UFR de mathématique et d'informatique
Université de Strasbourg

# Symbolic Regression

**Giulio Carpi Lapi** and **Abdou Wade**

supervised by

**Emmanuel Franck** and **Victor Michel-Dansac**

Date: **January 30, 2025**

# Contents

# 1 Introduction

In many practical applications across engineering and the physical sciences, one frequently encouters **partial differential equations** (PDEs) describing the evolution of a physical quantity in space and time. Among these, the **advection-diffusion** (or convection-diffusion) equation is ubiquitous for modeling the transport of a scalar field, such as the concentration of a pollutant in air or water, or heat in a conductive medium. Althouigh analytical solutions exist for special cases—like Gaussian initial conditions in one spatial dimension—realistic settings often require numerical simulations with potentially large grids. Such high-dimensional simulations can become computationally expensive, especially when one needs to explore multiple parameter variations (e.g., different initial conditions or physical coefficients).

To alleviate the computational burden, **reduced-order modelling** has emerged as a potent technique. Reduced-order models (ROMs) aim to capture the essential dynamics of a PDE in a significantly lower-dimensional space. Traditional ROM techniques often rely on linear transformations such as Proper Orthoganl Decomposition (POD). However, neural networks have introduced new ways to **nonlinearly** reduced dimensionality. In particular, **autoencoders**—a class of neural networks designed for unsupervised representation learning—have shown promise in compressing high-fidelity PDE solutions into compact latent spaces [1].

A persistent challenge remains: while neural networks can approximate complex, high-dimensional dynamics with impressive accuracy, they often lack interpretability. The network's "decoder" from the latent space back to physical space is usally a black-box feedforward function. This hinders the ability to gain analytical insights or to incorporate domain knowledge explicitly.

**Symbolic regression**—and, in particular, the family related to **Sparse Identification of Nonlinear Dynamics** (SINDy)—offers a solution: discover a symbolic or closed-form expression that maps the learned latent variables back into physical variables. By doing so, one aims to replace (or augment) the network's black-box decoding with a simplified and interpretable formula that captures the underlying PDE structure.

## 1.1 Project overview

In this project, we consider a **1D advection-diffusion** PDE with a Gaussian initial condition. We solve it numerically across different initial means ($\mu_0$) and standard deviations ($\sigma_0$). We then train:

1. A **convolutional autoencoder** to compress the PDE snapshots into a low-dimensional latent representation (in our case, $\mathbb{R}^2$).
2. A **symbolic decoder** (based on SINDy-like ideas) that attempts to learn an explicit formula linking latent variables back to the PDE state $u(x)$.

This report presents the mathematical background of the PDE, the methodology for data Generation and neural network training, the symbolic regression approach, anbd key results. The hope is taht such "hybrid" approach-combining the efficiency and expressivity of deep learning with the interpretability of symbolic regression-provides a robust framework for data-driven PDE modeling.

## 1.2 Project Roadmap

We created a roadmap on to ensure we meet the deadlines for the deliverables. The deadlines and descriptions for each milestone are as follows:

1. **V0 - 21th November 2024**
    - Presentation for V0: project overview and initial plans.
    - Initial Report: Detailed mathematical formulations and initial numerical methods.
    - Codebase: Initial numerical solver implemented and tested.
2. **V1 - 10th January 2025**
    - Trained Neural Network Models

- Updated Project Report
- Presentation for V1

3. **V2 - 27th February 2025**
   - Final Reduced-Order Model with Analytical Decompression
   - Comprehensive Project Report
   - Final Presentation for V2

Detailed roadmaps for each milestone are provided below.

### 1.2.1 V0 - 21st November 2024

- Define **Project Context** and **Objectives**.
- **Initial Project Setup**.
- Plan **Project Roadmap**.
- **Theoretical Analysis** & **Mathematical Formulation**.
- Initial **Numerical Implementation**.
- Initial **Data Description**.
- V0 **Report** & **Presentation**.
- LaTeX workflow.
- Gather **Feedback**.

### 1.2.2 V1 - 10th January 2025

- **Methodology**, **Objectives**, **Roadmap** update.
- Comprehensive **Data Generation**.
- Familiarizing with **NN** for **ROM**.

- **Data Preprocessing**.
- NN **Desing** and **Training**.
- First **Results Analysis**.
- Initial **Validation** & **Analysis**.
- V1 **Report**.
- Gather **Feedback**.

#### 1.2.3 V2 - 27th February 2025

- **Symbolic Regression** for **Analytical Decompression**.
- **Integration** into **Reduced Model**.
- Comprehensive **Validation** and **Analysis**.
- **Project Retrospective**.
- Set up **CI/CD** and **Docker Image**.
- V0 **Report** & **Presentation**.
- Prepare for **Defense**.
- Create **GitHub Discussion**.
- Gather **Feedback**.

## 2 Background

### 2.1 Advection-Diffusion and Its Importance

The **advection-diffusion equation** is a cornerstone of transport phenomena in fluid mechanics, heat transfer, and other disciplines [2].

For a scalar field $u(x,t)$-representing, for instance, the concentration of a chemical species-the $1D$ advection-diffusion PDE can be written in non-conservative form as:

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} - D\frac{\partial^2 u}{\partial x^2} = 0, \tag{1}$$

where $a$ is the advection speed and $D$ is the diffusion coefficient. Despite appearing simple, this PDE can be challenginf to solve numerically in high-dimensional setting because stability constraints typically require small time stemps and relatively fine grids.

Moreover, in inverse **inverse modeling** or **parameter estimation** tasks, one often wants repeated PDE solves for various values of $(\mu_0, \sigma_0)$ or other physical parameters. This repeated solving can be computationally prohibitive, thus motivating low-dimensional surrogate or reduced-order models.

