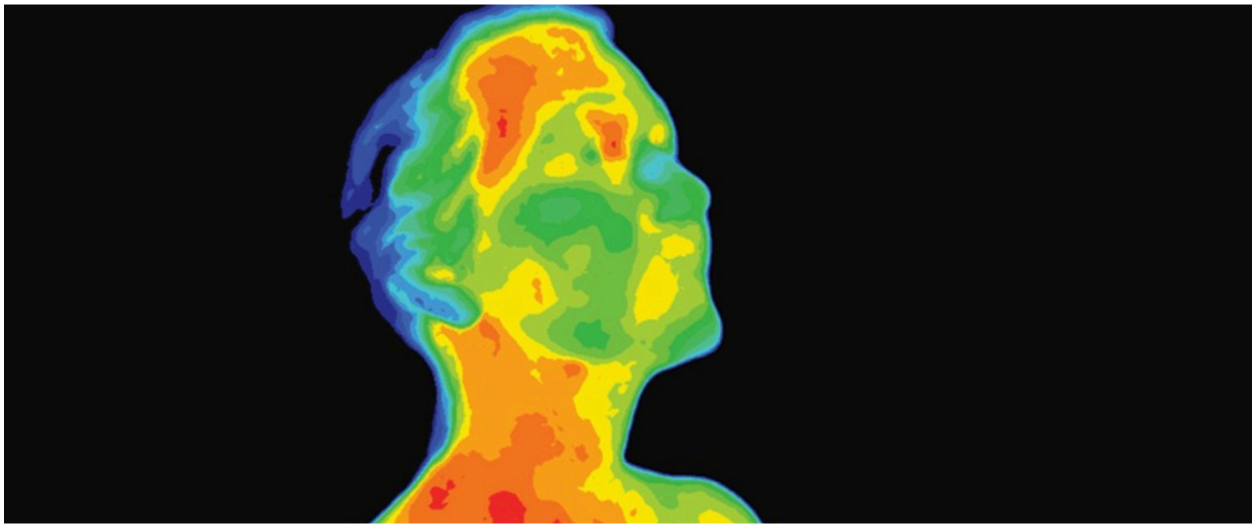


Determining Temperature Profile of a Metal Plate using Physics Informed Neural Networks (PINNs) - [when one side of the plate is continuously heated]

In this article, we explore how to solve the lid-driven cavity problem using **Physics-Informed Neural Networks (PINNs)**. PINNs integrate the governing physical laws, expressed as partial differential equations (PDEs), into the training process of neural networks. This approach allows us to approximate the solution to PDEs without the need for traditional discretization methods like finite elements or finite volumes.

Introduction to PINNs

Physics-Informed Neural Networks (PINNs) are a class of neural networks that incorporate physical laws into the learning process. Instead of solely relying on data, PINNs embed the governing equations (e.g., Navier-Stokes equations for fluid flow) into the loss function. This means that the network is trained not just to fit data but also to satisfy the underlying physics.

Advantages of using PINNs include:

- **Mesh-free Solutions**: No need for mesh generation or grid discretization.
- **Flexibility**: Can handle complex geometries and boundary conditions.
- **Generalization**: Potential to generalize solutions beyond the training data.

Problem Statement

We aim to determine the temperature profile $T(x, y, t)$ of a metal plate over time using Physics-Informed Neural Networks (PINNs). The metal plate is modeled as a two-dimensional domain $(x, y) \in [0,1] \times [0,1]$ over a simulation time interval $t \in [0, 40]$ seconds. The heat conduction within the plate is governed by the transient heat equation. One side of the plate $(x = 0,\ y = 0)$ is continuously heated to a fixed temperature of $T_{\text{heated}} = 100^{\circ}\text{C}$ while the rest of the boundaries are insulated (i.e., no heat flux crosses these boundaries). Initially, the plate is uniformly at $T_{\text{initial}} = 20^{\circ}\text{C}.$

Governing Equations and Boundary Conditions

Governing Equation

The heat conduction in the plate is described by the 2D transient heat equation:

$$\partial T/\partial t = \alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right)$$

where the thermal diffusivity α is defined by

$\alpha = \frac{k}{\rho C_p}$

with:

- ρ as the density (kg/m^3),
- Cp as the specific heat capacity $(J/(kg \cdot K))$,
- k as the thermal conductivity $(W/(m \cdot K))$.

