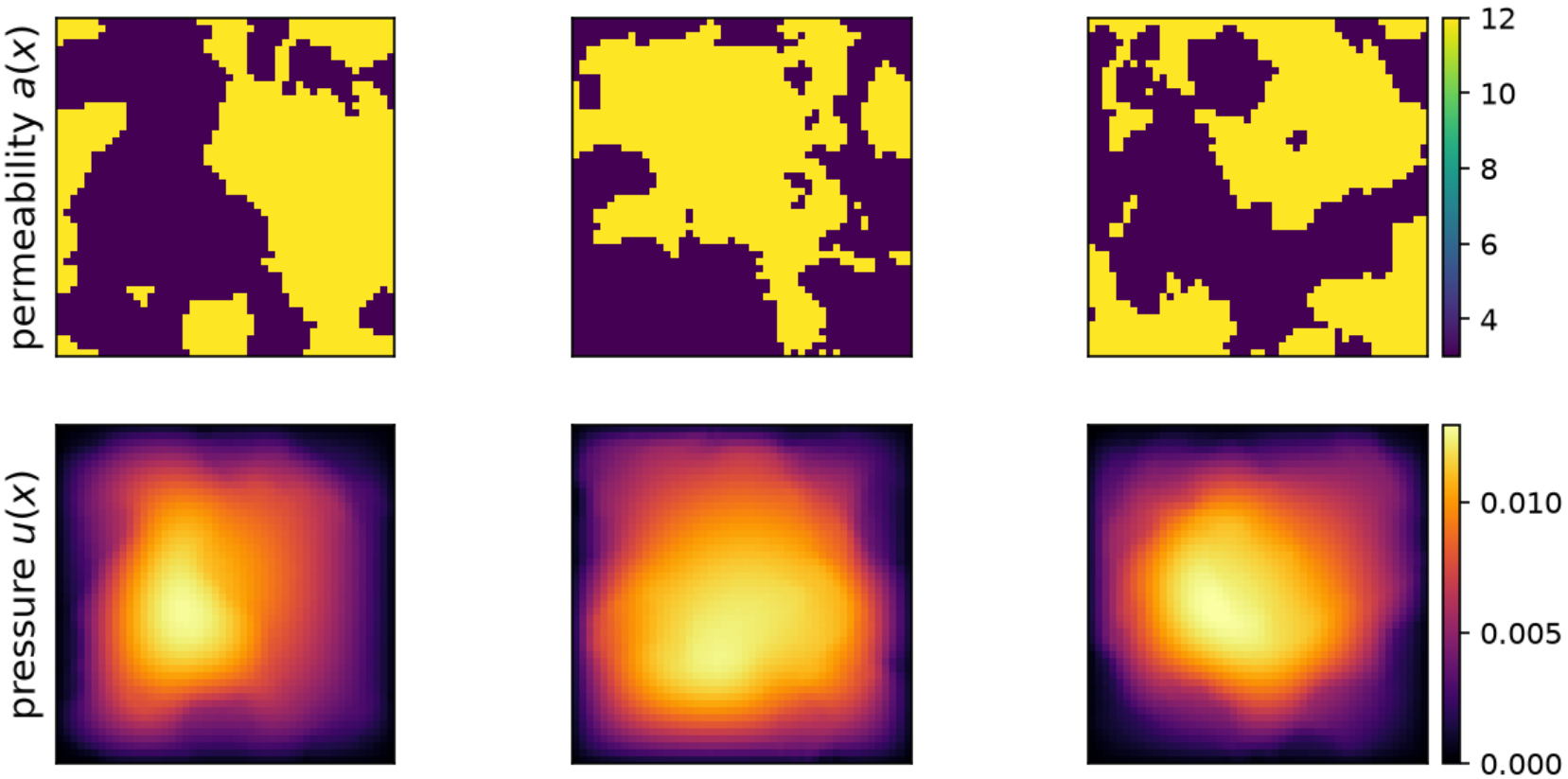
what flows in must flow out, up to what the source injects

Substituting **1** into **2** gives the **Darcy equation**:

$$\boxed{-\nabla \cdot \big(a(x)\,\nabla u(x)\big) = f(x)} \qquad u|_{\partial\Omega} = 0.$$

Constant $a \Rightarrow$ Poisson; the point here is that $a(x)$ **varies in space**, *inside* the divergence.

Solutions for many media



Each two-phase permeability field $a(x)$ (top) yields a pressure field $u(x)$ (bottom), from a finite-difference solver. A neural operator learns this map $a \mapsto u$ once, for the *whole family*.

How would a PINN solve this?

Represent the solution as a network of the *coordinates*, $u(x) \approx \widehat{u}(x;\boldsymbol{\theta})$, and fit $\boldsymbol{\theta}$ so the PDE residual and the boundary condition vanish:

$$r(x;\boldsymbol{\theta}) = \nabla \cdot \big(a(x)\,\nabla \widehat{u}(x;\boldsymbol{\theta})\big) + f(x),$$

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N_r}\sum_i \big|r(x_i;\boldsymbol{\theta})\big|^2 + \frac{\lambda}{N_b}\sum_j \big|\widehat{u}(x_j;\boldsymbol{\theta})\big|^2.$$

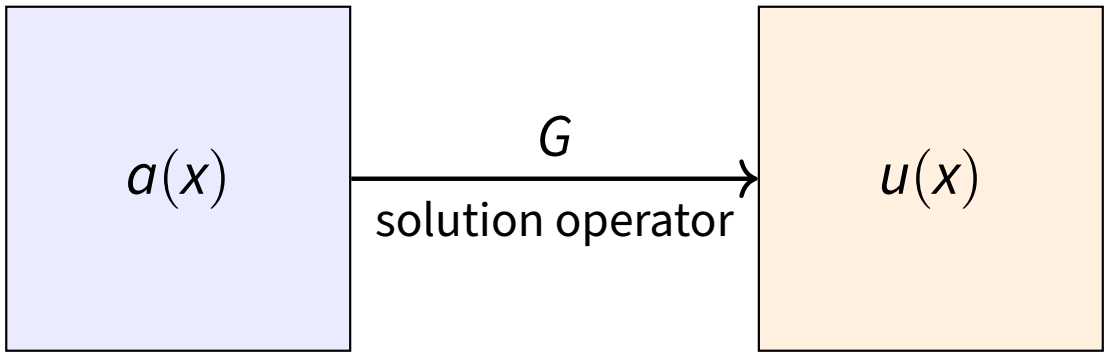
- The permeability $a(x)$ is **baked into the loss** — $\widehat{u}$ is fitted for this *one* field.
- The input is the coordinate x, **not** a: nothing tells the network about other media.
- Change a, f, or the boundary data $\Rightarrow$ the loss changes $\Rightarrow$ **retrain from scratch**.

*A PINN learns **one** solution $\widehat{u}$ — not the operator $a \mapsto u$.*

Solving a family of problems

A classical solver (or a PINN) takes **one** permeability $a(x)$ and computes **one** pressure $u(x)$. Change $a(x)$ — solve again.