## 2.2 Neural Network Surrogates and Autoencoders

A promising avenue for constructing surrogates lies in **deep learning**. Instead of explicitly deriving reduced bases through linear techniques like POD, we can use an **autoencoder**-a neural network architecture designed for **unsupervised** feature learning and dimensionality reduction.

**What is an Autoencoder?** An autoencoder typically consists of two main parts:

1. **Encoder:** Takes the high-dimensional input (e.g, a PDE snapshot $u(x)$) and maps it into a low-dimensional latent vector $z$. Convolutional layers are especially useful for spatially structured data like images or $1D$ fields.

2. **Decoder:** Takes the latent vector $z$ and attempts to reconstruct the original high-dimensional data.

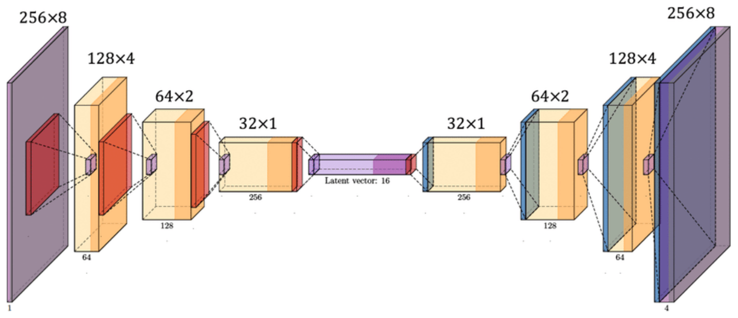


Figure 1: Autoencoder convolutional neural network (CNN) structure. [3]

The autoencoder is trained by minimizing a **reconstruction loss** (often the mean squared error) between the network's output and the original data. Through training, it learns to encode the essential features of the snapshots in a compressed form (the latent space) [1]. One can then view $z$ as the reduced coordinates that capture the dominant variability of the PDE solutions.

We decided to use a convolutional architecture because convolutional layers exploit local spatial correlations, significantly reducing the number of parameters and making the network more efficient for grid-based PDE data. Our project uses **Conv1d** and **ConvTranspose1d** layers to handle the $1D$ solution array $u(x)$.

The main difference between a **Conv** layer (standard convolution) and a **ConvTranspose** layer (often called "deconvolution" or "transposed convolution") is how they handle the spatial dimension (the output size relative to the input):

- **Conv** (standard convolution):
  - Performs a local filtering/aggregation operation on the input data.
  - Tends to **reduce** the spatial resolution (or maintain it, depending on the configuration) through the use of strides and padding.
  - Commonly used in the "encoding" phases to extract features.
- **ConvTranspose** (transposed convolution or "deconvolution"):
  - Allows for **increasing** (or reconstructing) the spatial resolution.

- Conceptually reverses the convolution operation (even though mathematically it is not always a simple inversion).
- Used in the "decoding" or reconstruction phases, for example in an autoencoder to return to the original spatial dimension after compression.

In summary, **Conv1d/2d/3d** layers are frequently found in the **encoder** part of the network (where information is to be compressed) and **ConvTranspose1d/2d/3d** layers in the **decoder** part (where the original resolution is to be restored).

## 2.3 Symbolic Regression and SINDy

**Symbolic regression** aims to find analytical expressions that best fit a given dataset, typically with constraints or objectives that prioritize simplicity or interpretability. A well-known approach in this domain is **Sparse Indentification of Nonlinear Dynamics (SINDy)**, which was introduced to discover equations of dynamical systems directly from time-series data [4].

The essential steps of SINDy-like methods include:

1. **Library Construction:** Create a library of candidate functions $\theta_1(z), \theta_2(z), \ldots$ of the latent variables $z$. These candidate functions might include polynomials, trigonometric functions, exponentials, and products.
2. **Sparse Regression:** Use a regression technique with $L_1$-type regularization to enforce many coefficients to zero, thereby discovering a sparse combination of candidate functions that reconstruct the data.
3. **Thresholding:** After the regression step, small coefficients are pruned to zero to enforce further sparsity, effectively yielding a human-readable symbolic expression.

In this project, the "symbolic decoder" attemps to learn a directr mapping $z \to u(x)$. Of course, $u$ still depends on spatial coordinate $x$. However, if we gather multiple snapshots of $PDE$ solutions for different parameters, each snapshot is associated with a $2D$ latent vector $(z_1, z_2)$. The symbolic decoder tries to find an expression of the form:

$$(\hat{u})(x) = \sum_{k=1}^{m} \alpha_k \theta_k(z_1, z_2),$$

where $\alpha_k$ are learned coefficients and $\theta_k$ are candidates basis functions. BEcause we aim for a function of $z_1$ and $z_2$ only, the approach affectively "collapses" the PDE solution into the latent space representation discovered by the autoencoder, then reconstructs it in an interpretable symbolic form.

## 3 Mathematical Formulation

### 3.1 Advection-Diffusion Equation

#### 3.1.1 Conservative Form

$$\frac{\partial u}{\partial t} = \nabla \cdot (D\nabla u - au), \tag{2}$$

Equation (2) represents the conservative form of the advection-diffusion equation.

#### 3.1.2 Non-Conservative Form

Assuming constant coefficients and a one-dimensional spatial domain, we can expand the divergence operator to obtain the non-conservative form:

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} - D\frac{\partial^2 u}{\partial x^2} = 0, \tag{3}$$

where

- $\frac{\partial u}{\partial t}$ is the time derivative of $u$,
- $a\frac{\partial u}{\partial x}$ represents the advection term,
- $D\frac{\partial^2 u}{\partial x^2}$ represents the diffusion term.

