UE22AM351A - AOML Course Project

Harnessing Machine Learning for Physics: A Synergistic Approach with PINNs and KANs in Solving Complex PDEs

Broccubali
Namita Achyuthan PES1UG22AM100
Shusrith S PES1UG22AM155
Siddhi Zanwar PES1UG22AM161

Outline

- Introduction
- Data Generation
- Physics-Informed Neural Networks (PINNs)
- Kolmogorov-Arnold Networks (KANs)
- Concluding Statements

Purpose:

- The primary goal is to demonstrate how integrating **physical laws into neural network training** enhances the **modeling of dynamic systems**.
- By using PINNs, we can **derive solutions that respect the underlying physics** without needing extensive datasets.
- KANs, on the other hand, excel in **approximating complex functions** while effectively **removing noise** from data.

Objectives:

- **Data Generation**: Illustrate methods for creating synthetic datasets with both clean and noisy samples, crucial for training our models.
- **PINNs**: Explain the architecture and training processes of PINNs, focusing on how they incorporate PDEs into their loss functions.
- **KANs**: Introduce KANs as a robust framework for function approximation and noise removal in high-dimensional spaces.

Section 1
Generation of Noisy Partial Differential Equations to Break PINN

- **Importance of Data:**
  - Essential for training machine learning models, especially in physics-based applications.
- **Types of Data:**
  - Clean Data: Accurate representations of solutions.
  - Noisy Data: Introduces variability to simulate real-world conditions
    - **Skewed Normal Noise**: Asymmetrical distribution affecting data points.
    - **Exponential Noise**: Rapidly decreasing probability of extreme values.

Initial Conditions

- **Initialization Modes:**
  - **Sine Wave:** Represents periodic behavior.
  - **Gaussian Distribution:** Common in statistical modeling.
  - **Step Function:** Useful for sudden changes in conditions.
  - **Positive Sine Wave:** Ensures all values remain positive.
  - **Double Sine Wave:** Combines multiple frequencies for complexity

```python
def init(xc, mode="sin", u0=1.0, du=0.1):
    """
    :param xc: cell center coordinate
    :param mode: initial condition
    :return: 1D scalar function u at cell center
    """
    modes = ["sin", "sinsin", "Gaussian", "react", "possin"]
    assert mode in modes, "mode is not defined!!"
    if mode == "sin":  # sinusoidal wave
        u = u0 * jnp.sin((xc + 1.0) * jnp.pi)
    elif mode == "sinsin":  # sinusoidal wave
        u = jnp.sin((xc + 1.0) * jnp.pi) + du * jnp.sin((xc + 1.0) * jnp.pi * 8.0)
    elif mode == "Gaussian":  # for diffusion check
        t0 = 0.01
        u = jnp.exp(-(xc**2) * jnp.pi / (4.0 * t0)) / jnp.sqrt(2.0 * t0)
    elif mode == "react":  # for reaction-diffusion eq.
        logu = -0.5 * (xc - jnp.pi) ** 2 / (0.25 * jnp.pi) ** 2
        u = jnp.exp(logu)
    elif mode == "possin":  # sinusoidal wave
        u = u0 * jnp.abs(jnp.sin((xc + 1.0) * jnp.pi))
    return u
```

Adding Noise to the Data

- In our project, we introduced noise into the generated datasets to simulate real-world conditions and enhance the robustness of our model.
- **Noise types:**
  - **Skewed Normal:**
    - Characterized by its asymmetrical distribution
    - Generated using a skewness parameter that controls the degree of asymmetry
  - **Exponential Noise:**
    - This noise follows an exponential distribution- this is useful for models that exhibit rapid decay.
    - It adds variability to the data. This is useful in scenarios where extreme values are less likely

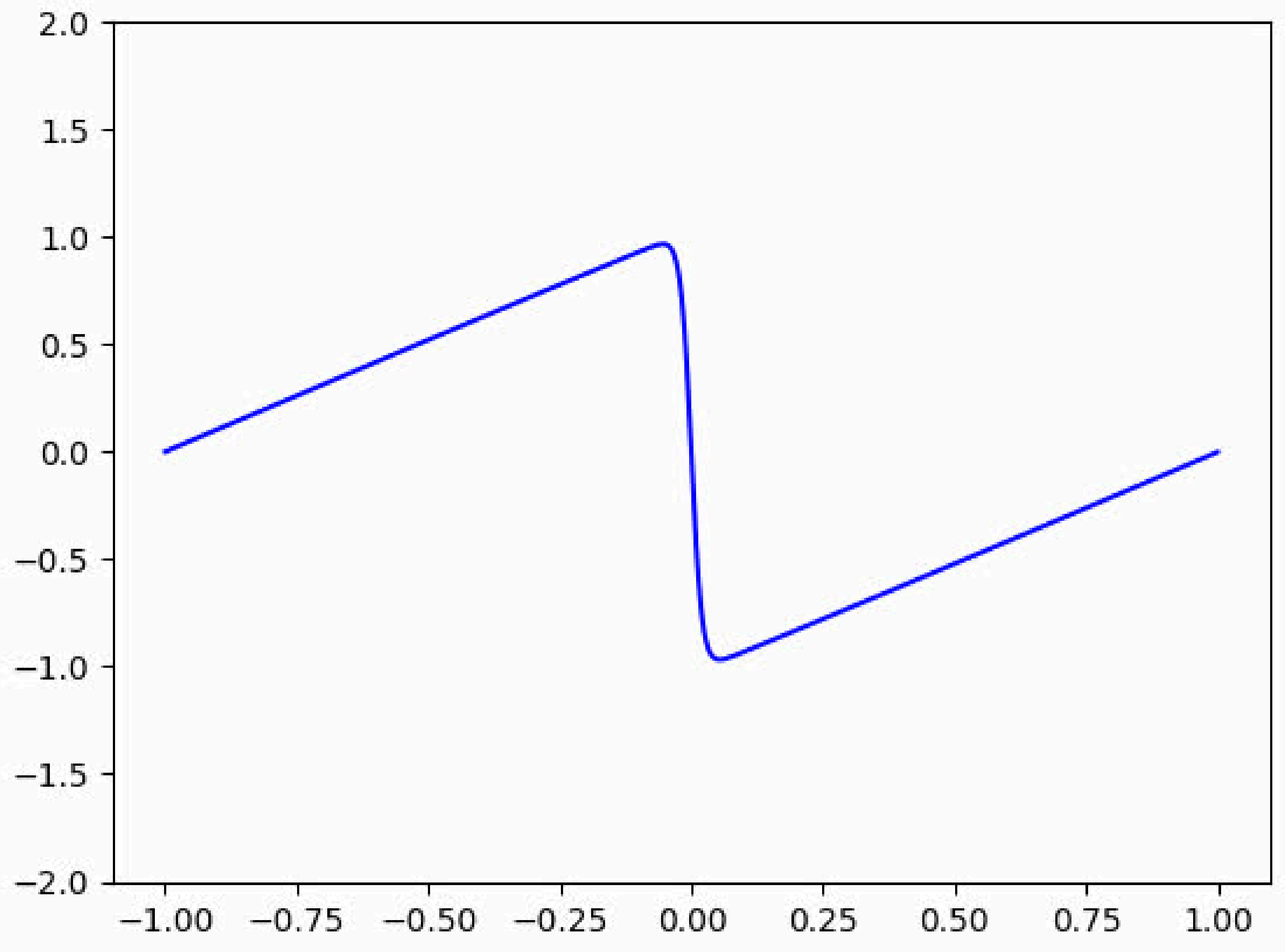
Adding Noise to the Data: Implementation