Boundary Conditions

1. **Dirichlet Boundary Condition (Heated Side):**

   On the left side ($x = 0$):

   $T(0, y, t) = 100 \circ C$

2. **Neumann Boundary Conditions (Insulated Boundaries):**

   The remaining boundaries are insulated, which implies zero heat flux:

   - At the right boundary $(x = 1)$:

     x=1x=1

     $\frac{\partial T}{\partial x}(1, y, t) = 0$

   - At the bottom $(y = 0)$:

     $\frac{\partial T}{\partial y}(x, 0, t) = 0$

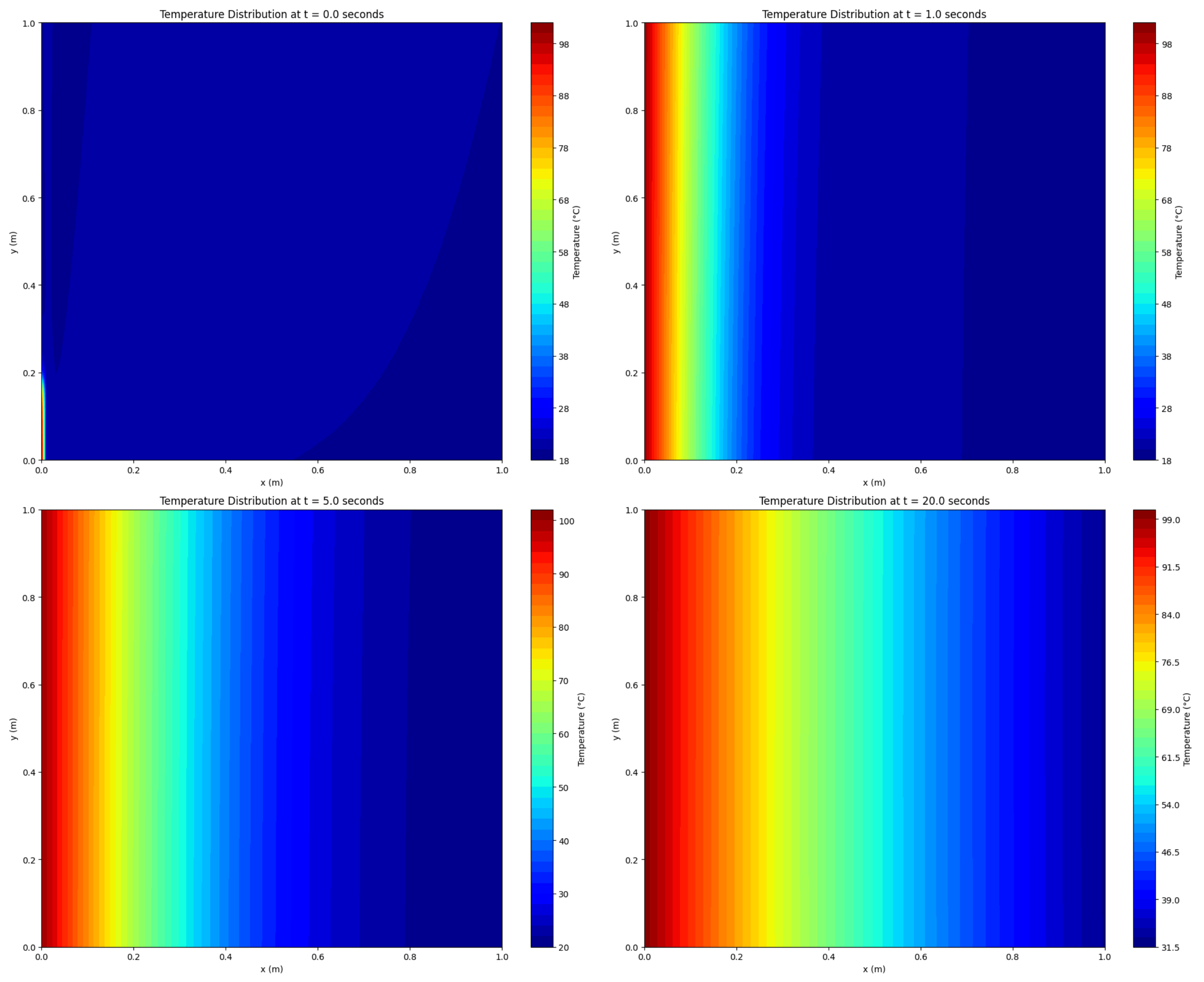
- At the top $(y = 1)$:

  $\frac{\partial T}{\partial y}(x, 1, t) = 0$

Initial Condition

At t=0, the temperature is uniformly:

$$T(x, y, 0) = T_{\text{initial}} = 20°\text{C} \quad \text{for all } (x, y) \in [0, 1] \times [0, 1].$$



Implementation Details

We implement the solution using TensorFlow to construct and train the neural network. We use **Google Colab** as our platform for this project.

Below is an example of a problem statement—with the governing equations and boundary conditions—followed by a brief explanation of the code.

Below is a detailed, part-by-part explanation of the code. This explanation breaks down each segment to show how the PINN for heat conduction is implemented.

1. GPU Configuration and Random Seed Setup

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import os

# Configure GPU (if available)
physical_devices = tf.config.list_physical_devices('GPU')
if physical_devices:
    try:
        for device in physical_devices:
            tf.config.experimental.set_memory_growth(device, True)
        print("GPU is available and will be used.")
    except RuntimeError as e:
        print(e)
else:
    print("No GPU available. Running on CPU.")

# Set random seeds
tf.random.set_seed(42)
np.random.seed(42)
```

- **GPU Configuration:**

  The code first checks for available GPU devices. If a GPU is detected, it enables memory growth (so that TensorFlow allocates memory as needed rather than pre-allocating all of it). This ensures efficient use of resources.

- **Random Seeds:**

  Setting the random seeds for both `TensorFlow` and `NumPy` guarantees that results are reproducible. Every run will generate the same random numbers, which is important for debugging and comparing experiments.

2. Problem Parameters

```
# Problem parameters
rho = 8000.0       # Density (kg/m³) [Example for steel]
C_p = 0.466       # Specific heat capacity (J/(kg·K)) [Example for steel]
k = 45.0          # Thermal conductivity (W/(m·K)) [Example for steel]
alpha = k / (rho * C_p)  # Thermal diffusivity (m²/s)
```

```
T_heated = 100.0   # Temperature at heated side (°C)
T_initial = 20.0   # Initial temperature (°C)

# Domain boundaries
x_min, x_max = 0.0, 1.0  # Plate dimensions (meters)
y_min, y_max = 0.0, 1.0
t_min, t_max = 0.0, 20.0 # Simulation time (seconds)
```

- **Physical Parameters:**

  Here, the code defines the properties of the metal plate (steel in this example) by setting density, specific heat capacity, and thermal conductivity.

  The thermal diffusivity α is computed from these properties.

- **Temperature Conditions:**

  The heated boundary is maintained at $100 \circ C$ while the initial temperature of the plate is set to $20 \circ C$.

- **Domain and Time:**

  The spatial domain is a unit square$[0,1] \times [0,1]$ and the simulation runs from $t = 0$ to $t = 20$t seconds.

3. Loading or Generating Training Data