### 3.2 Initial Condition

We use a Gaussian initial condition, which is widely used in problems involving diffusion and advection-diffusion equations due to its mathematical simplicity and its ability to model smooth, localized distributions [5]:

$$u(x,0) = \frac{1}{\sigma_0\sqrt{2\pi}} \exp\left(-\frac{(x-\mu_0)^2}{2\sigma_0^2}\right), \tag{4}$$

where

- $\mu_0$ is the initial mean,
- $\sigma_0^2$ is the initial variance.

## 4 Methodology

### 4.1 Analytical Solution

We assume that the solution remains Gaussian over time, which is supported by the mathematical properties of the advection-diffusion equation under certain conditions [5]:

$$u(x,t) = \frac{1}{\sigma(t)\sqrt{2\pi}} \exp\left(-\frac{(x-\mu(t))^2}{2\sigma^2(t)}\right). \tag{5}$$

### 4.1.1 Derivation of Time Evolution

We compute the derivatives and substitute into Equation (3):

**Time derivative:**

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial \mu}\frac{d\mu}{dt} + \frac{\partial u}{\partial \sigma^2}\frac{d\sigma^2}{dt}$$

Where:

$$\frac{\partial u}{\partial \mu} = u\left(-\frac{(x-\mu)}{\sigma^2}\right)$$

$$\frac{\partial u}{\partial \sigma^2} = u\left(\frac{(x-\mu)^2 - \sigma^2}{2\sigma^4}\right)$$

Thus we have:

$$\frac{\partial u}{\partial t} = u\left[-\frac{(x-\mu)}{\sigma^2}\frac{d\mu}{dt} + \frac{(x-\mu)^2 - \sigma^2}{2\sigma^4}\frac{d\sigma^2}{dt}\right]$$

**Spatial derivatives:**

First derivative:

$$\frac{\partial u}{\partial x} = u\left(-\frac{(x-\mu)}{\sigma^2}\right)$$

Second derivative:

$$\frac{\partial^2 u}{\partial x^2} = u\left(\frac{(x-\mu)^2 - \sigma^2}{\sigma^4}\right)$$

**Substitute into the Non-Conservative Form:**

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} - D\frac{\partial^2 u}{\partial x^2} = 0,$$

$$u\left[-\frac{(x-\mu)}{\sigma^2}\frac{d\mu}{dt} + \frac{(x-\mu)^2-\sigma^2}{2\sigma^4}\frac{d\sigma^2}{dt}\right] + a\cdot u\left(-\frac{(x-\mu)}{\sigma^2}\right) - D\cdot u\left(\frac{(x-\mu)^2-\sigma^2}{\sigma^4}\right) = 0$$

$$u\left[-\frac{(x-\mu)}{\sigma^2}\frac{d\mu}{dt} + \frac{(x-\mu)^2-\sigma^2}{2\sigma^4}\frac{d\sigma^2}{dt}\right] + a\cdot u\left(-\frac{(x-\mu)}{\sigma^2}\right) = D\cdot u\left(\frac{(x-\mu)^2-\sigma^2}{\sigma^4}\right)$$

Simplify:

$$\left[-\frac{(x-\mu)}{\sigma^2}\frac{d\mu}{dt}\right] + \left[\frac{(x-\mu)^2-\sigma^2}{2\sigma^4}\frac{d\sigma^2}{dt}\right] + a\left(-\frac{(x-\mu)}{\sigma^2}\right) = D\left(\frac{(x-\mu)^2-\sigma^2}{\sigma^4}\right)$$

### 4.1.2 Resulting ODEs

After substituing and equating coefficients, we obtain the following two ordinary differential equations (ODEs):

$$\frac{d\mu}{dt} = a, \tag{6}$$

$$\frac{d\sigma^2}{dt} = 2D. \tag{7}$$

**Interpretation:**

The complex PDE (3) can be reduced to two simpler ODEs under the assumption of Gaussian preservation.

- The mean $\mu(t)$ evolves linearly with time, with a rate of change equal to the advection velocity $a$,
- The variance $\sigma^2(t)$ evolves linearly with time, with a rate of change equal to twice the diffusion coefficient $D$.

## 4.2 Numerical Methods

### 4.2.1 Spatial Discretization

We discretize the spatial domain into $N+1$ grid points with uniform spacing $\Delta x$, such that $x_i = x_0 + i\Delta x$ for $i = 0, 1, \ldots, N$.

**Advection Term** For the advection term $a\frac{\partial u}{\partial x}$, we will use the **upwind scheme**. The upwind scheme is appropriate for positive advection-dominated problems as it accounts for the direction of the flow and provides numerical stability.

Since the advection velocity $a$ is positive, we use the backward difference approximation:

$$\left(\frac{\partial u}{\partial x}\right)_i \approx \frac{u_i - u_{i-1}}{\Delta x}, \quad \text{for} \quad a > 0.$$

If $a$ were negative, we would use the forward difference:

$$\left(\frac{\partial u}{\partial x}\right)_i \approx \frac{u_{i+1} - u_i}{\Delta x}, \quad \text{for} \quad a < 0.$$

**Diffusion Term** For the diffusion term $D\frac{\partial^2 u}{\partial x^2}$, we will use the **three-point centered scheme** for the second derivative. This scheme is second-order accurate and symmetric, providing a good balance between accuracy and computational efficiency.

The three-point centered scheme approximates the second derivative as follows:

$$\left(\frac{\partial^2 u}{\partial x^2}\right)_i \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2}.$$

### 4.2.2 Temporal Discretization

For the temporal derivative $\frac{\partial u}{\partial t}$, we will use the **explicit Euler method**, a first-order-time stepping scheme. This method is straightfoward to implement and suitable for problems where high temporal accuracy is not critical.

The explicit Euler approximation is:

$$\left(\frac{du}{dt}\right)_i^n \approx \frac{u^{n+1} - u^n}{\Delta t},$$

where:

- $u_i^n$ is the numerical approximation of $u$ at grid point $i$ and time level $n$,
- $\Delta t$ is the time step size,
- $n$ is the time level, and $n + 1$ is the next time level.

