Using PINN to solve diffusion, wave and Laplace equations

ZHOUYI Xiaoxiao

Date: May 31, 2024

Abstract

In this paper, we use the Physics-Informed Neural Network (PINN) to solve the diffusion, wave, and Laplace equations. PINN is a deep learning-based approach that can solve partial differential equations (PDEs) without requiring a mesh or grid. We demonstrate the application of PINN to solve the 1D diffusion equation, 2D wave equation, and 2D and 3D Laplace equations. The PINN solutions are obtained and visualized to understand the behavior of these systems. Due to the absence of analytical or numerical solutions for comparison, the results for the wave and Laplace equations are included in the appendix. This allows us to focus on the methodology and visualization of the PINN solutions in the main text. The results show that PINNs can accurately approximate the solutions to these PDEs and satisfy the given initial and boundary conditions.

Keywords: PINN, diffusion equation, wave equation, Laplace equation

1 Introduction

1.1 Background

Partial differential equations (PDEs) are fundamental in describing a wide range of physical phenomena, such as heat conduction, wave propagation, and electromagnetism. Traditional numerical methods, such as finite difference methods, finite element methods, and spectral methods, often rely on mesh or grid discretization to solve PDEs. However, these methods can become computationally expensive and require significant memory, especially when dealing with high-dimensional problems, complex geometries, or irregular boundary conditions.

In recent years, deep learning has shown great potential in tackling complex problems. The Physics-Informed Neural Network (PINN) is a novel approach that integrates physical laws described by PDEs into the neural network framework. Unlike traditional methods, PINNs do not require a mesh or grid, allowing for more flexibility and efficiency in solving PDEs. PINNs leverage the power of deep learning to approximate solutions to PDEs by minimizing a loss function that encodes the residuals of the PDEs and the boundary and initial conditions.

1.2 Related Work

Several studies have demonstrated the effectiveness of PINNs in solving various types of PDEs. For instance, Raissi et al. (2019) introduced the concept of PINNs and applied it to solve forward and inverse problems for several PDEs, including the Navier-Stokes equations. Other researchers have extended this approach to different applications, such as fluid dynamics, structural mechanics, and biomedical engineering. These studies have shown that PINNs can achieve high accuracy and generalize well to different types of PDEs and boundary conditions.

1.3 Objective and Contribution

In this paper, we aim to apply the Physics-Informed Neural Network (PINN) to solve the diffusion, wave, and Laplace equations. We will demonstrate the implementation of PINNs using PyTorch and evaluate their performance in accurately approximating the solutions to these PDEs. Specifically, we will focus on the following contributions:

Diffusion Equation: We solve the one-dimensional diffusion equation with Dirichlet boundary conditions and compare the PINN solution with the analytical solution.

Laplace Equation: We extend our approach to solve the two-dimensional Laplace equation and analyze the results using both uniform and random sampling strategies.

By showcasing the application of PINNs to these fundamental PDEs, we aim to highlight the potential of this method in solving a wide range of PDEs encountered in scientific and engineering problems.

2 diffusion equation

In one space dimension, the diffusion equation with Dirichlet boundary conditions reads as

$$\begin{aligned} &u_t - u_{xx} = 0, \quad x \in [0,1], \quad t \in [0,1] \\ &u(0,x) = \sin(\pi x) \\ &u(t,0) = u(t,1) = 0 \end{aligned}$$

the analytical solution is:

$$u(t,x) = e^{-\pi^2 t}\sin(\pi x),$$

2.1 Physics-Informed Neural Network Implementation

We implemented the Physics-Informed Neural Network (PINN) using PyTorch. The neural network architecture consists of three fully connected layers with 128 neurons each and Tanh activation functions.

The training was performed using the Adam optimizer with a learning rate of 0.001 and a learning rate scheduler to adapt the learning rate during training.

2.2 Sampling Points

For training the PINN, we used the following sampling strategy:

- **Domain points**: 1000 points uniformly sampled from the domain $[0,1] \times [0,1]$.
- **Random points**: 500 random points sampled from the domain $[0,1] \times [0,1]$ at each training iteration to calculate the PDE residual.
- **Boundary condition points**: 100 points each for $x = 0$ and $x = 1$ uniformly sampled over the time domain $[0,1]$.
- **Initial condition points**: 100 points uniformly sampled from $x \in [0,1]$ at $t = 0$.

2.3 Training and Results

The training was conducted over 5000 epochs. The total loss comprises the mean squared error (MSE) of the PDE residual, the initial condition, and the boundary conditions. The results of the PINN solution are compared with the analytical solution. The MSE analysis shows how well the PINN approximates the analytical solution.

2.4 MSE Error Analysis

We analyzed the mean squared error (MSE) between the PINN solution and the analytical solution to evaluate the performance of the network. The MSE is calculated as follows:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(u_{\text{PINN}}(t_i, x_i) - u_{\text{analytical}}(t_i, x_i))^2$$

where N is the number of sampling points, u_{PINN} is the solution obtained from the PINN, and $u_{\text{analytical}}$ is the analytical solution. The MSE helps quantify the accuracy of the PINN in approximating the true solution of the diffusion equation.