- A **random selection mechanism** is employed to choose between skewed normal and exponential noise for each data point.
- This **stochastic approach** ensures a **diverse range of noise** characteristics across the dataset.
- The noise is then added to the clean solution data generated from the PDEs, resulting in a noisy dataset that **retains the underlying structure** of the original signal while incorporating **realistic perturbations.**

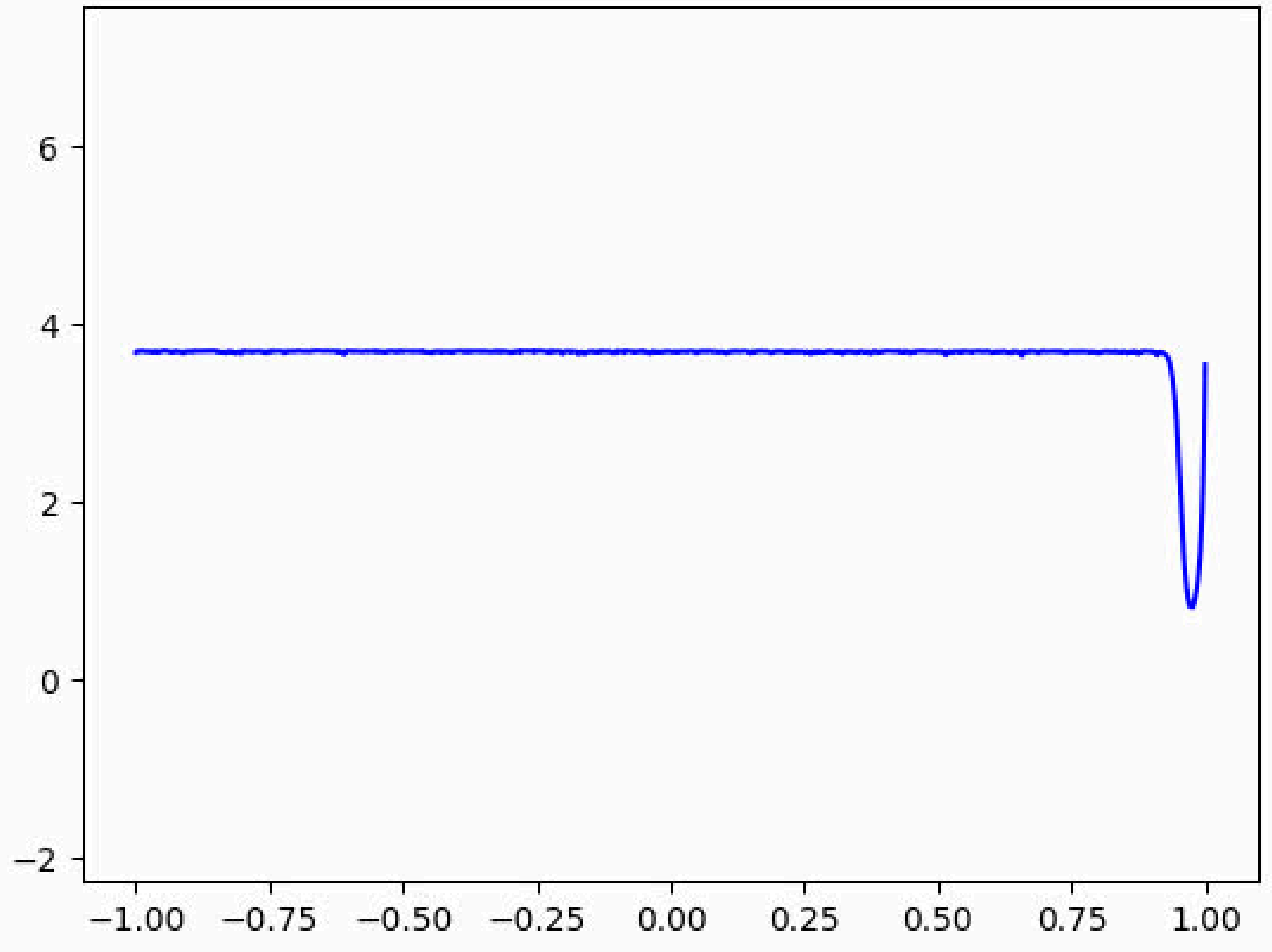
```
def generate_noise(shape, noise_level):
    a = np.random.rand()
    if a < 0.5:
        parameter = np.random.uniform(-5, 5)
        return skewnorm.rvs(a=parameter, size=shape) * noise_level
    else:
        parameter = np.random.uniform(0, 4)
        return np.random.exponential(scale=parameter, size=shape) * noise_level
```

Outcome of Section 1: Noise Addition

Clean Data



Noisy Data



Section 2
Building a Physics-Informed Neural Network (PINN) Just to Break It Again with Noisy Data

- **What is PINN? Why use it?**
  - A neural network that incorporates physical laws as constraints during training.
  - Effective in solving PDEs in fields like fluid dynamics and heat transfer- equations that help model the real world

- **Model structure:**
  - Fully connected layers with tanh activation functions.
  - Output layer designed to predict continuous values based on inputs (spatial and temporal coordinates).