## 4.3 Discrete Equation

Combining the spatial and temporal discretizations, we derive the fully discrete form of the advection-diffusion equation.

Starting from the non-conservational form:

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} - D\frac{\partial^2 u}{\partial x^2} = 0,$$

we approximate each term as follows:

**Time Derivative:**

$$\left(\frac{\partial u}{\partial t}\right)_i^n \approx \frac{u_i^{n+1} - u_i^n}{\Delta t}.$$

**Advection Term:**

$$\left(a\frac{\partial u}{\partial x}\right)_i^n \approx a \cdot \frac{u_i - u_{i-1}}{\Delta x}.$$

**Diffusion Term:**

$$\left(D\frac{\partial^2 u}{\partial x^2}\right)_i^n \approx D \cdot \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2}.$$

Substituing these approximations into the PDE, we obtain:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + a \cdot \frac{u_i - u_{i-1}}{\Delta x} - D \cdot \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2} = 0.$$

Rearranging the equation to solve for $u_i^{n+1}$, we get:

$$u_i^{n+1} = u_i^n - \Delta t \left[a \cdot \frac{u_i^n - u_{i-1}^n}{\Delta x} - D \cdot \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}\right].$$

### 4.3.1 Implementation Details

**Boundary Conditions**

The boundary conditions enforce that the solution $u$ is zero at the ends of the spatial domain, that is, at $x = 0$ and $x = L$. These Dirichlet conditions ensure that the concentration remains zero at the boundaries throughout the simulation. They are implemented in the **advection_diffusion_step** function in the **solver.py** file as follows:

```
# Dirichlet BCs
u_new[0] = 0.0
u_new[-1] = 0.0
```

This approach ensures that the values of $u$ at the boundary points are maintained at zero at each time step.

**Stability Considerations** The explicit Euler method combined with explicit spatial discretization schemes imposes stability constraints on the time step $\Delta t$.

**Diffusion Stability Condition:**

The diffusion term requires:

$$\Delta t \leq \frac{\Delta x^2}{2D}.$$

**Advection Stability Condition (CFL Condition):**

The Courant-Friedrichs-Lewy (CFL) condition for the advection term is:

$$\Delta t \leq \frac{\Delta x}{a}.$$

To ensure numerical stability for both terms, we choose the smallest time step that satisfies both conditions:

$$\Delta t \leq \min\left(\frac{\Delta x^2}{2D}, \frac{\Delta x}{a}\right).$$

### 4.3.2 Theoretical and Numerical PDE Setup:

The initial step involves constructing a finite difference scheme to solve the PDE, as the reduced model will weed numerical solutions for training.

The **advection term** will use an **upwind-centered scheme** (appropriate given a positive velocity $a$).

The **diffusion term** will use a **three-point** centered scheme, with an explicit **Euler** method for time integration.

## 4.4 Generating Training Data

We vary initial condition parameters $\mu_0$ and $\sigma_0$. For each par $(\mu_0, \sigma_0)$:

1. We initialize $u(x, 0)$ as Gaussian with mean $\mu_0$ and s standard deviation $\sigma_0$.
2. We step through time until $T$, saving snapshots at specific intervals.

The result is a dictionary-like dataset. This provides a variety of PDE solutions to train both the autoencoder and the symbolic decoder.

## 4.5 Convolutional Autoencoder

We organize our PDE solutions into a **PyTorch** *TensorDataset*. Each sample is a $1D$ array of lenght $Nx$, reshaped as *1, NX* for a Conv1d-based autoencoder. The autoencoder architecture typically looks like:

- **Encoder:**
  - A sequence of 1D convolutions with strides that reduce the spatial dimension step by step.

  - Eventually flattens to a latent dimension of size 2.

- **Decoder:**
  - A sequence of transposed (fractionally strided) 1D convolutions that invert the process of the encoder, reconstructing a *(1, Nx)* field.

We train this model using mean squared error (MSE) loss between the reconstructed and original PDE snapshots, using an optimizer like Adam. After training, the encoder effectively learns to compress PDE snapshots into a 2D latent representation.

## 4.6 Symbolic Decoder (SINDy-Like)

Once we have a trained autoencoder, we do the following:

1. **Encode** all the PDe snapshots in our dataset to obtain latent vectors $(z_1, z_2)$.
2. **Define a symbolic decoder** as a linear combination of candidate functions in $(z_1, z_2)$:
$$\hat{u}(x_i) = \Theta \cdot \alpha,$$
where $\Theta(z)$ is a vector of basis functions (e.g, polynomials, sines, cosines, exponentials) and $\alpha$ are trainable coefficients.
3. **Add an $\mathbf{L}_1$ penalty** to the loss so that many coefficients get driven to zero, encouraging a sparse solution.
4. **Train** this symbolic decoder to minimize the MSE between $\hat{u}(x)$ and the original PDE snapshots, plus the regularization term.
5. **Apply thresholding** to prune coefficients below some small value.

The final result is a set of expressions that map from the latent space $(z_1, z_2)$ to each spatial coordinate $u(x_i)$. Although we still treat each $x_i$ as an independent output, the discovered expressions may reveail interpretable relationships tied to $(z_1, z_2)$.