```
# Load training data if available
try:
    with np.load('training_data.npz') as data:
        X_f = tf.constant(data['X_f'], dtype=tf.float32)
        X_b = tf.constant(data['X_b'], dtype=tf.float32)
        T_b = tf.constant(data['T_b'], dtype=tf.float32)
        X_i = tf.constant(data['X_i'], dtype=tf.float32)
        T_i = tf.constant(data['T_i'], dtype=tf.float32)
    print("Training data loaded from training_data.npz")
except FileNotFoundError:
    print("Training data file not found. Generating new data...")
    # ... (Your existing code for generating training data) ...
```

- **Data Reuse:**

  The code attempts to load pre-generated training data from a compressed file (`training_data.npz`). If the file exists, it avoids re-generating the data, which saves time.

- **Fallback:**

  If the file isn't found, the code prints a message indicating that new data will be generated. (The data generation code follows below.)

4. Neural Network Definition

```
# Neural network definition
class HeatConductionPINN(tf.keras.Model):
    def __init__(self, num_hidden_layers=8, num_neurons_per_layer=40, **kwargs):
        super(HeatConductionPINN, self).__init__(**kwargs)
        self.input_layer = tf.keras.layers.InputLayer(input_shape=(3,)) # (x, y, t)
        self.hidden_layers = tf.keras.Sequential([
            tf.keras.layers.Dense(num_neurons_per_layer,
                                  activation='tanh',
                                  kernel_initializer='glorot_normal')
            for _ in range(num_hidden_layers)
        ])
        self.temp_output = tf.keras.layers.Dense(1, activation=None)

    def call(self, inputs, **kwargs):
        x = self.hidden_layers(inputs)
        return self.temp_output(x)
```

- **Model Architecture:**
  The custom
  Keras model `HeatConductionPINN` is defined to predict the temperature T.
  - **Input Layer:** Accepts 3 inputs: x, y and t.
  - **Hidden Layers:** A sequential stack of 8 fully connected layers with 40 neurons each using tanh activation.
  - **Output Layer:** Produces a single output representing the temperature.

This network serves as a function approximator for the solution$T(x, y, t)$.

---

5. Generating Training Data

Collocation Points

```
# Collocation points (x, y, t)
N_f = 10000  # Collocation points
x_f = tf.random.uniform((N_f, 1), x_min, x_max, dtype=tf.float32)
y_f = tf.random.uniform((N_f, 1), y_min, y_max, dtype=tf.float32)
t_f = tf.random.uniform((N_f, 1), t_min, t_max, dtype=tf.float32)
X_f = tf.concat([x_f, y_f, t_f], axis=1)
```

- **Purpose:**
  These points cover the entire space–time domain where the PDE (heat equation) is enforced. The neural network’s predictions at these points must satisfy the governing heat equation.

Boundary Points (Heated Side)

```
# Boundary points (heated left side at x=0)
N_b = 2000  # Boundary points
x_left = x_min * tf.ones((N_b, 1), dtype=tf.float32)
y_left = tf.random.uniform((N_b, 1), y_min, y_max, dtype=tf.float32)
t_left = tf.random.uniform((N_b, 1), t_min, t_max, dtype=tf.float32)
X_b = tf.concat([x_left, y_left, t_left], axis=1)
T_b = T_heated * tf.ones((N_b, 1), dtype=tf.float32)
```

- **Heated Boundary:**
  The code generates points along the left boundary (where $x = 0$) and assigns a fixed temperature of $100^\circ\mathrm{C}$. This is the Dirichlet condition for the heated side.

Initial Condition Points

```
# Initial condition points (t=0)
N_i = 5000  # Initial condition points
x_i = tf.random.uniform((N_i, 1), x_min, x_max, dtype=tf.float32)
y_i = tf.random.uniform((N_i, 1), y_min, y_max, dtype=tf.float32)
t_i = t_min * tf.ones((N_i, 1), dtype=tf.float32)
X_i = tf.concat([x_i, y_i, t_i], axis=1)
T_i = T_initial * tf.ones((N_i, 1), dtype=tf.float32)
```

- **Initial Condition:**
  Points across the plate at time $t = 0$ are generated, with the temperature set to the initial temperature $20 \circ C$.

---

6. Loss Function Definition

```
@tf.function
def heat_loss_fn(model, X_f, X_b, T_b, X_i, T_i):
    with tf.GradientTape(persistent=True) as tape:
        # First forward pass for predictions at collocation points
        tape.watch(X_f)
        T_pred = model(X_f)
        T_pred = tf.squeeze(T_pred)  # Ensure scalar output