Physics Informed Loss Function

- Components of Loss Function:
  - **Gradient Calculation**: Derivatives computed via tf.GradientTape.
  - **PDE Residual**: Ensures the model's output satisfies the PDE.
  - **Initial Condition Loss**: Measures accuracy at t=0.
  - **Boundary Condition Loss**: Enforces constraints at domain boundaries.

```python
def physics_informed_loss(
    model, x, t, initial_condition, boundary_condition, du, epsilon
):
    with tf.GradientTape(persistent=True) as tape:
        tape.watch([x, t])
        epsilon_tensor = tf.fill(x.shape,  tf.constant(epsilon, dtype=tf.float32))
        u = model(tf.concat([x, t, epsilon_tensor], axis=1))

        u_t = tape.gradient(u, t)
        u_x = tape.gradient(u, x)
        u_xx = tape.gradient(u_x, x)

    residual = u_t + u * u_x - epsilon * u_xx

    # Initial condition loss
    initial_loss = tf.reduce_mean(
        tf.square(u[tf.equal(t, tf.reduce_min(t))] - initial_condition)
    )

    # Periodic boundary condition loss
    periodic_loss = tf.reduce_mean(
        tf.square(u[tf.equal(x, tf.reduce_min(x))] - u[tf.equal(x, tf.reduce_max(x))])
    ) + tf.reduce_mean(
        tf.square(
            u_x[tf.equal(x, tf.reduce_min(x))] - u_x[tf.equal(x, tf.reduce_max(x))]
        )
    )

    # Residual loss
    residual_loss = tf.reduce_mean(tf.square(residual))

    return initial_loss + periodic_loss + residual_loss
```

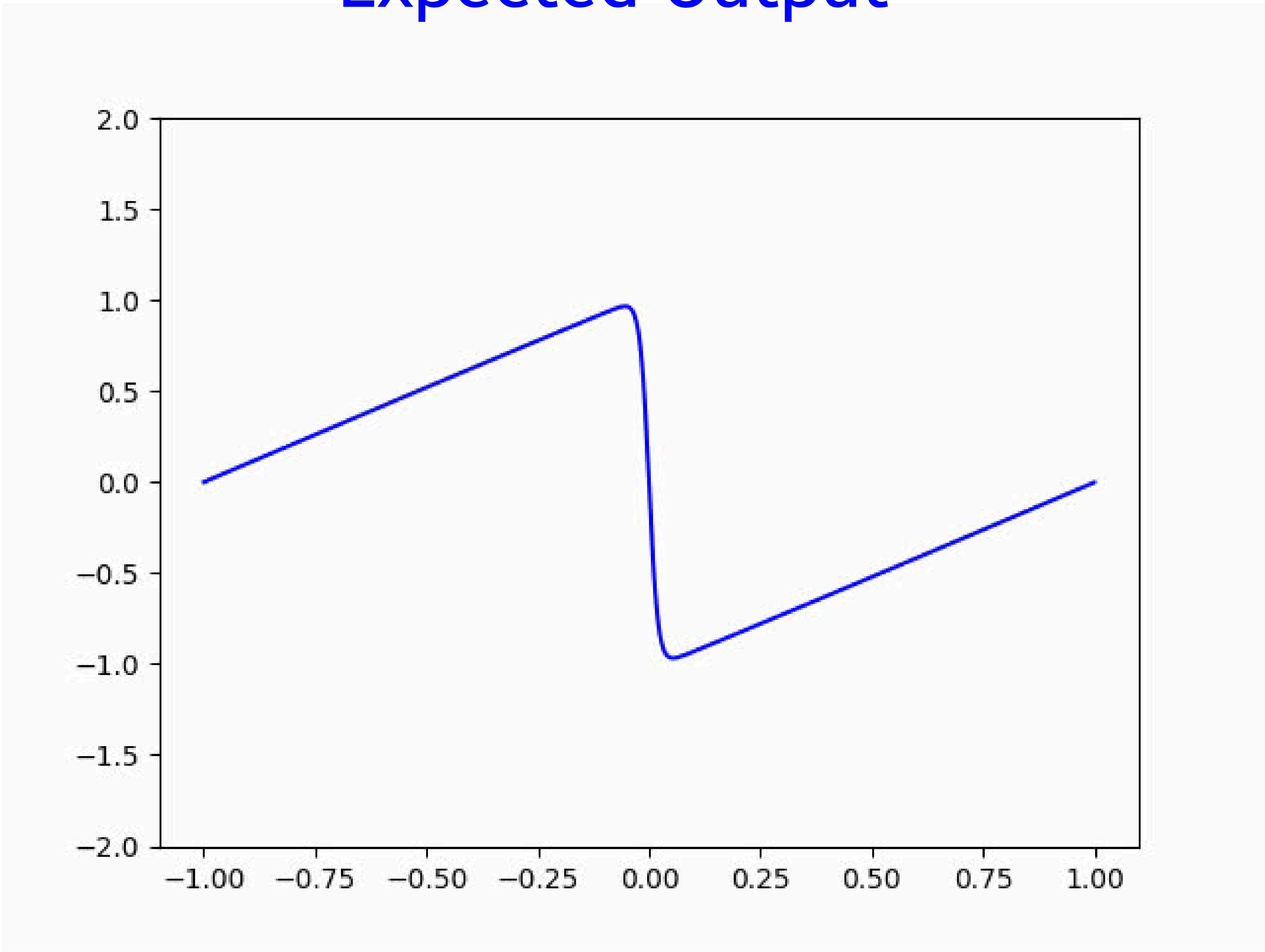
Training Loop

```python
def train_model(
    model,
    x,
    t,
    initial_condition,
    boundary_condition,
    du,
    epsilon,
    epochs,
    learning_rate,
):
    x_grid, t_grid = tf.meshgrid(x[:, 0], t[:, 0])
    x_flat = tf.reshape(x_grid, [-1, 1])
    t_flat = tf.reshape(t_grid, [-1, 1])

    optimizer = tf.keras.optimizers.Adam(learning_rate)

    for epoch in range(epochs):
        with tf.GradientTape() as tape:
            loss = physics_informed_loss(
                model,
                x_flat,
                t_flat,
                initial_condition,
                boundary_condition,
                du,
                epsilon,
            )
        grads = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))

        if epoch % 100 == 0:
            print(f"Epoch {epoch}, Loss: {loss.numpy():.6f}")
```