# 5 Implementation Details

## 5.1 GitHub Repository

The source code, numerical implementations, and other project resources are available on GitHub:

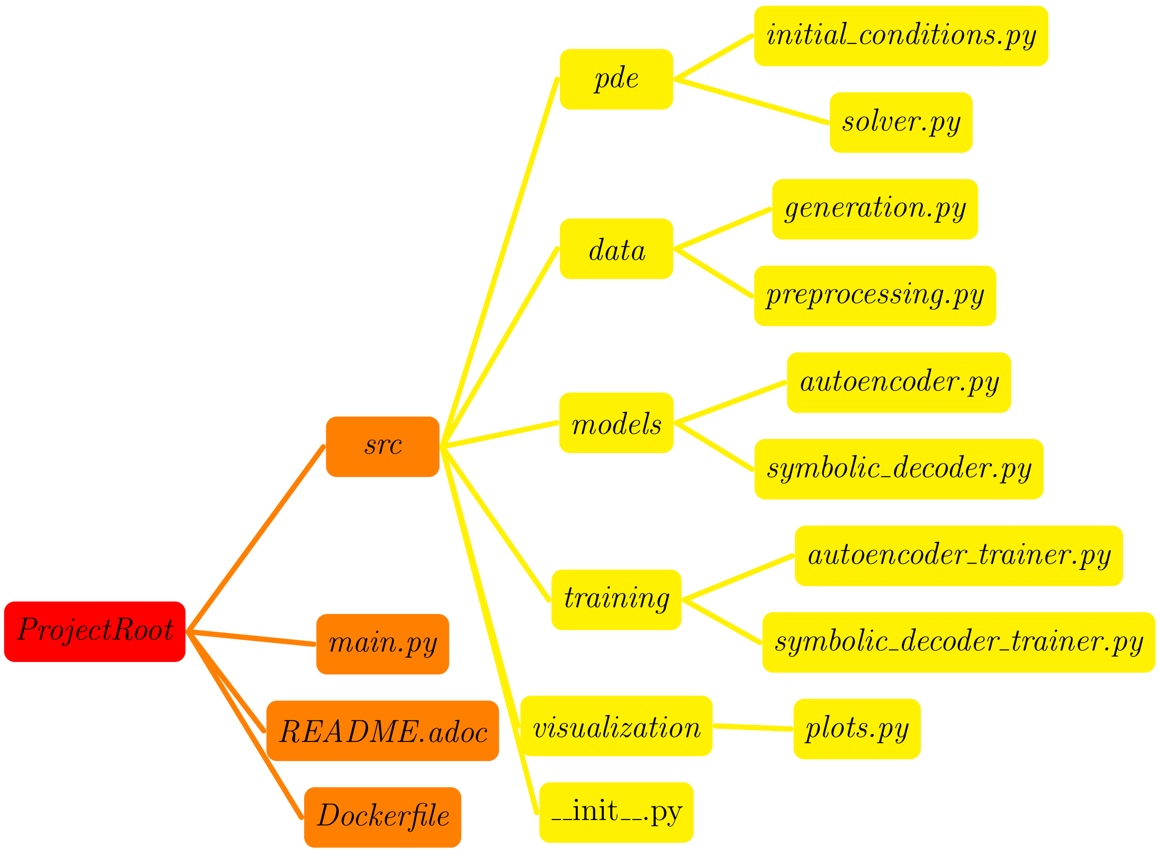
github.com/master-csmi/2024-m2-project-symbolic-regression

The repository contains Python scripts for numerical simulations, neural network training, and symbolic methods, along with examples and documentation.

## 5.2 Software and Libraries

- **Python 3.10+**
- **NumPy** for numerical operations
- **Matplotlib** for visualization
- **PyTorch** for deep learning modules
- **PyTest** for unit testing

## 5.3 Project Structure



## 5.4 Computational Domain and Parameters

A typical setup might be:

- Spatial domain: $[0, 1]$ discretized into $Nx = 96$ points
- Time domain: $t \in [0, 0.3]$ with $\Delta t$ chosen to meet the CFL constraints.
- Advection speed: $a = 1.0$
- Diffusion coefficient: $D = 0.1$

For a variety of $u_0 \in 0.3, 0.5, 0.7$ and $\sigma_0 \in 0.05, 0.1, 0.15$, we generate PDE solutions. The final dataset might contain several thousand snapshots, enough to train and validate the models.

## 5.5 Implementation

Below are the main functions:

```python
def generate_advection_diffusion_data(mu0_list, sigma0_list,
   a, D, x, dx, dt, Nt, time_stride):

    solutions = {}

    for mu0 in mu0_list:
        for sigma0 in sigma0_list:
            # Initial condition
            u = gaussian_initial_condition(x, mu0, sigma0)

            snapshots = []

            # Store initial snapshot at t=0
            snapshots.append(u.copy())

            # Time stepping
            for n in range(1, Nt):
                u = advection_diffusion_step(u, a, D, dx, dt)

                if n % time_stride == 0 or n == Nt - 1:
                    snapshots.append(u.copy())

            u_stored = np.array(snapshots)

            key = f"mu0_{mu0}_sigma0_{sigma0}"
            solutions[key] = u_stored

    return solutions
```