        # First derivatives
        grads = tape.gradient(T_pred, X_f)
        dT_dx = grads[:, 0]
        dT_dy = grads[:, 1]
        dT_dt = grads[:, 2]
        # Second derivatives (using persistent tape)
        try:
            d2T_dx2 = tape.gradient(dT_dx, X_f)[:, 0]
            d2T_dy2 = tape.gradient(dT_dy, X_f)[:, 1]
        except TypeError:
            print("Second derivative computation failed, using fallback")
            d2T_dx2 = tf.zeros_like(dT_dx)
            d2T_dy2 = tf.zeros_like(dT_dy)
        finally:
            del tape  # Manually delete persistent tape

    # Physics equation residual: dT/dt - alpha*(d2T/dx2 + d2T/dy2) = 0
    pde_residual = dT_dt - alpha * (d2T_dx2 + d2T_dy2)
    pde_loss = tf.reduce_mean(tf.square(pde_residual))

    # Dirichlet Boundary Condition loss (heated side)
    T_pred_b = model(X_b)
    bc_loss = tf.reduce_mean(tf.square(T_pred_b - T_b))

    # Initial Condition loss
    T_pred_i = model(X_i)
    ic_loss = tf.reduce_mean(tf.square(T_pred_i - T_i))

    return pde_loss + bc_loss + ic_loss
```

- **Automatic Differentiation:**

  The loss function uses `tf.GradientTape` to compute first and second derivatives of the network output with respect to its inputs (x, y, t).

  - **First Derivatives:** ∂T/∂x, ∂T/∂y, and ∂T/∂t.
  - **Second Derivatives:**

    $\frac{\partial^2 T}{\partial x^2}$ and

    $\frac{\partial^2 T}{\partial y^2}$.

- **PDE Residual:**

  The residual of the heat equation is computed as

  $$\text{residual} = \frac{\partial T}{\partial t} - \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right)$$

  and its mean square error is minimized.

- **Boundary and Initial Losses:**

  Additional terms penalize the difference between predicted and prescribed temperatures at the heated boundary and the initial condition.

7. Model Initialization, Optimizer, and Training Loop

```
# Initialize model and optimizer
model = HeatConductionPINN()
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)

@tf.function
def train_step():
    with tf.GradientTape() as tape:
        loss = heat_loss_fn(model, X_f, X_b, T_b, X_i, T_i)
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
    return loss

# Training loop
def train_model(epochs=10000):
    for epoch in range(epochs):
        loss = train_step()
        if epoch % 100 == 0:
            print(f"Epoch {epoch}, Loss: {loss.numpy():.5f}")
```

- **Training Step:**

  Each training step computes the total loss (PDE, boundary, and initial conditions) and then updates the model parameters using the Adam optimizer.

- **Training Loop:**

  The loop runs for a specified number of epochs (here, 10,000), printing the loss every 100 epochs to monitor progress.

8. Saving the Training Data

```
# Save training data
np.savez_compressed('training_data.npz',
                    X_f=X_f.numpy(), X_b=X_b.numpy(), T_b=T_b.numpy(),
                    X_i=X_i.numpy(), T_i=T_i.numpy())
print("Training data saved to training_data.npz")
```

- **Purpose:**
  The generated training data (collocation, boundary, and initial points) is saved to a compressed file. This allows you to reload the data in future runs without needing to re-generate it.

9. Visualization of Heat Distribution

```
def visualize_heat_distribution(model):
    # Create spatial grid
    x = np.linspace(x_min, x_max, 100)
    y = np.linspace(y_min, y_max, 100)
    X, Y = np.meshgrid(x, y)

    # Time points for visualization
    time_points = [0.0, 1.0, 5.0, t_max] # t=0, intermediate times, and steady-state

    plt.figure(figsize=(20, 16))
    for i, t in enumerate(time_points):
        # Create input grid with constant time
        T = t * np.ones_like(X).flatten()[:, None]
        XY = np.hstack([X.flatten()[:, None], Y.flatten()[:, None], T])