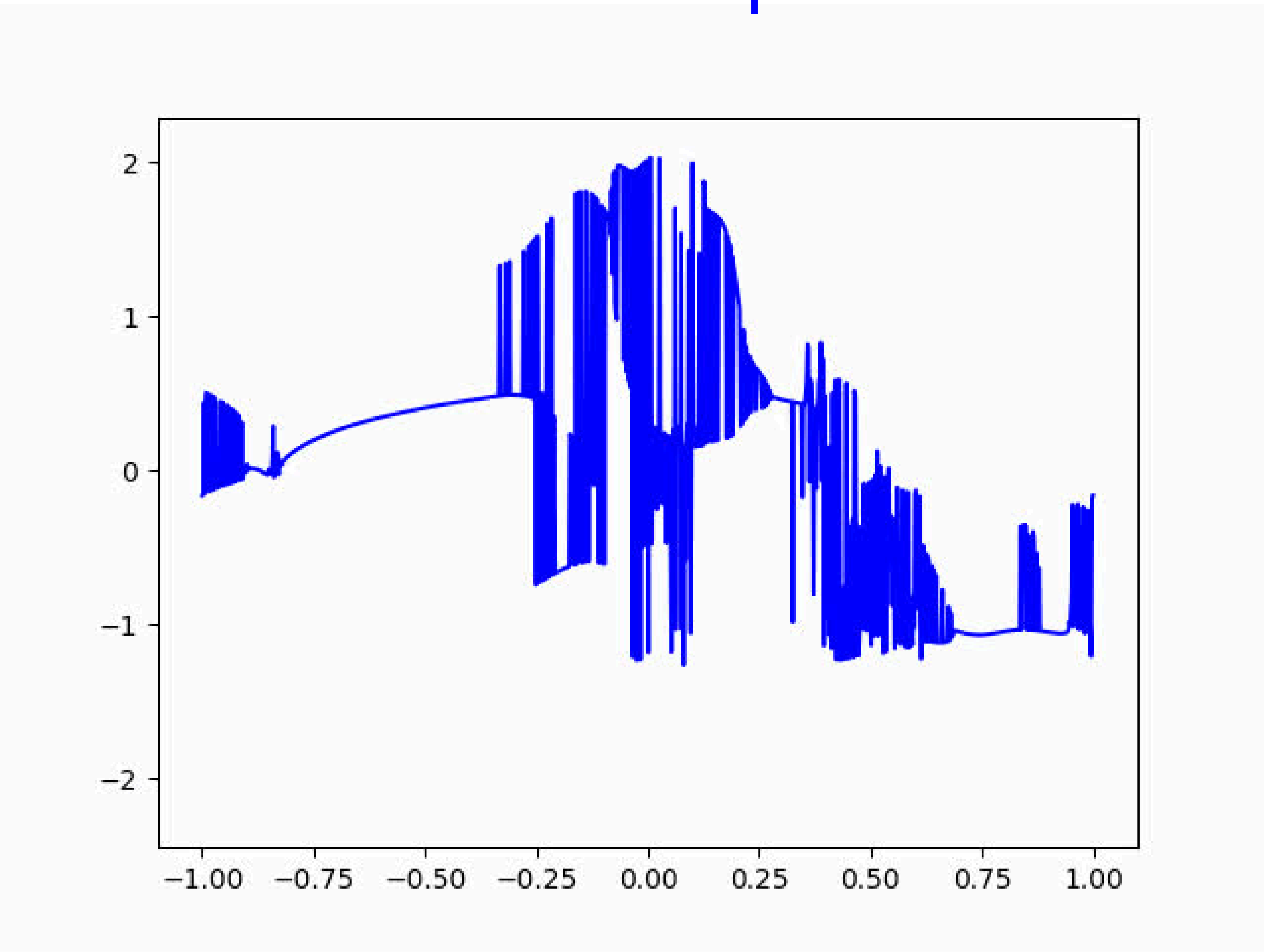
- **Step 1:**
  - Grid creation for spatial and time values.
- **Step 2:**
  - Setup of Adam optimizer with specified learning rates.
- **Step 3:**
  - Iterative training that updates model parameters based on computed losses.

Outcome of Section 2: Breaking a PINN with Noisy Data

Expected Output



PINN’s Output



Section 3

Using a Kolmogorov-Arnold Network (KAN) to Clean the Noisy, Incorrect Solution Generated by the PINN

- **What is KAN? Why use it?**
  - They're networks that are based on the Kolmogorov-Arnold representation theorem, allowing complex functions to be expressed as sums of simpler functions.
  - This adaptability is crucial when working with real-world, noisy data, leading to more accurate predictions from the PINN model
- **Model structure:**
  - **Function decomposition:** High-dimensional mappings are broken down into one-dimensional functions, enhancing approximation capabilities.

Denoising With KAN

- We found that KANs effectively **separate structured signals** from noise by learning **deterministic relationships** inherent in the data governed by PDEs.