Listing 1: Function to solve the advection-diffusion PDE for multiple $(\mu_0, \sigma_0)$ `generation.py`

```
class ConvAutoencoder(nn.Module):
    def __init__(self, input_dim=100, latent_dim=2):
        super().__init__()
        # Encoder
        self.encoder = nn.Sequential(
            nn.Conv1d(1, 32, kernel_size=5, stride=2, padding=2),
            nn.ELU(),
            nn.Conv1d(32, 64, kernel_size=5, stride=2, padding=2),
            nn.ELU(),
            nn.Conv1d(64, 128, kernel_size=5, stride=2, padding=2),
            nn.ELU(),
            nn.Flatten(),
            nn.Linear(128*(input_dim//8), 128),
            nn.ELU(),
            nn.Linear(128, latent_dim),
        )
        # Decoder
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 128),
            nn.ELU(),
            nn.Linear(128, 128*(input_dim//8)),
            nn.ELU(),
            nn.Unflatten(1, (128, input_dim//8)),
            nn.ConvTranspose1d(128, 64, kernel_size=5, stride=2,
                                padding=2, output_padding=1),
            nn.ELU(),
            nn.ConvTranspose1d(64, 32, kernel_size=5, stride=2,
                                padding=2, output_padding=1),
            nn.ELU(),
            nn.ConvTranspose1d(32, 1, kernel_size=5, stride=2,
                                padding=2, output_padding=1),
        )

    def forward(self, x):
        z = self.encoder(x)
        x_recon = self.decoder(z)
        return x_recon
```
Listing 2: Convolutionsl Autoencoder class deifnition `autoencoder.py`

```python
class SymbolicDecoder(nn.Module):
    def __init__(self, latent_dim=2, output_dim=96, library_size=16):
        super(SymbolicDecoder, self).__init__()

        self.latent_dim = latent_dim
        self.output_dim = output_dim
        self.library_size = library_size

        self.coefficients = nn.Parameter(torch.randn(output_dim, library_size) * 0.01)

        self.nonlinear_params = nn.Parameter(torch.randn(library_size) * 0.1)

        self.library_functions = [
            lambda z: torch.ones_like(z[:, 0]),
            lambda z: z[:, 0],
            lambda z: z[:, 1],
            lambda z: z[:, 0] ** 2,
            lambda z: z[:, 1] ** 2,
            lambda z: torch.sin(self.nonlinear_params[0] * z[:, 0]),
            lambda z: torch.cos(self.nonlinear_params[1] * z[:, 1]),
            lambda z: z[:, 0] * z[:, 1],
            lambda z: z[:, 0] ** 3,
            lambda z: z[:, 1] ** 3,
            lambda z: torch.exp(self.nonlinear_params[2] * z[:, 0]),
            lambda z: torch.exp(self.nonlinear_params[3] * z[:, 1]),
        ]

        while len(self.library_functions) < self.library_size:
            self.library_functions.append(lambda z: torch.zeros(z.size(0), device=z.device))

def forward(self, z):

    library_matrix = []
    for func in self.library_functions[:self.library_size]:
        library_matrix.append(func(z))

    Theta = torch.stack(library_matrix, dim=1)

    x_recon = Theta @ self.coefficients.t()
    return x_recon
```

Listing 3: Symbolic Decoder class deifnition `autoencoder.py`

## 6 Results

### 6.1 Autoencoder Performance

We trained the convolutional autoencoder for around 100 epochs using the Adam optimizer with a learning rate of $1e-3$. The final reconstruction loss on the traininf set typically droped below $1e-3$.

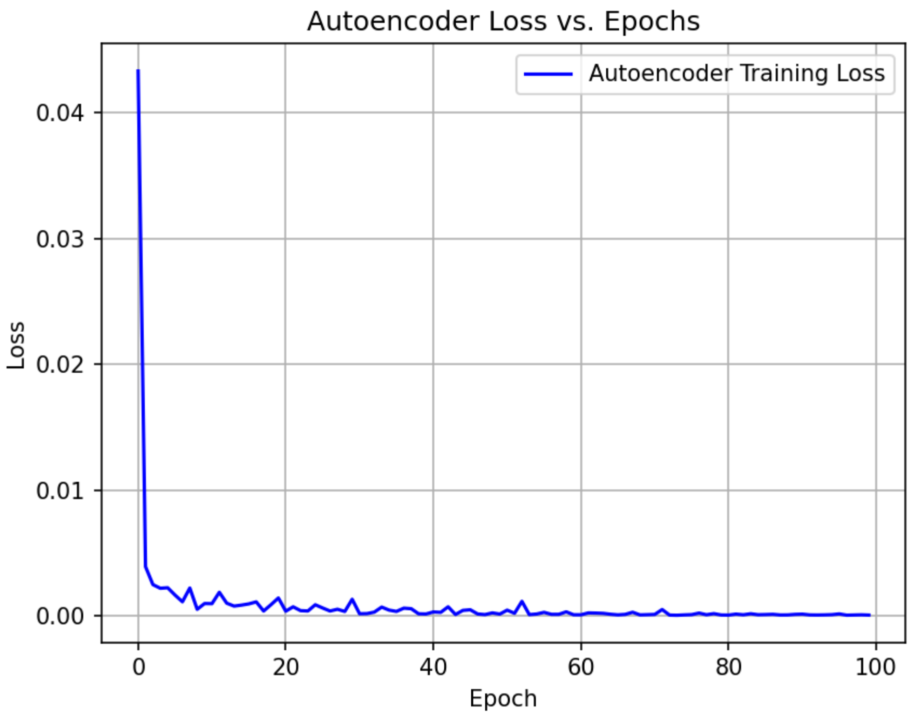


Figure 2: Training loss evolution of the autoencoder over epochs. [6]

When visually comparing original vs. reconstructed PDE snapshots at different times, the reconstructions are almost indistinguishable from the ground truth.

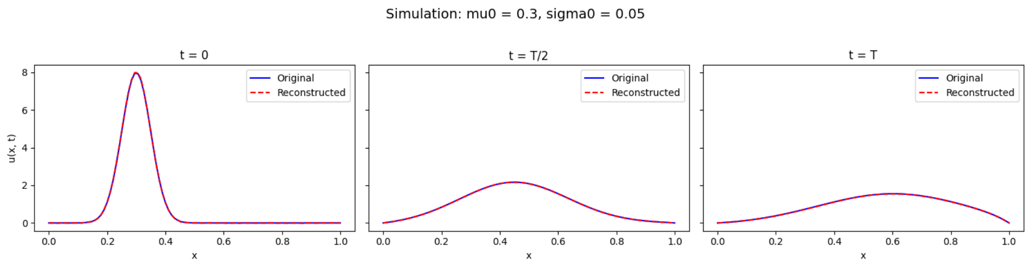


Figure 3: Autoencoder's reconstruction of PDE snapshots at different times. [7]