We would instead like a *single* model that maps **any** permeability field directly to its pressure field — a map from the function a to the function u:



This map G is the **solution operator** of the PDE. Both its input *and* its output are whole **functions**, not fixed-length vectors.

Functions are infinite-dimensional

- A vector $x \in \mathbb{R}^n$ is a list of n numbers $(x_1, \ldots, x_n)$ — one entry per index i.
- A function a is like a vector with one "entry" $a(x)$ for *every* point $x \in \Omega$ — **infinitely many**.
- Admissible functions form a **function space** $\mathcal{A}$, e.g. square-integrable fields $L^2(\Omega)$ or continuous fields $C(\Omega)$.

A function space is still a *vector space* — we can add functions and scale them — but it is **infinite-dimensional**.

Think of a as a point in an ∞-dimensional space.

What is an operator?

An **operator** is the natural object when inputs and outputs are functions:

a **function**	maps numbers to numbers	$f : \mathbb{R} \to \mathbb{R}$
a **matrix**	maps vectors to vectors	$W : \mathbb{R}^n \to \mathbb{R}^m$
an **operator**	maps functions to functions	$G : \mathcal{A} \to \mathcal{U}$

Operators you already know:

- differentiation $\frac{d}{dx} : u \mapsto u'$,
- integration $u \mapsto \int_0^{(\cdot)} u(s)\, ds$,
- the **solution operator** of a PDE $a \mapsto u$.

Operator learning, formally

The PDE defines a **solution operator** between function spaces,

$$G^\dagger : \mathcal{A} \to \mathcal{U}, \qquad a \mapsto u,$$

which we do not know in closed form. Given data pairs $\{(a_i, u_i)\}_{i=1}^N$ (from a solver or from measurements), we learn a parametric operator

$$G_\theta \approx G^\dagger \qquad [8].$$

Operator learning is regression — but the inputs and outputs are ***functions****, and $\mathcal{A}, \mathcal{U}$ are infinite-dimensional.*

From matrices to kernels

A standard network layer on vectors is $x \mapsto \sigma(Wx + b)$. To act on *functions* we need the analogue of the **matrix multiply**.

In $\mathbb{R}^n$ a linear map sums over indices:

$$(Wv)_i = \sum_{j=1}^{n} W_{ij}\, v_j.$$

Its continuous analogue replaces the **sum by an integral** and the **matrix by a kernel** $\kappa(x, y)$:

$$(\mathcal{K}v)(x) = \int_{\Omega} \kappa(x, y)\, v(y)\, dy.$$

This **integral operator** is the linear layer of a neural operator — and it is *global*: every value $v(y)$ can influence the output at x.

A neural operator layer

Assemble the same ingredients as an MLP layer, now acting on functions:

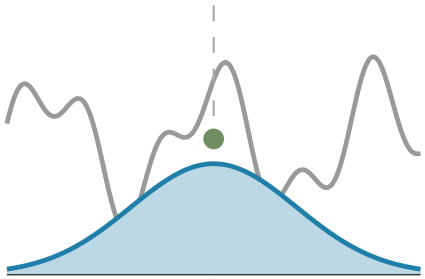
$$v_{\ell+1}(x) = \sigma\Big(\underbrace{W v_\ell(x)}_{\text{local (pointwise)}} + \underbrace{\int_\Omega \kappa_\theta(x,y)\, v_\ell(y)\, dy}_{\text{global (integral operator)}} + b(x) \Big).$$

- W — pointwise linear mixing of channels (a 1×1 layer),
- integral term — the learned **global** coupling; κ_θ is learned,
- b, σ — bias and nonlinearity, exactly as usual.

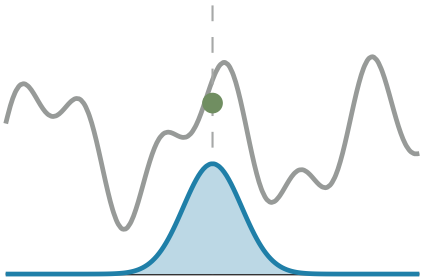
Stack L such layers, with a lift in and a projection out.

The pointwise term

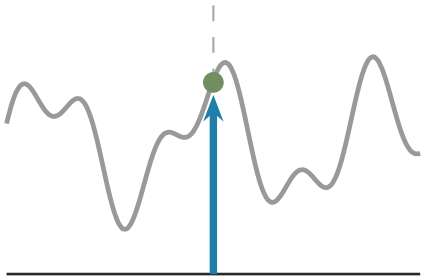
The integral already mixes globally — so why keep $Wv(x)$ at all? Read the integral as a **weighted average** of v, with $\kappa_\theta(x_i, \cdot)$ as the weight profile. Narrow the weight and the average becomes local:



wide weight
a broad average



narrower
a local average



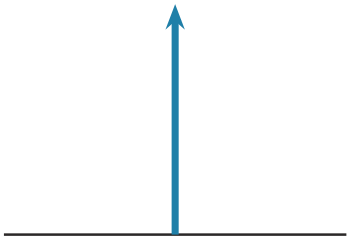
δ
exactly $v(x_i)$

Push it to the end: a weight with **all** its mass at x_i does no averaging at all — it just reads off the value. So "use the value right here" is *already* an integral operator, whose kernel is a Dirac δ:

$$W\,v(x) \;=\; \int_\Omega W\,\delta(x-y)\,v(y)\,dy.$$

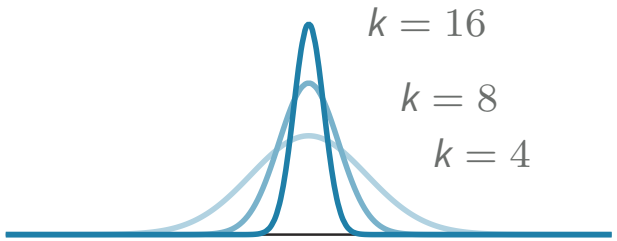
The Dirac kernel

So in the continuum the local term is redundant — *if* kernels may be distributions. They may not: κ_θ is a function.



$\delta(x-y)$

$\neq$



the sharpest κ_θ a k-mode family can build: width $\sim 1/k$

No choice of θ turns a bump into a spike, so the δ-component is supplied by hand — that is $Wv(x)$. (It is not only the *learnable* families that are too small: an L^2 kernel gives a compact operator, and the identity is not compact.) Lift and project are pointwise on purpose: because they mix only channels, never locations, they work at **any** discretisation.

Vectors and functions

Vector network	**Neural operator**
input $x \in \mathbb{R}^n$	input function $v \in \mathcal{A}$
matrix multiply Wx	integral $\int \kappa(x, y)v(y)\, dy$
weight entry W_{ij}	kernel $\kappa(x, y)$
bias b	bias function $b(x)$
nonlinearity $\sigma(\cdot)$	pointwise $\sigma(\cdot)$
fixed dimension n	**any resolution**

A neural operator is an MLP whose linear layers are integral operators.