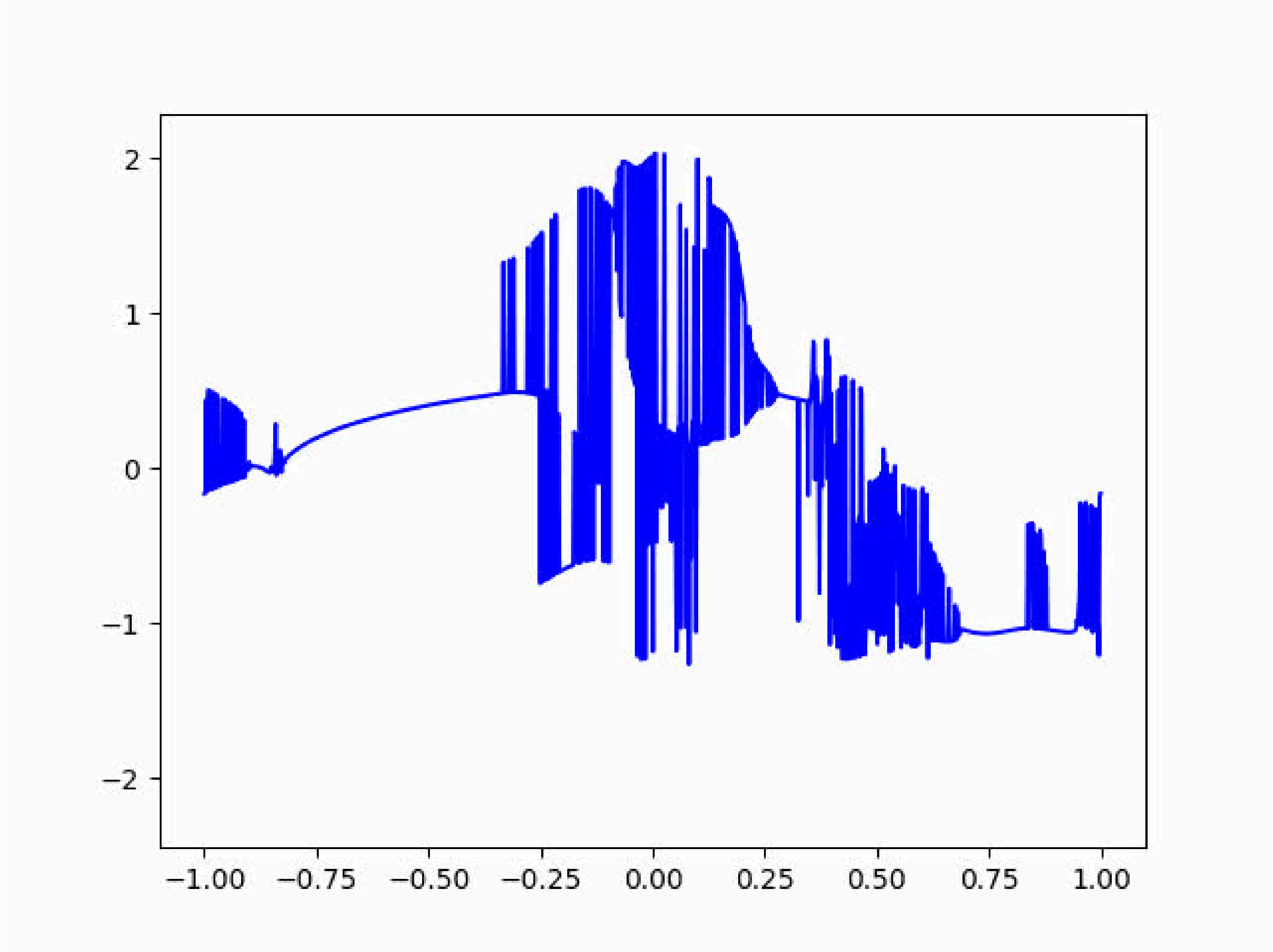
- The key components of our implementation include:
  - **B-Spline Basis Calculation** for smooth interpolation.
  - **Regularization techniques** to prevent overfitting during training.

```python
class KAN(torch.nn.Module):
    def __init__(
        self,
        layers_hidden,
        grid_size=5,
        spline_order=3,
        scale_noise=0.1,
        scale_base=1.0,
        scale_spline=1.0,
        base_activation=torch.nn.SiLU,
        grid_eps=0.02,
        grid_range=[-1, 1],
    ):
        super(KAN, self).__init__()
        self.grid_size = grid_size
        self.spline_order = spline_order

        self.layers = torch.nn.ModuleList()
        for in_features, out_features in zip(layers_hidden, layers_hidden[1:]):
            self.layers.append(
                KANLinear(
                    in_features,
                    out_features,
                    grid_size=grid_size,
                    spline_order=spline_order,
                    scale_noise=scale_noise,
                    scale_base=scale_base,
                    scale_spline=scale_spline,
                    base_activation=base_activation,
                    grid_eps=grid_eps,
                    grid_range=grid_range,
                )
            )
```

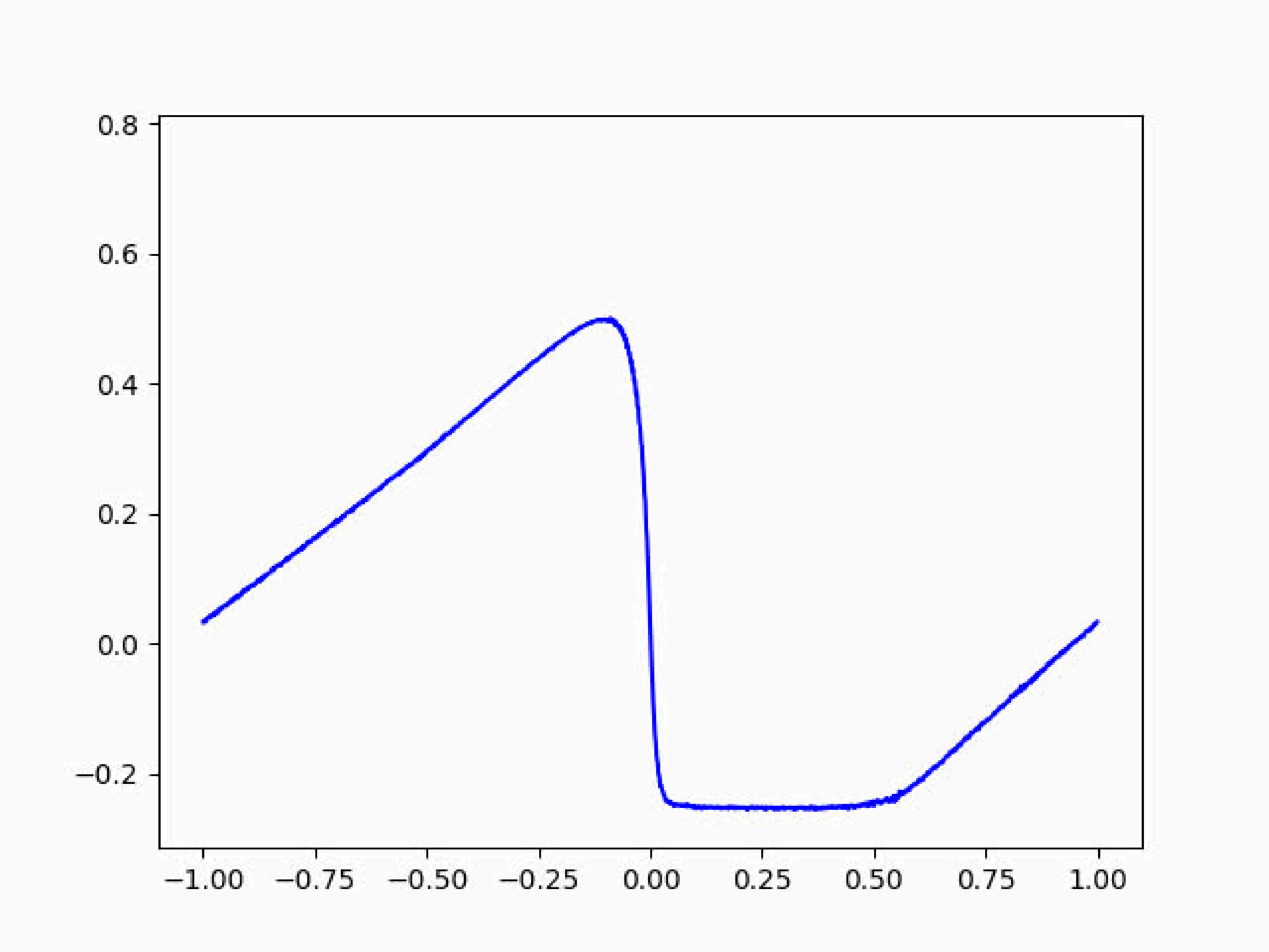
Outcome of Section 3: Cleaning the output of the broken PINN