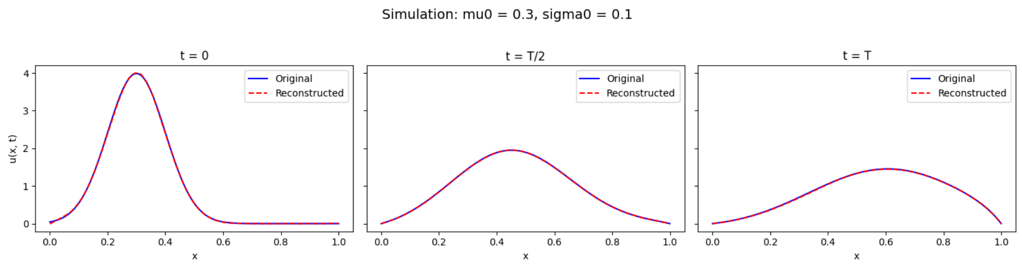


Figure 4: Autoencoder's reconstruction of PDE snapshots at different times. [8]

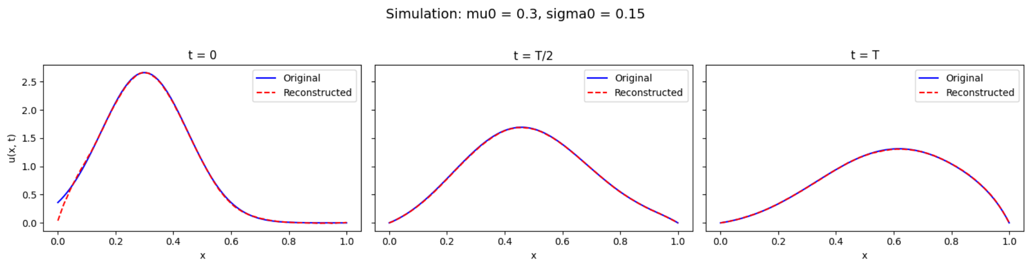


Figure 5: Autoencoder's reconstruction of PDE snapshots at different times. [9]

## 6.2 Symbolic Decoder and SINDy-Like Expressions

Next, we trained the symbolic decoder by defining a library of candidate functions (polynomials, sines, cosines, exponentials of $z_1$ $z_2$) and applying an $L_1$ penalty. Over many epochs (e.g, 10,000), a large graction of the learned coefficients were driven to zero. The final printed expressions for each output index $i \in [1, Nx]$ show that, while some terms remain (often polynomials and trigonometric functions), many are removed. Although these discovered expressions can look dense, they are more interpretable than an opaque neural network. In a perfect scenario, if the PDE solutions were strictly parametrized by $\mu$ and $\sigma$, we might see simpler polynomials or exponentials. However, the method still approximates well even in a purely data-driven context.

### 6.2.1 Training Loss

Once the autoencoder was trained, we froze the encoder and collected latent vectors for each PDE snapshot. We then trained the symbolic decoder with an $L_1$ penalty.

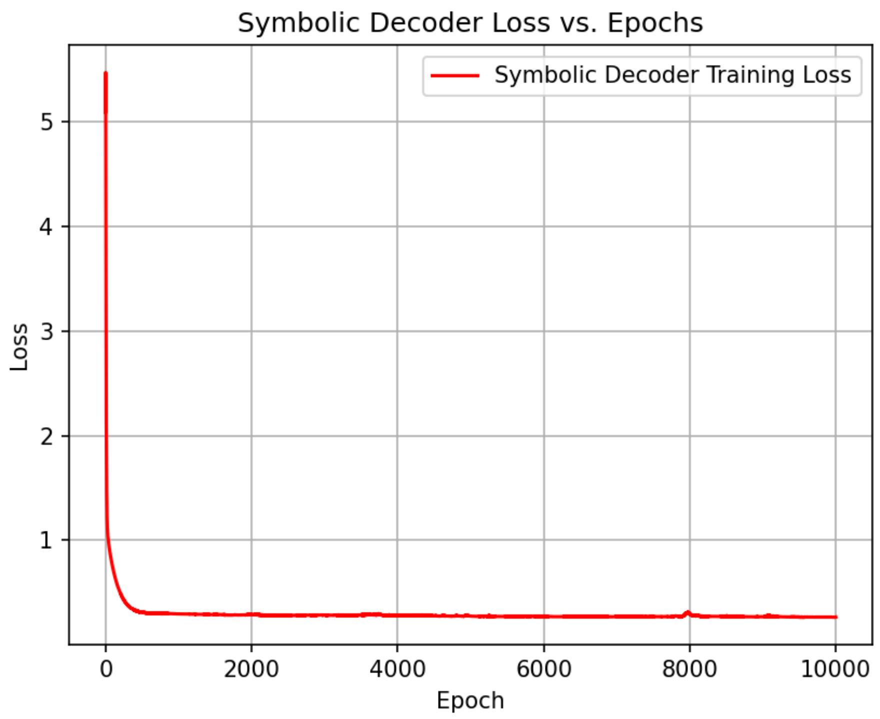


Figure 6: Training loss evolution for the symbolic decoder, including both MSE and $L_1$ penalty. [10]

### 6.2.2 SINDy Reconstructions

To visualize the performance of the symbolic decoder, we compare the ground truth PDE snapshots with the symbolic decoder's reconstruction in Figures 7, 8 and 9.