        # Predict temperatures
        T_pred = model(tf.constant(XY, dtype=tf.float32))
        T_plot = T_pred.numpy().reshape(X.shape)

        # Create subplot for each time point
        plt.subplot(2, 2, i+1)
        plt.contourf(X, Y, T_plot, levels=50, cmap='jet')
        plt.colorbar(label='Temperature (°C)')
        plt.title(f'Temperature Distribution at t = {t} seconds')
        plt.xlabel('x (m)')
        plt.ylabel('y (m)')

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

- **Visualization Grid:**

  A spatial grid over the plate is created using NumPy's `meshgrid`.

- **Time Points:**

  The code selects four time instants (initial, two intermediate, and steady-state) at which the temperature field is visualized.

- **Contour Plots:**

  For each time point, the network predicts the temperature on the grid and a contour plot is generated. The heated boundary is implicitly visible, and colorbars indicate the temperature scale.

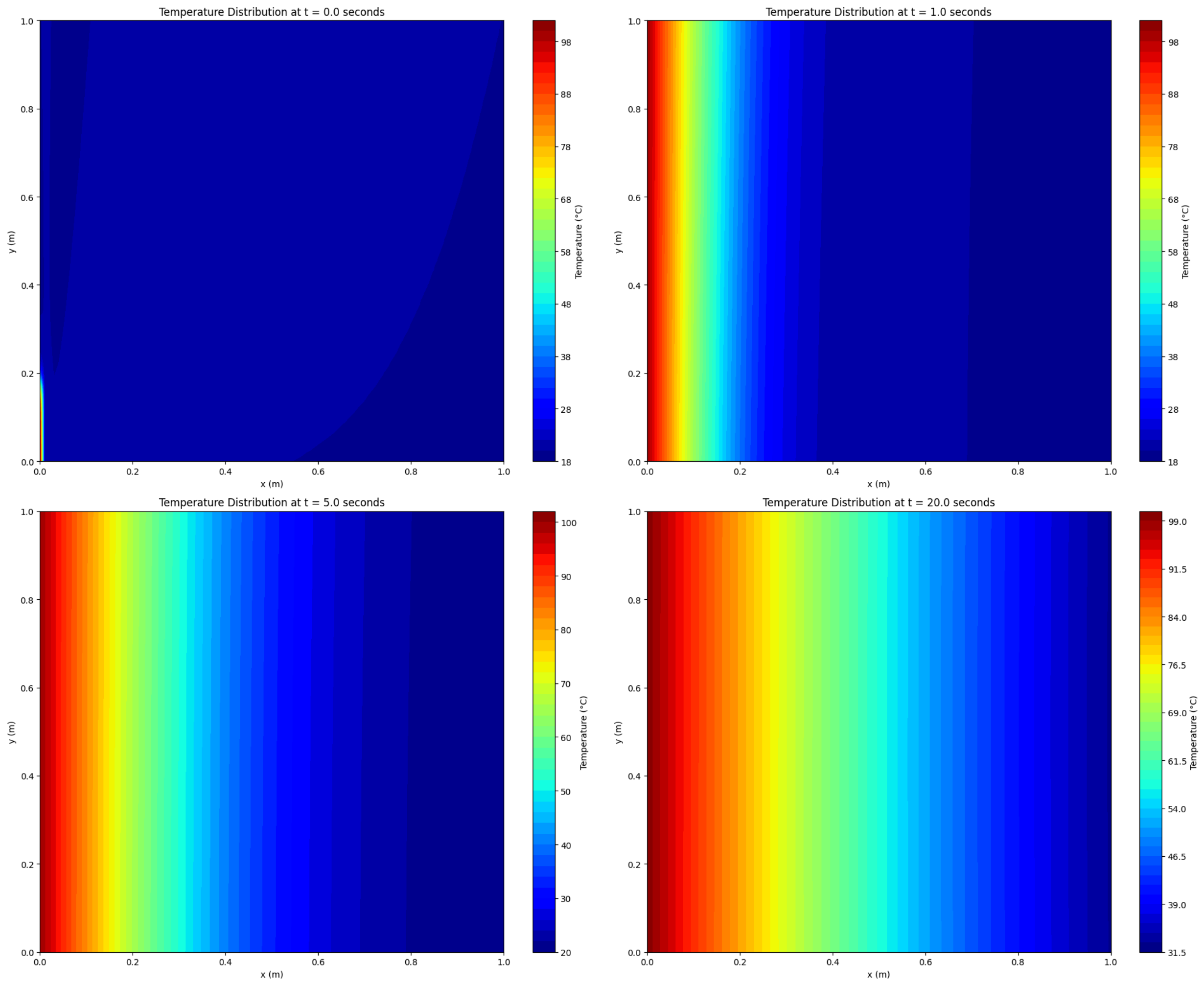
---

10. Running the Training and Visualization

```
# Train and visualize
train_model(30000)
visualize_heat_distribution(model)
```

- **Final Execution:**
  The model is trained for 30,000 epochs, and afterward, the trained model is used to generate and display contour plots of the temperature distribution at the selected time points.

Output



Analysis of the Temperature Distribution at Four Time Snapshots

The figure shows four contour plots of the temperature distribution in a 2D plate at different times $\{0.0, 1.0, 5.0, 20.0\}$ seconds. Here's what is happening in each subplot:

1. **Top-Left**$(t = 0.0s)$

- The plate is uniformly at the initial temperature of $20 \circ C$, which appears as a solid blue region.
- Since no heat has yet been conducted from the heated boundary, there is no temperature gradient in the plate at this instant.

2. **Top-Right ($t = 1.0s$)**
   - Heat has started diffusing from the heated boundary (on the left side), creating a narrow warmer zone (shown by a slight change in colour from blue to a faint gradient).
   - Most of the plate is still near the initial temperature $20 \circ C$, indicating that only a small amount of heat has penetrated into the domain in 1 second.

3. **Bottom-Left ($t = 5.0s$)**
   - A clearer temperature gradient has formed from left (hot) to right (cool).