PES UNIVERSITY

PINN's Output

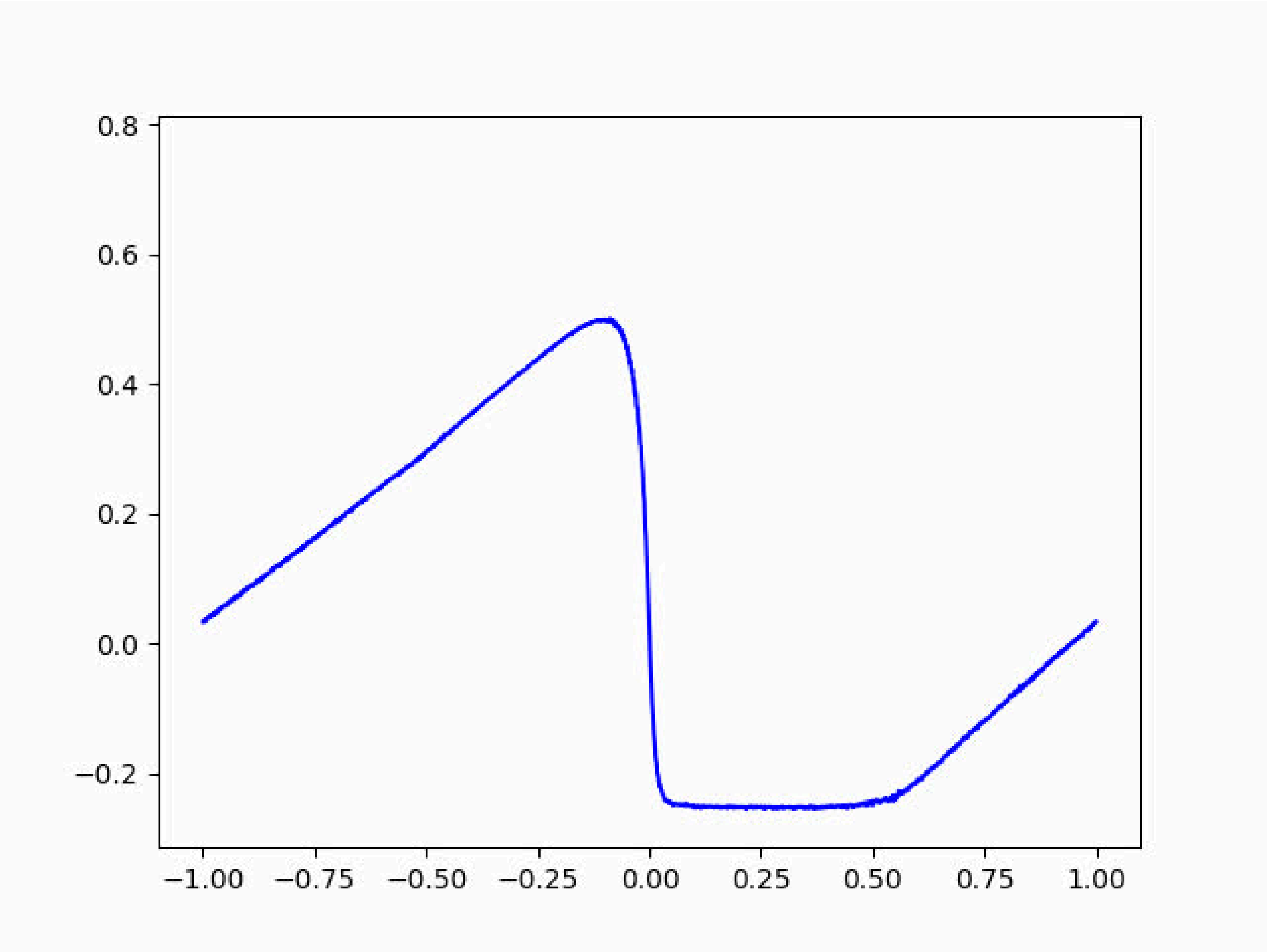


KAN's Output After Cleaning

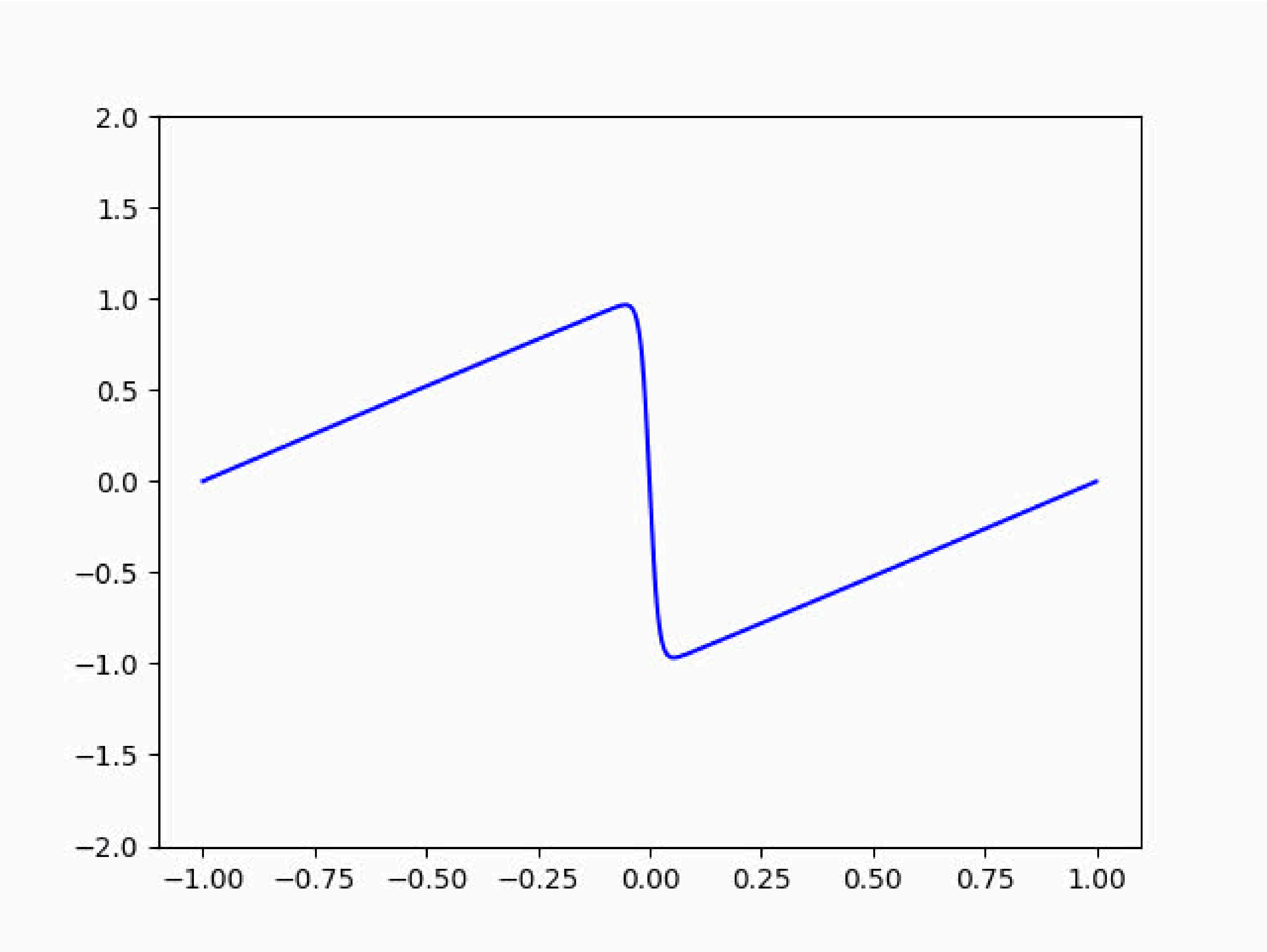


Outcome of Section 3: Cleaning the output of the broken PINN

KAN's Output



Expected Output



Section 4
Concluding Statements

Key Insights