2.5 Results with Uniform Sampling

2.6 Results with Random Sampling

2.7 Conclusion

The results of this study demonstrate the effectiveness of Physics-Informed Neural Networks (PINNs) in solving the one-dimensional diffusion equation. By comparing the uniform and random sampling methods, we can draw the following conclusions:

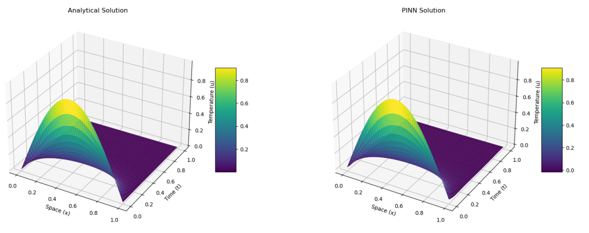


Figure 1: Analytical solution and PINN solution using uniform points.

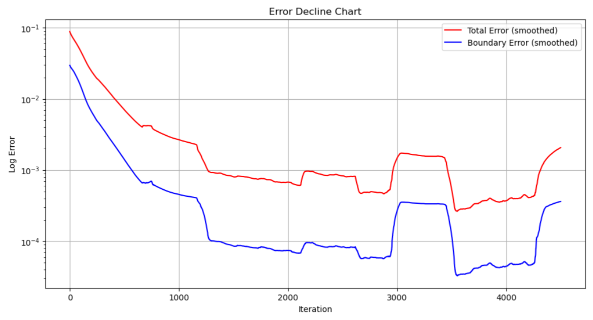


Figure 2: Error of PINN solution using uniform points.

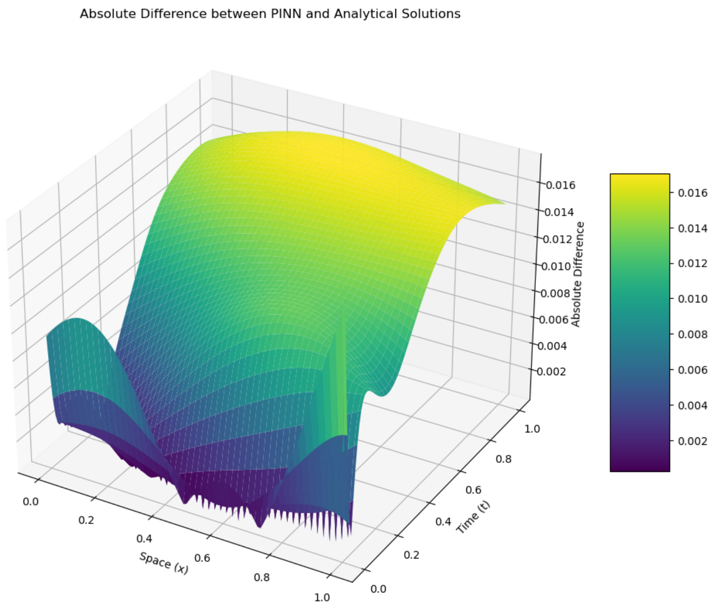


Figure 3: Difference error of PINN solution using uniform points.

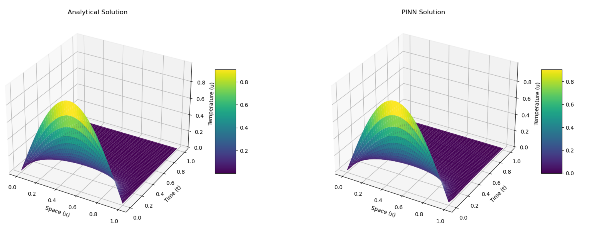


Figure 4: Analytical solution and PINN solution using random points.

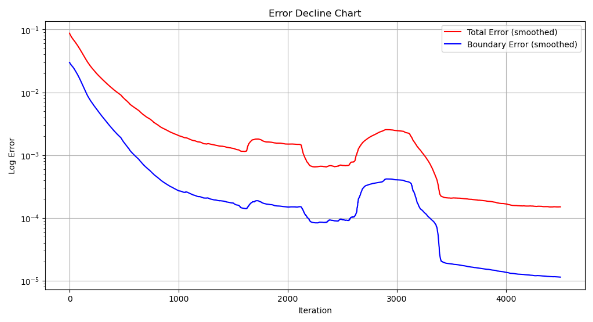


Figure 5: Difference error of PINN solution using random points.