Discretisation invariance

Couldn't we just flatten $a(x)$ on a grid and train an MLP/CNN?

- Yes — but then W has a shape **tied to that grid**.
- Refine or change the grid $\Rightarrow$ the matrix no longer matches $\Rightarrow$ retrain.
- The kernel $\kappa(x, y)$ is defined **continuously**, independent of any grid; discretise it on whatever mesh you like.

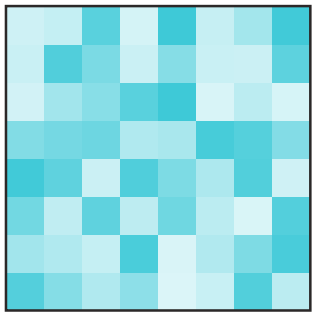
So the *same* weights act on **any** sampling of the input: train cheaply on a **coarse** grid, evaluate on a **fine** one (ideally zero-shot), and reuse one model across a whole **family** of inputs.

Discretising the integral

A grid turns the layer's integral into a quadrature sum:

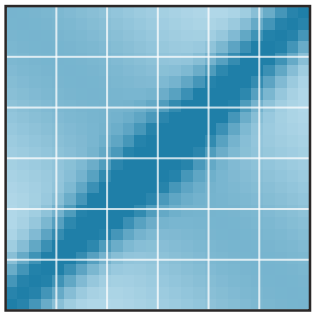
$$\int_\Omega \kappa_\theta(x_i, y)\, v(y)\, dy \;\approx\; \sum_{j=1}^{N} \underbrace{w_j\, \kappa_\theta(x_i, y_j)}_{K_{ij}^{(N)}}\, v(y_j) \;=\; \left(K^{(N)} v\right)_i, \qquad w_j = \frac{|\Omega|}{N}$$

dense layer: $M \in \mathbb{R}^{N \times N}$



the entries **are** the parameters
N^2 of them, N fixed

operator layer: $\kappa_\theta(x, y)$

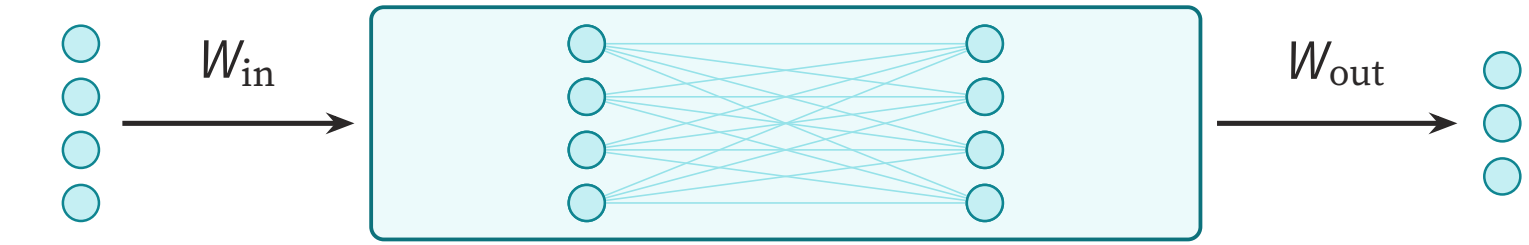


θ is the parameter; the grid
only says *where to sample*

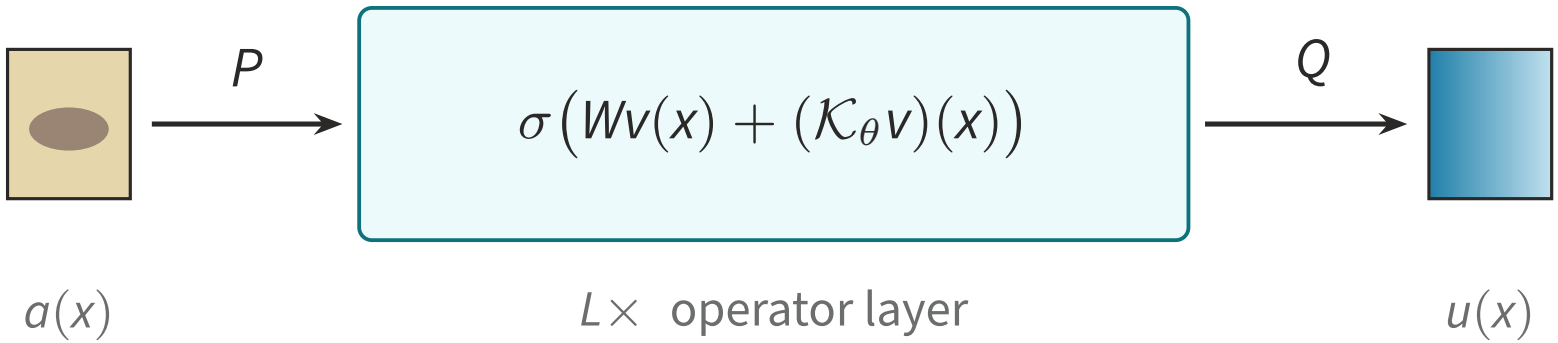
On any grid the layer is a matrix multiply — but the dense layer **stores** that matrix, while the operator **evaluates** it from the same θ, at whatever resolution arrives.

The neural operator template

Feed-forward net — vectors



Neural operator — functions



Same architecture, same three stages. What changes is **what flows through it**: vectors in $\mathbb{R}^n$ become **functions** on Ω, so the matrix multiply becomes an integral operator.

Neural Operators

Operator Learning Framework

Fourier Neural Operators

Applications

Which kernel should we learn?

The template left one choice open: how to *write down* the kernel in

$$v_{\ell+1}(x) = \sigma\Big(W v_\ell(x) + \int_\Omega \kappa_\theta(x, y)\, v_\ell(y)\, dy + b\Big).$$

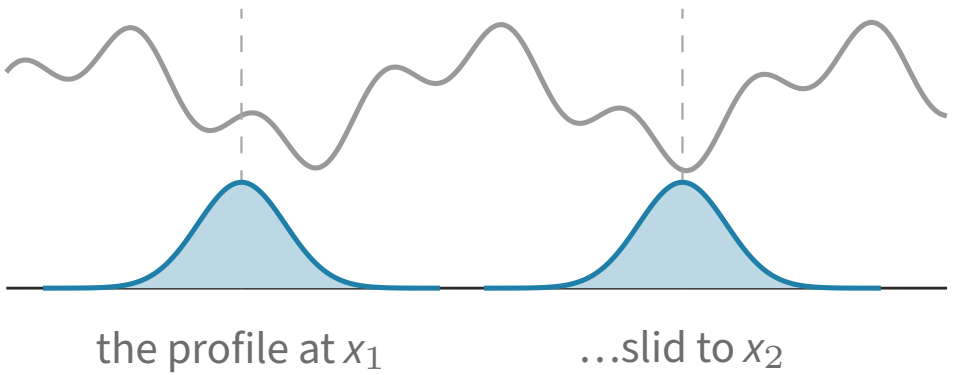
- Most direct: a small network that takes the pair (x, y) as input. It works — but on an N-point grid the sum touches all N^2 pairs: **global mixing at quadratic cost**.
- And a free $\kappa_\theta(x, y)$ knows nothing about the problem. If we know a structural property in advance, we can build it into the kernel.