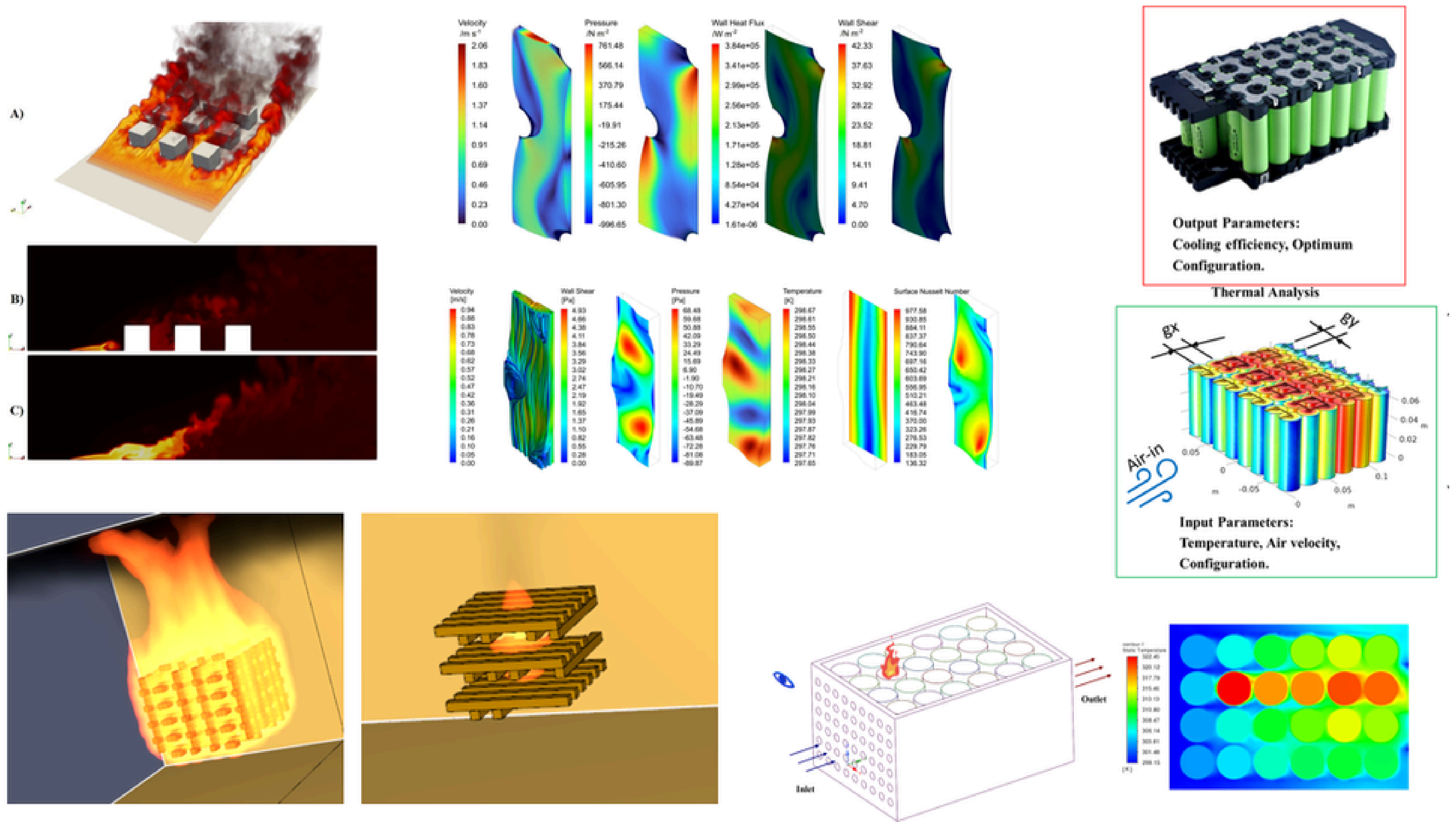
In this presentation, we explored the integration of Physics-Informed Neural Networks (PINNs) and Kolmogorov-Arnold Networks (KANs) as powerful tools for solving partial differential equations (PDEs) in the presence of noise and complex dynamics.