   - The left boundary is significantly warmer (transitioning toward $100 \circ C$), while the rest of the plate remains cooler but is gradually heating up as the heat diffuses inward.
4. **Bottom-Right ($t = 20.0s$)**
   - By 20 seconds, a large portion of the plate has warmed, but there is still a distinct gradient from left (near $100°\mathrm{C}$) to right (closer to the initial temperature, though noticeably higher than at $t = 0$).
   - This suggests the system has not reached a uniform steady-state across the plate. Given enough time (and assuming all other boundaries remain insulated), the temperature would eventually become more uniform, but the simulation snapshot at $t = 20$s shows the process is still ongoing.

Key Insights:

- **Heat Diffusion Process:**

  The sequence of plots illustrates the classical diffusion behavior: heat flows from the heated boundary into the cooler regions, creating a moving "heat front" that gradually warms the plate.
- **Temperature Gradient Over Time:**

  Early in the simulation, the gradient is sharp and localized near the boundary. Over time, the gradient extends further into the plate.
- **Approach to Steady State:**

  Although 20 seconds is long enough to significantly heat the plate, there is still a noticeable difference between the left (heated) and right (insulated) edges. In a purely insulated domain (except for the heated boundary), it can take much longer for the entire plate to reach a nearly uniform temperature.

Overall, these four snapshots confirm that the PINN solution captures the expected physics of heat conduction: a uniform initial state, a gradually expanding thermal boundary layer, and an evolving temperature field that moves from the heated boundary into the interior of the plate.

This step-by-step explanation covers how the PINN is structured—from setting up the physical problem and generating training data to defining the neural network, computing the physics-based loss, training the model, and finally visualizing the resulting temperature distribution. This approach embeds the governing heat conduction equation and boundary/initial conditions directly into the loss function, allowing the network to learn a solution that is consistent with the underlying physics.