The FNO [9] fixes both problems with a single restriction on κ.

Same law at every point

In the Darcy equation $-\nabla \cdot \big(a(x)\nabla u(x)\big) = f(x)$ the **law** is the same at every point — only the *data* a, f vary, and they reach the network as **input**, not as weights. So let the weights be position-blind: how y influences x may depend only on the **offset** $x - y$,

$$\kappa_\theta(x,y) = \kappa_\theta(x-y) \quad \Longrightarrow \quad (\mathcal{K}v)(x) = \int_\Omega \kappa_\theta(x-y)\, v(y)\, dy = (\kappa_\theta * v)(x).$$



The integral becomes a **convolution**: one mixing profile, slid across the whole domain.

The convolution theorem

On a periodic domain $\Omega = [0,1]^d$ the **Fourier transform** and its inverse are

$$(\mathcal{F}v)(m) = \int_\Omega v(x)\, e^{-2\pi i\, m\cdot x}\, dx, \qquad (\mathcal{F}^{-1}\widehat{v})(x) = \sum_{m\in\mathbb{Z}^d} \widehat{v}(m)\, e^{2\pi i\, m\cdot x}.$$

Convolution theorem. For $\kappa, v \in L^2(\Omega)$ and every mode $m \in \mathbb{Z}^d$, writing $\widehat{\kappa} := \mathcal{F}\kappa$,

$$\mathcal{F}(\kappa * v)(m) = \widehat{\kappa}(m)\,(\mathcal{F}v)(m), \qquad \text{i.e.} \qquad \kappa * v = \mathcal{F}^{-1}\big(\widehat{\kappa}\cdot \mathcal{F}v\big).$$

In space, the integral couples every pair x, y. In frequency the same operation is **diagonal**: mode m of the output is mode m of the input, scaled by $\widehat{\kappa}(m)$ — no mixing between modes.

Learning in Fourier space

A convolution is completely described by **one gain per frequency**, $\widehat{\kappa}(m)$. Choosing the kernel and choosing this list of gains are the same choice — so skip κ and make the list the parameters:

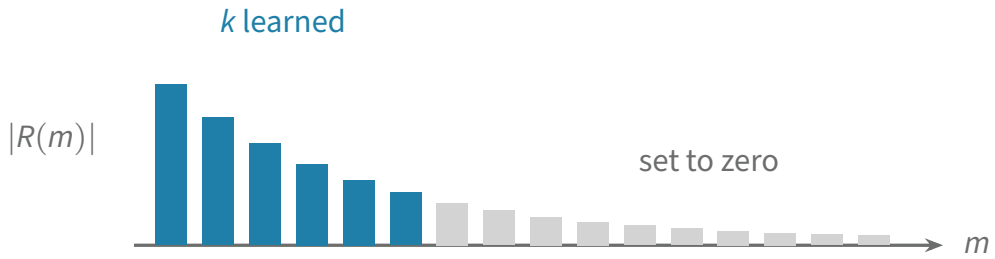
$$R(m) := \widehat{\kappa}(m), \qquad (\mathcal{K}v)(x) = \mathcal{F}^{-1}\big(R \cdot \mathcal{F}v\big)(x).$$

- **Transform, scale each mode, transform back** — we never form the kernel or evaluate the integral.
- On a grid, $\mathcal{F}$ is the FFT: $\mathcal{O}(N \log N)$ for a *global* operation — no N^2 pairs.

One catch: a function sampled at N points has N modes — so R still has one entry per grid point.

Keep only the lowest modes

Order the modes by frequency, learn the first k, set the rest to zero:



- Smooth kernels have fast-decaying coefficients — the tail did little anyway.
- k is a **hyperparameter**, N is a property of the **data**: the parameter count is ours to set again.
- A frequency belongs to the *domain*, not to the sampling. On a finer grid the first k modes are the **same modes** — the same weights apply. *This is the discretisation invariance we asked for.*

The Fourier layer, formally

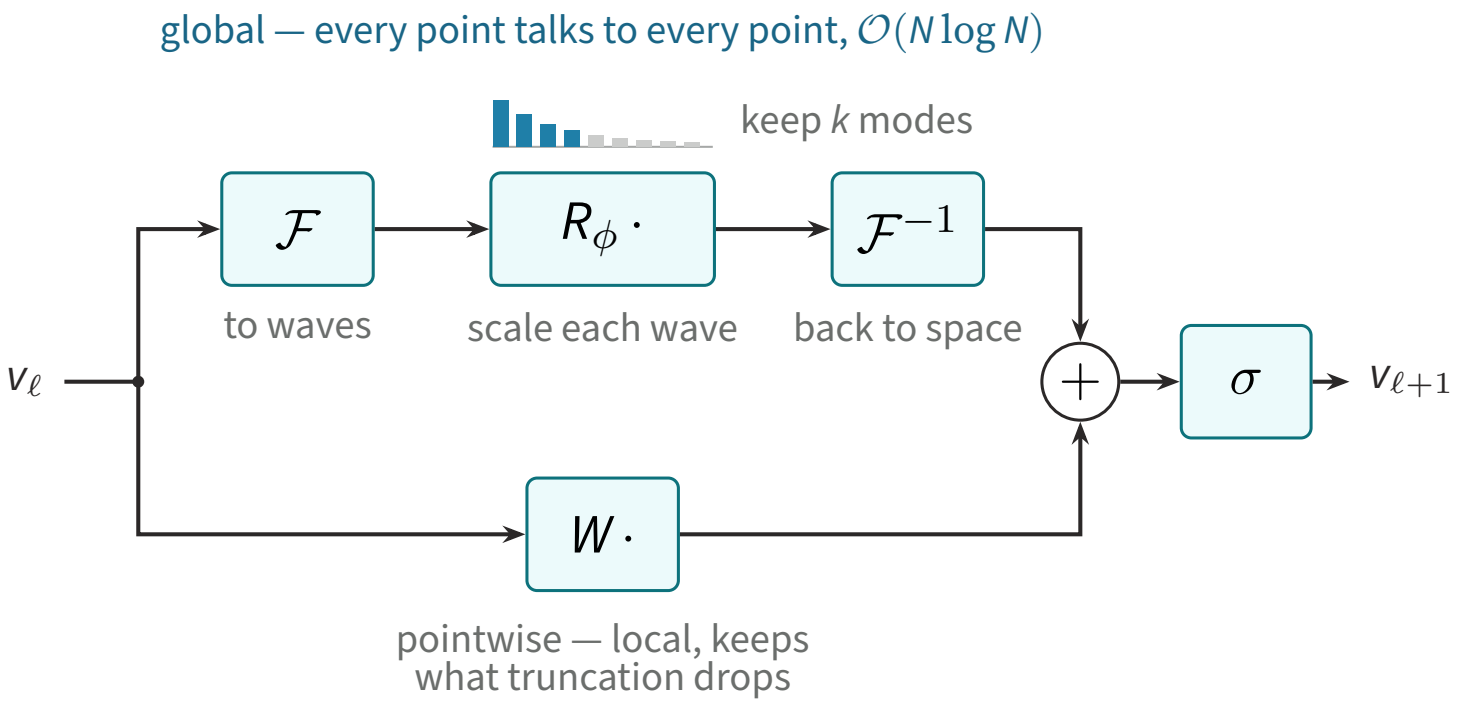
In practice $v_\ell : \Omega \to \mathbb{R}^c$ carries c channels, and each kept mode gets a full channel-mixing matrix. The **Fourier layer** [9]:

$$v_{\ell+1}(x) = \sigma\Big(W v_\ell(x) + \mathcal{F}^{-1}\big(R_\phi \cdot \mathcal{F} v_\ell\big)(x) + b\Big),$$

$$\big(R_\phi \cdot \mathcal{F} v_\ell\big)(m) = R_\phi(m)\,\big(\mathcal{F} v_\ell\big)(m), \quad R_\phi(m) \in \mathbb{C}^{c\times c} \text{ for } |m| \le k, \text{ else } 0.$$

- Parameters per layer: $k\,c^2$ complex entries in R_ϕ, plus c^2 in W — **independent of the resolution** N.
- Still the kernel layer: it realises $\kappa_\theta(x-y)$ with $\widehat{\kappa} = R_\phi$ — the kernel's Fourier coefficients *are* the weights.
- W, b, σ act pointwise, exactly as before.

The FNO block



The upper path *is* the kernel integral — computed as FFT, scale, inverse FFT; its weights are indexed by **frequency**, so the block runs unchanged at any resolution. Stack L blocks between the lift P and the projection Q: that is the whole FNO.

DeepONet (branch/trunk nets) and graph neural operators are other operator-learning architectures; they parameterise the mapping differently [8].

Neural Operators

Operator Learning Framework

Fourier Neural Operators

Applications

The input can be an initial state

For Darcy the input was a *coefficient* $a(x)$. Nothing in the framework requires that — the input can be any function the solution depends on. In particular, the **initial condition** of an evolution problem:

$$G_{\Delta t}: \; u(\cdot, t) \;\mapsto\; u(\cdot, t + \Delta t).$$

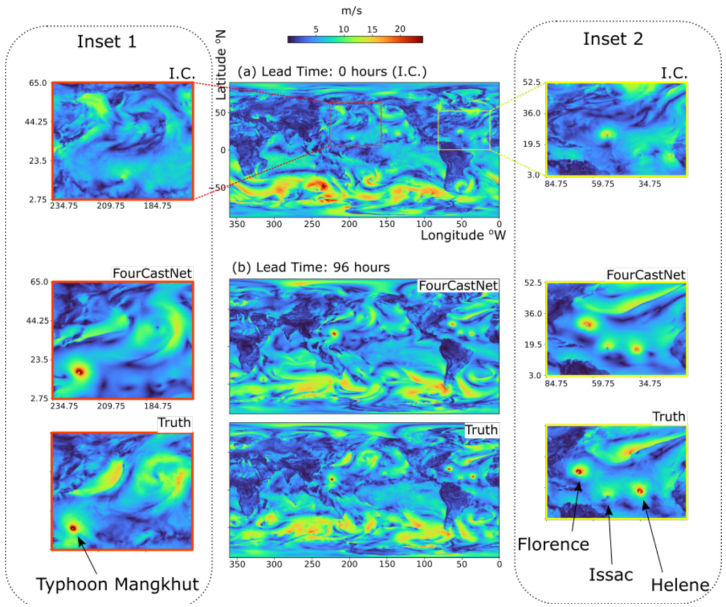
- Training data: pairs of **consecutive snapshots** from a simulation or from measurements.
- Apply $G_{\Delta t}$ repeatedly — a learned **time-stepper**: each step costs one network evaluation, regardless of the physics inside.
- The operator analogue of the Neural ODE: there we learned the pointwise dynamics f of a vector state — here the state is a whole **field**, and we learn the solution map over a step.

Every application that follows learns such a map between fields.

FourCastNet: learned weather

The state of the atmosphere is a set of fields on the globe — wind, temperature, humidity, pressure. Forecasting is an operator, G : state now $\mapsto$ state in 6 h, applied repeatedly for a week-long forecast.

- $\sim$20 fields on a 0.25° grid (721×1440), trained on $\sim$40 years of the ERA5 reanalysis [12].
- Backbone: **AFNO** [6] — the FNO block with one small shared network as the filter.
- The paper reports a week-long forecast in $\sim$2 s on one GPU, 10^4–$10^5\times$ less compute than a numerical weather model — large ensembles become affordable.



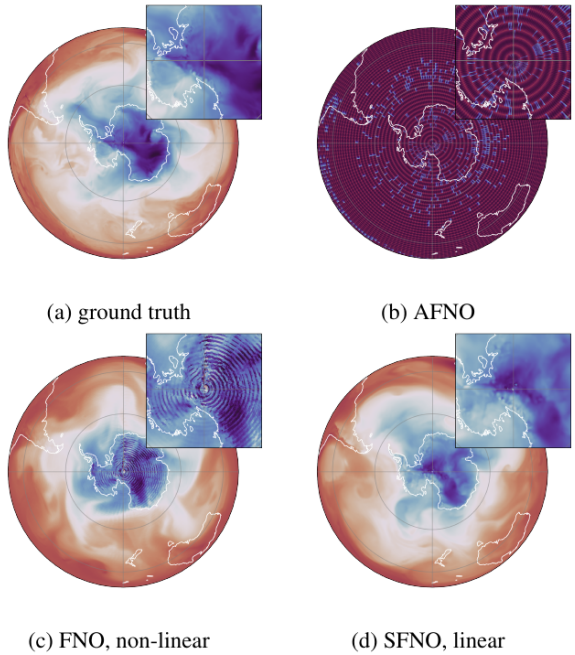
96-hour global wind forecast vs. truth: Typhoon Mangkhut and three Atlantic hurricanes. Figure from [12].

SFNO: weather lives on a sphere

The FFT assumes a periodic box. A latitude–longitude grid is periodic east–west *only*; toward the poles its cells shrink and the flat-torus picture is plain wrong. Artefacts build up there, and long rollouts of torus-FNO weather models **blow up**.

The **SFNO** [1] keeps the block but replaces $\mathcal{F}$ by the *spherical harmonic transform* — the convolution theorem holds on the sphere too.

Rollouts stay stable for a **simulated year** (1,460 steps); this powers FourCastNet's successors.