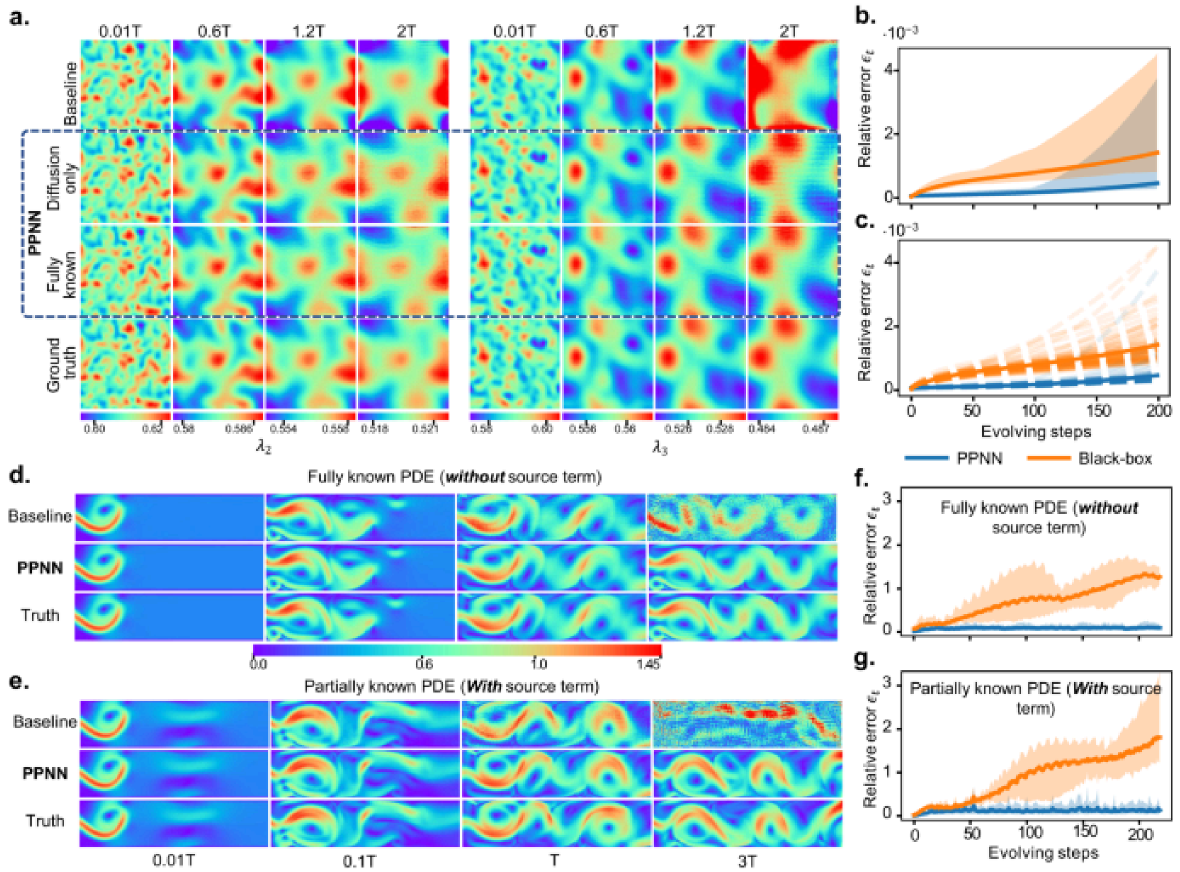
Here's what we learnt:

- PINNs effectively incorporate physical laws into the learning process, enabling accurate solutions to PDEs without extensive datasets.
  - By enforcing initial and boundary conditions, they ensure that the solutions adhere to the underlying physics.
- KANs provide a robust framework for approximating multivariate functions while effectively filtering out noise.
  - Their ability to decompose complex relationships into simpler components enhances their performance in recovering clean solutions from noisy data.

Practical Implications

- The methodologies presented offer significant advancements in computational modeling across various fields, including fluid dynamics, heat transfer, and other engineering applications.
- They enable researchers and practitioners to tackle real-world problems more efficiently, even with limited or corrupted data.





Future Directions

- Continued exploration of these techniques can lead to further improvements in accuracy and efficiency.
- Future work may include expanding the application of PINNs and KANs to more complex systems and integrating them with other machine learning approaches for enhanced predictive capabilities.

References

- PDEBench Repository
- Efficient Kolmogorov-Arnold Networks
- KINN: A physics-informed deep learning framework for solving forward and inverse problems
- KAN-ODEs: Kolmogorov-Arnold network ordinary differential equations for learning dynamical systems
- Physics-informed Neural Networks with Unknown Measurement Noise

View Our Work

- Broccubali Github
- AOML Submission