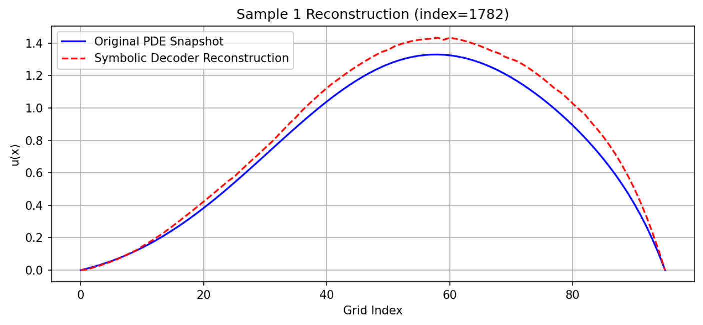


Figure 7: Ground truth (blue) vs. SINDy reconstruction (red). [11]

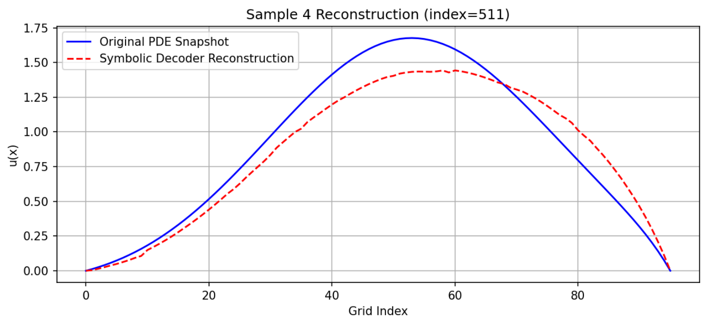


Figure 8: Ground truth (blue) vs. SINDy reconstruction (red). [12]

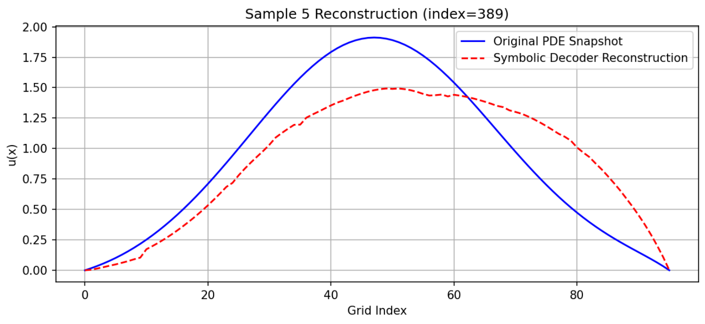


Figure 9: Ground truth (blue) vs. SINDy reconstruction (red). [13]

## 7 Prospects

Several avenues remain open for further investigation:

1. **Higher-dimensional PDEs:** Extending to 2D or 3D would require convolutional autoencoders with Conv2d or Conv3d, but the pipeline remains similar.
2. **Library Engineering:** Carefully choosing or learning the candidate function library can improve the final symbolic expressions. One could include known basis functions for advection-diffusion or focus on simpler polynomials.
3. **Combining PDE Knowledge:** Hybrid approaches can embed partial PDE knowledge into the autoencoder or the symbolic regression step, potentially yielding more physically grounded expressions.
4. **Real-time Control:** If real-time or multi-query scenarios demand rapid PDE evaluation, these reduced-order models could provide near-instantaneous predictions once trained.

## 8 Conclusion

In this project, we combined convolutional autoencoders and symbolic regression (SINDy-like methods) to build a reduced-order model for the 1D advection-diffusion equation with Gaussian initial conditions. The autoencoder effectively compresses each PDE snapshot into two latent dimensions, capturing the essential “mean” and “variance” dynamics. The symbolic decoder then attempts to reconstruct the PDE fields from these latent variables using a sparse combination of candidate functions. The results are promising in terms of both accuracy and partial interpretability.

This methodology illustrates a data-driven approach for discovering simpler models of complex PDEs. It can be extended to multi-dimensional problems, other PDEs, or real-time parameter exploration. Further work could refine the symbolic regression library and examine the generalization properties of such hybrid networks under more challenging PDE settings.

## 9 References

## References

[1] E. Franck. Scientific machine learning (sciml) master course, 2024.

[2] Pieter Wesseling. *Principles of Computational Fluid Dynamics*, volume 29 of *Springer Series in Computational Mathematics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[3] Fangcao Xu, Guido Cervone, Gabriele Franch, and Mark Salvador. Multiple geometry atmospheric correction for image spectroscopy using deep learning. *Journal of Applied Remote Sensing*, 14(02):024518, 2020. Figure: *autoencoder-CNN-structure.png* taken from this paper.

[4] Steven L. Brunton, Joshua L. Proctor, and J. Nathan Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 113(15):3932–3937, 2016.

[5] David Skinner. Green's functions for pdes. Lecture Notes, Mathematical Methods, University of Cambridge, 2024. Available at `http://www.damtp.cam.ac.uk/user/dbs26/1BMethods/All.pdf`.

[6] Giulio Carpi Lapi and Abdou Wade. Generated plot of the loss evolution of the autoencoder over epochs. Generated using Python and Matplotlib, 2025.

[7] Giulio Carpi Lapi and Abdou Wade. Generated plot of the autoencoder reconstruction. Generated using Python and Matplotlib, 2025.

[8] Giulio Carpi Lapi and Abdou Wade. Generated plot of the autoencoder reconstruction. Generated using Python and Matplotlib, 2025.

[9] Giulio Carpi Lapi and Abdou Wade. Generated plot of the autoencoder reconstruction. Generated using Python and Matplotlib, 2025.

[10] Giulio Carpi Lapi and Abdou Wade. Generated plot of the loss evolution of the sindy decoder over epochs. Generated using Python and Matplotlib, 2025.

[11] Giulio Carpi Lapi and Abdou Wade. Generated plot of the symbolic decoder's reconstruction. Generated using Python and Matplotlib, 2025.

[12] Giulio Carpi Lapi and Abdou Wade. Generated plot of the symbolic decoder's reconstruction. Generated using Python and Matplotlib, 2025.

[13] Giulio Carpi Lapi and Abdou Wade. Generated plot of the symbolic decoder's reconstruction. Generated using Python and Matplotlib, 2025.