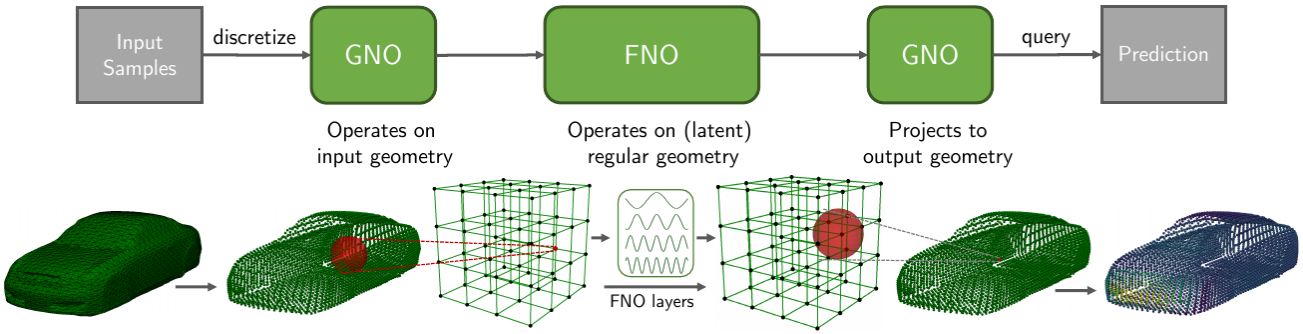


Temperature over Antarctica after 730 autoregressive steps: torus-based blocks develop polar artefacts or break down; the SFNO stays physical. Figure from [1].

Flows around shapes: Geo-FNO, GINO

Here the input is the *geometry itself* — airfoil or car shape $\mapsto$ the flow around it — but such meshes defeat the FFT.

- **Geo-FNO** [10] learns a *deformation* φ from the physical mesh to a regular grid, applies the Fourier layers there, and maps back.
- **GINO** [11]: graph layers map *point cloud* $\leftrightarrow$ regular latent grid, an FNO core between — the paper reports car surface pressure in milliseconds.
- This makes design loops cheap: each candidate shape is a forward pass instead of a meshed CFD run.

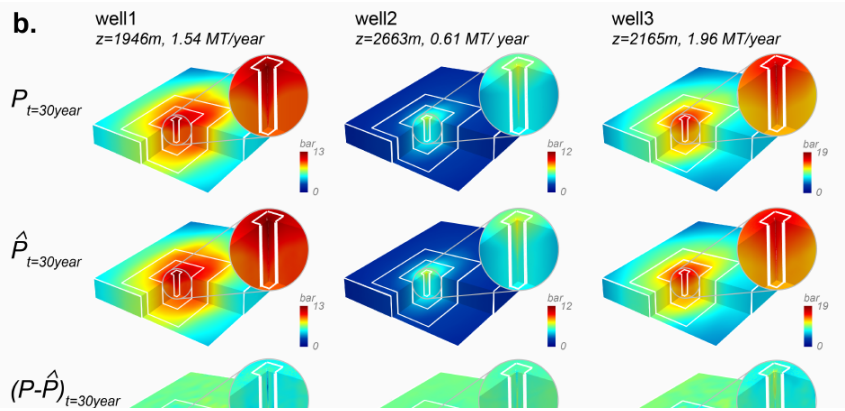


GINO: point cloud $\rightarrow$ GNO $\rightarrow$ FNO on a latent grid $\rightarrow$ GNO $\rightarrow$ surface pressure [11].

Darcy Example 2.0: CO_2 storage

Storing CO_2 underground is the multiphase, time-dependent version of our running example, *Darcy flow*: G : permeability, injection plan $\mapsto$ pressure & plume, 30 years.

- **Nested FNOs** [18]: FNOs on nested domains — coarse across the basin, fine around each well; the authors report a $\sim 700{,}000\times$ inference speedup over the reservoir simulator.
- The true permeability is never known: screen **thousands of geological realisations** for pressure build-up and leakage risk.



30-year pressure build-up at three wells: truth P, prediction $\widehat{P}$, difference [18].

F-FNO [16], U-NO [13], WNO [17] and LNO [2] keep the transform–scale–transform-back pattern, with different transforms or layer structure.

Hands-on

Four experiments: solver, spectral layer, FNO, weather

Hands-on 1: a Darcy solver

The data generator behind this whole part: sample a medium, solve the PDE (full script: `examples_py/darcy_flow.py`).

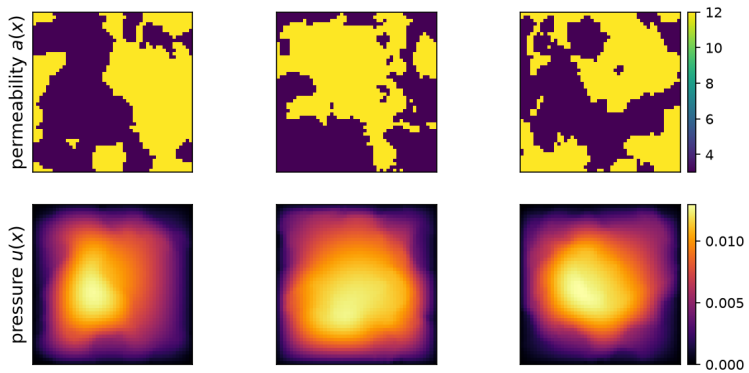
```
def grf(n, rng, alpha=2.0, tau=3.0):                # smooth random field
    xi = rng.standard_normal((n, n))
    kx, ky = np.meshgrid(*2*[np.fft.fftfreq(n, 1/n)], indexing="ij")
    c = (np.pi**2*(kx**2 + ky**2) + tau**2)**(-alpha/2); c[0, 0] = 0
    return _normed(np.fft.ifft2(np.fft.fft2(xi) * c).real)

a = np.where(grf(64, rng) >= 0, 12.0, 3.0)      # two-phase medium
u = solve_darcy(a, f_val=1.0)                   # 5-pt FD, sparse solve
```

`solve_darcy` assembles the five-point stencil (harmonic averaging at the cell faces) and solves the sparse system — ∼40 lines.

Hands-on 1: many media, one map

Every new field is one more example of the map $a \mapsto u$:

```
rng = np.random.default_rng(0)
A = np.stack([np.where(grf(64, rng) >= 0, 12., 3.)
              for _ in range(1200)])
U = np.stack([solve_darcy(a) for a in A])    # a few min on a laptop
np.savez("darcy64.npz", a=A, u=U)            # 1000 train / 200 test
```



Try: change the phase contrast (12 vs. 3), the correlation length `tau`, or the forcing `f_val` — each change gives a different family of media.

Hands-on 2: a spectral layer

```python
class Spectral2d(nn.Module):
    def __init__(self, c, k):
        super().__init__(); self.k = k
        self.R1 = nn.Parameter(0.02*torch.randn(
            c, c, k, k, dtype=torch.cfloat))   # C x C per kept mode,
        self.R2 = nn.Parameter(0.02*torch.randn(
            c, c, k, k, dtype=torch.cfloat))   # one R per corner
    def forward(self, v):
        vh  = torch.fft.rfft2(v)                                  # F
        out = torch.zeros_like(vh)
        out[..., :self.k, :self.k] = torch.einsum(                # R . Fv
            "bixy,ioxy->boxy", vh[..., :self.k, :self.k], self.R1)
        out[..., -self.k:, :self.k] = torch.einsum(               # negative r
            "bixy,ioxy->boxy", vh[..., -self.k:, :self.k], self.R2)
        return torch.fft.irfft2(out, s=v.shape[-2:])              # F^-1
```

(`rfft2` stores positive *and* negative row frequencies — a real-valued kernel needs both corners of the kept block.)

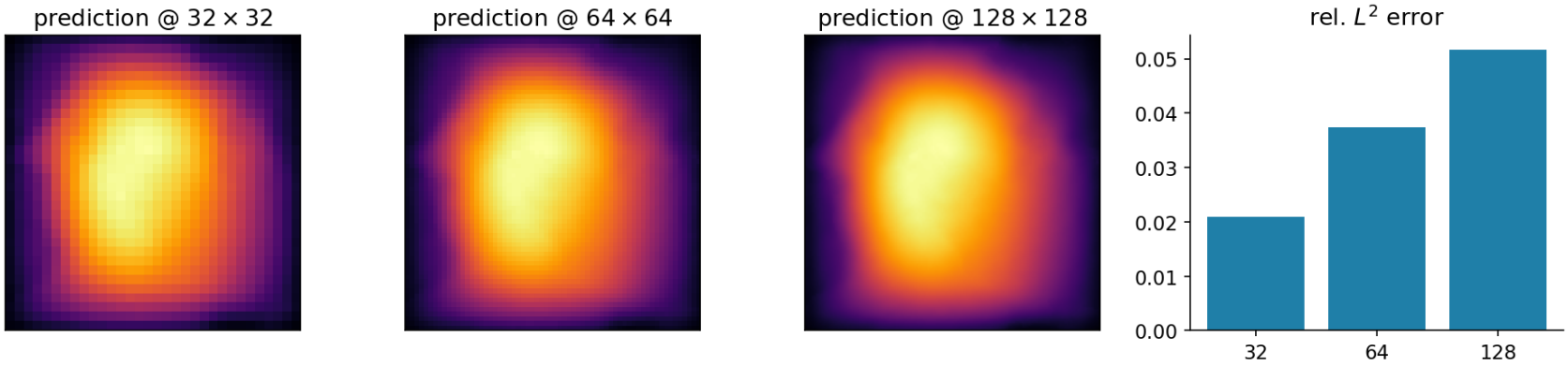
Hands-on 2: block and coordinates

```python
class Block(nn.Module):                                  # sigma(W v + K v)
    def __init__(self, c, k):
        super().__init__()
        self.S, self.W = Spectral2d(c, k), nn.Conv2d(c, c, 1)
    def forward(self, v): return F.gelu(self.S(v) + self.W(v))

class WithGrid(nn.Module):                               # (a) -> (a, x, y)
    def forward(self, v):
        b, _, h, w = v.shape
        gy, gx = torch.meshgrid(torch.linspace(0, 1, h),
            torch.linspace(0, 1, w), indexing="ij")
        g = torch.stack([gx, gy]).to(v).expand(b, -1, -1, -1)
        return torch.cat([v, g], 1)
```

The boundary pins $u = 0$, so the model must know *where* it is: append coordinate channels, as the original FNO does. Without them the stack is translation-equivariant and collapses to $\approx$0.53 test error.

Hands-on 2: train coarse, test fine

```
model = nn.Sequential(WithGrid(), nn.Conv2d(3, 32, 1),   # lift P
                      *[Block(32, k=12) for _ in range(4)],
                      nn.Conv2d(32, 1, 1))                # project Q
train(model, a32, u32)                                    # 32x32 only!
for res in (32, 64, 128):    # test data: re-run the solver there
    print(res, rel_l2(model(a_test[res]), u_test[res]))
```



Same weights, unseen grids: the error grows but stays small — R is indexed by **frequency**, and the first k modes of a finer grid are the same modes. *Try:* drop `WithGrid` and retrain.

Hands-on 3: FNO hyperparameters

The same model, production-grade, from `neuraloperator`:

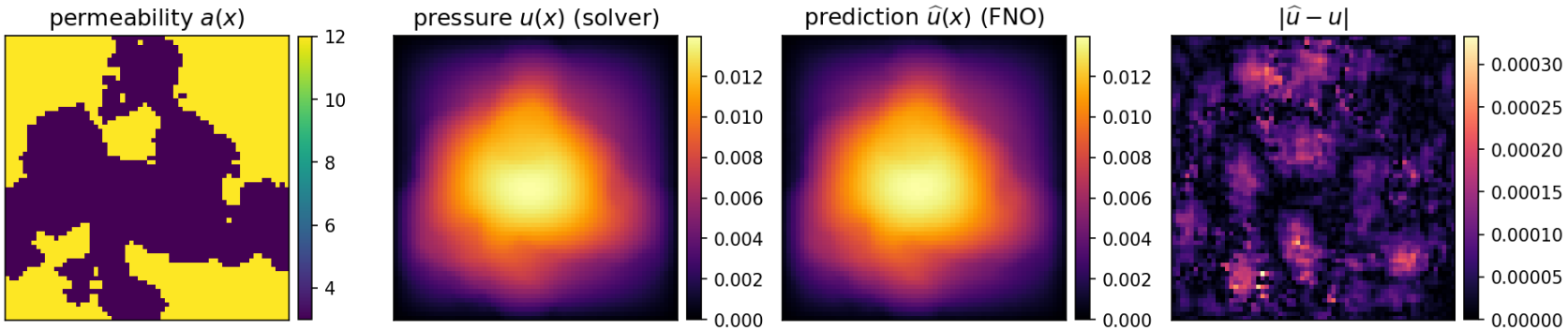
```python
from neuralop.models import FNO
from neuralop import LpLoss

model = FNO(n_modes=(16, 16), hidden_channels=64, n_layers=4,
            in_channels=1, out_channels=1)
opt   = torch.optim.AdamW(model.parameters(), lr=1e-3)
loss = LpLoss(d=2)                          # relative L2, resolution-safe
for a, u in train_loader:
    opt.zero_grad(); l = loss(model(a), u); l.backward(); opt.step()
```

- `n_modes` is the truncation: sweep $4 \to 32$ and watch error *and* parameter count (quadratic in `n_modes` and in c per layer).
- `hidden_channels` c, `n_layers` L: the usual capacity knobs.
- The library appends (x, y) coordinate channels by default (`positional_embedding="grid"`) — the boundary breaks translation invariance, and the model must know where it is.
- Too few modes $\Rightarrow$ ringing at the interfaces of a; too many $\Rightarrow$ parameters explode.

Hands-on 3: prediction vs. error

Always look at the fields, not only the loss curve:

```
u_hat = model(a[None, None]).squeeze()
for ax, img, t in zip(axs, [a, u, u_hat, (u_hat - u).abs()],
                      ["a", "u true", "u pred", "|error|"]):
    ax.imshow(img); ax.set_title(t)
```



Our Exp-2 model at $c{=}64$, $k{=}16$: test rel. L^2 0.017 (`examples_py/fno_darcy_panels.py`). The error concentrates along the **interfaces** of a — the high frequencies the truncation dropped.

Hands-on 4: a tiny slice of ERA5

Weather, at toy scale: WeatherBench [15] serves ERA5 regridded to $5.625°$ — a 32×64 global grid.

```
import xarray as xr     # two fields, two years: ~40 MB
z500 = xr.open_mfdataset("geopotential_500/*.nc")["z"]
t850 = xr.open_mfdataset("temperature_850/*.nc")["t"]

X = np.stack([z500, t850], axis=1)[::6]      # 6-hourly, 2 channels
X = (X - X.mean((0, 2, 3), keepdims=True)) \
      / X.std((0, 2, 3), keepdims=True)
x_now, x_next = X[:-1], X[1:]                # state -> state + 6 h
```

Same shape of problem as Darcy — only now the input function is the **current state** and the output is the state **6 hours later**.

Hands-on 4: mini-FourCastNet

```
model = FNO(n_modes=(8, 16), hidden_channels=48,
            in_channels=2, out_channels=2)  # very small FourCastNet
train(model, x_now, x_next)                 # one-step prediction

xk = x_test[:1]                             # autoregressive rollout
for k in range(20):                         # 20 x 6 h = 5 days
    xk = model(xk)
    rmse[k] = (xk - x_test[k+1:k+2]).pow(2).mean().sqrt()
```

- Compare `rmse[k]` against **persistence** ($\widehat{x}_{t+k} = x_t$): the model should win for several days.
- The real FourCastNet: same idea with $\sim$20 channels, 0.25°, an AFNO block and 40 years of data.
- *Try:* more channels or years; longer rollouts (where does it drift?); swap in a spherical transform (`torch-harmonics`).

Thank You!

Thank you for your time and patience!

Acknowledgements

This work was funded by Fundação para a Ciência e Tecnologia (Portuguese Foundation for Science and Technology) through CMAT projects UIDB/00013/2020 and the support of the High-Performance Computing Center at the University of Évora funded by FCT I.P. under the project “OptXAI: Constrained optimisation in NNs for Explainable, Ethical and Greener AI”, reference 2024.00191.CPCA.A1, platform Vision, and national funds through the FCT/MCTES (PIDDAC)under the project 2022.06672.PTDC—iMAD (Improving the Modelling of Anomalous Diffusion and Viscoelasticity: solutions to industrial problems), DOI 10.54499/2022.06672.PTDC (`https://doi.org/10.54499/2022.06672.PTDC`); and by the projects LA/P/0045/2020 (ALiCE), UIDB/00532/2020, and UIDP/00532/2020 (CEFT). It was also financially supported by Fundação “la Caixa”|BPI and FCT through project PL24-00057: “Inteligência Artificial na Otimização da Rega para Olivais Resilientes às Alterações Climáticas”. C. Coelho would also like to thank the KIBIDZ project funded by dtec.bw—Digitalization and Technology Research Center of the Bundeswehr; dtec.bw is funded by the European Union—NextGenerationEU. B. Zimmering would also like to thank the LaiLa project funded by dtec.bw—Digitalization and Technology Research Center of the Bundeswehr;

References I

[1] Boris Bonev et al. “Spherical Fourier Neural Operators: Learning Stable Dynamics on the Sphere”. In: *International Conference on Machine Learning (ICML)*. 2023.

[2] Qianying Cao, Somdatta Goswami, and George Em Karniadakis. “Laplace Neural Operator for Solving Differential Equations”. In: *Nature Machine Intelligence* 6.6 (June 2024), pp. 631–640. ISSN: 2522-5839. DOI: 10.1038/s42256-024-00844-4. URL: https://www.nature.com/articles/s42256-024-00844-4 (visited on 12/25/2024).

[3] Ricky T. Q. Chen. *torchdiffeq*. 2018. URL: https://github.com/rtqichen/torchdiffeq.

[4] Ricky TQ Chen et al. “Neural ordinary differential equations”. In: *Advances in neural information processing systems* 31 (2018).

[5] C. Coelho, M. Fernanda P. Costa, and L.L. Ferrás. “Neural fractional differential equations”. In: *Applied Mathematical Modelling* 144 (2025), p. 116060. DOI: 10.1016/j.apm.2025.116060.

[6] John Guibas et al. “Adaptive Fourier Neural Operators: Efficient Token Mixers for Transformers”. In: *International Conference on Learning Representations (ICLR)*. 2022.

[7] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. “Multilayer feedforward networks are universal approximators”. In: *Neural networks* 2.5 (1989), pp. 359–366.

[8] Nikola Kovachki et al. “Neural operator: Learning maps between function spaces with applications to pdes”. In: *Journal of Machine Learning Research* 24.89 (2023), pp. 1–97.

[9] Zongyi Li et al. *Fourier Neural Operator for Parametric Partial Differential Equations*. 2020. DOI: 10.48550/ARXIV.2010.08895. URL: https://arxiv.org/abs/2010.08895 (visited on 10/23/2025).

[10] Zongyi Li et al. “Fourier Neural Operator with Learned Deformations for PDEs on General Geometries”. In: *Journal of Machine Learning Research* (2023).

[11] Zongyi Li et al. “Geometry-Informed Neural Operator for Large-Scale 3D PDEs”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2023.

[12] Jaideep Pathak et al. “FourCastNet: A Global Data-Driven High-Resolution Weather Model Using Adaptive Fourier Neural Operators”. In: *arXiv preprint arXiv:2202.11214* (2022).

References II

[13] Md Ashiqur Rahman, Zachary E. Ross, and Kamyar Azizzadenesheli. "U-NO: U-shaped Neural Operators". In: *Transactions on Machine Learning Research* (2023).

[14] 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.

[15] Stephan Rasp et al. "WeatherBench: A Benchmark Data Set for Data-Driven Weather Forecasting". In: *Journal of Advances in Modeling Earth Systems* 12.11 (2020).

[16] Alasdair Tran et al. "Factorized Fourier Neural Operators". In: *International Conference on Learning Representations (ICLR)*. 2023.

[17] Tapas Tripura and Souvik Chakraborty. "Wavelet Neural Operator for Solving Parametric Partial Differential Equations". In: *Computer Methods in Applied Mechanics and Engineering* (2023).

[18] Gege Wen et al. "Real-Time High-Resolution CO2 Geological Storage Prediction Using Nested Fourier Neural Operators". In: *Energy & Environmental Science* 16.4 (2023), pp. 1732–1741.