1. **Accuracy of Solutions**: Both sampling methods yield solutions that closely match the analytical solution, as shown in Figures 1 and 4. This demonstrates that PINNs are capable of accurately

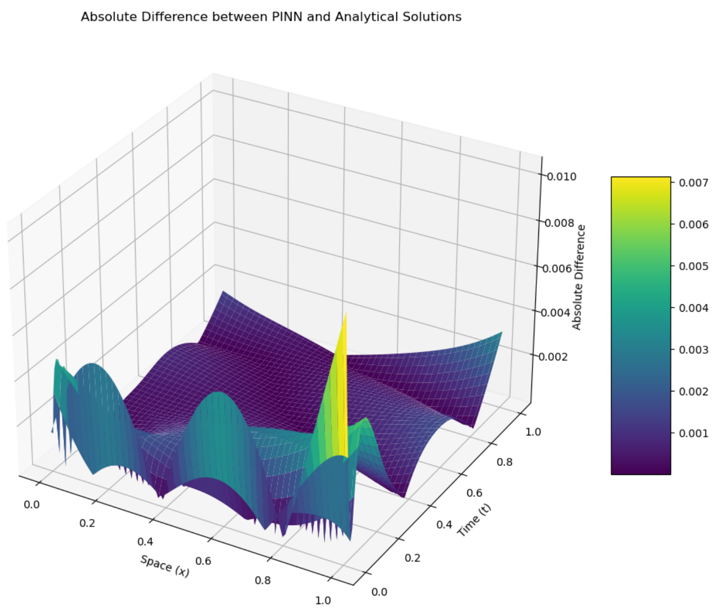


Figure 6: Error of PINN solution using random points.

approximating the solution to the diffusion equation.

2. **Error Analysis**: The error decline charts, depicted in Figures 2 and 5, indicate that the total error and boundary error decrease more smoothly with random sampling, suggesting a more stable and effective training process with random points.
3. **Difference Error**: The difference error plots in Figures 3 and 6 reveal that the random sampling method results in lower error variations compared to the uniform sampling method. This suggests that random sampling is more effective in capturing the complex dynamics of the solution.
4. **Convergence Behavior**: From the error decline charts (Figures 2 and5), it is evident that the random sampling method provides a more consistent and faster convergence compared to uniform sampling, indicating a more stable learning process.

In conclusion, while both uniform and random sampling methods are effective, random sampling tends to offer more consistent and stable results, leading to better overall performance in solving the diffusion equation with PINNs. Future work could explore hybrid sampling strategies to further enhance the accuracy and stability of PINN solutions.

3 2D Laplace Equation

3.1 Physics-Informed Neural Network Implementation

We implemented the Physics-Informed Neural Network (PINN) using PyTorch. The neural network architecture consists of three fully connected layers with 128 neurons each and Tanh activation functions.

The training was performed using the Adam optimizer with a learning rate of 0.001 and a learning rate scheduler to adapt the learning rate during training.

3.2 Sampling Points

For training the PINN, we used the following sampling strategy:

- **Interior points**: 10000 points sampled from the interior of the domain $[0, \pi] \times [0, \pi]$.
- **Boundary condition points**:
  - 100 points for $y = 0$ uniformly sampled from $x \in [0, \pi]$.
  - 100 points for $y = \pi$ uniformly sampled from $x \in [0, \pi]$.
  - 100 points for $x = 0$ uniformly sampled from $y \in [0, \pi]$.
  - 100 points for $x = \pi$ uniformly sampled from $y \in [0, \pi]$.

We use the equation $u(x, y)$ in the square $D = \{0 < x < \pi, 0 < y < \pi\}$ with the boundary conditions:

$$\begin{cases} u_y = 0 & \text{for } y = 0 \text{ and for } y = \pi \\ u = 0 & \text{for } x = 0 \\ u = \frac{1}{2}(1 + \cos 2y) & \text{for } x = \pi \end{cases}$$

By using the method of separation of variables, the analytic solution is:

$$u(x, y) = \frac{1}{2\pi}x + \frac{1}{2\left(e^{2\pi} - e^{-2\pi}\right)} \cos 2y \left(e^{2x} - e^{-2x}\right)$$

3.3 Results with Uniform Sampling

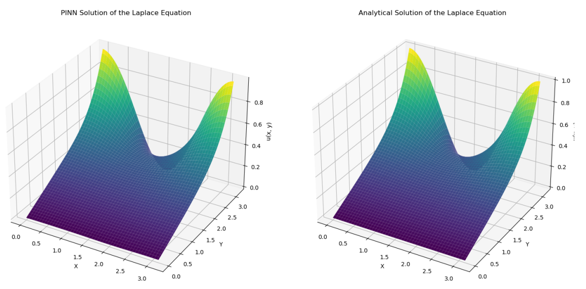


Figure 7: Analytical solution and PINN solution using uniform points.

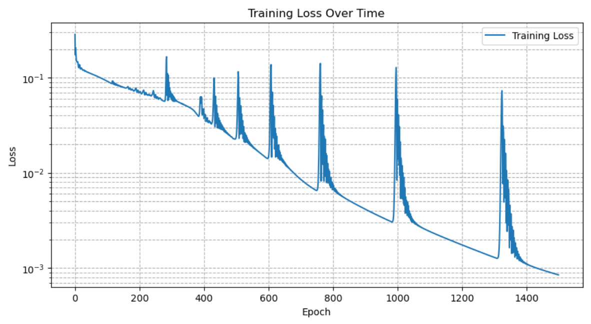


Figure 8: Loss during training for the solution $u(x, y)$.

3.4 Results with Random Sampling

3.5 Conclusion

The results of this study demonstrate the effectiveness of Physics-Informed Neural Networks (PINNs) in solving the two-dimensional Laplace equation. By comparing the analytical solution with the PINN solution,

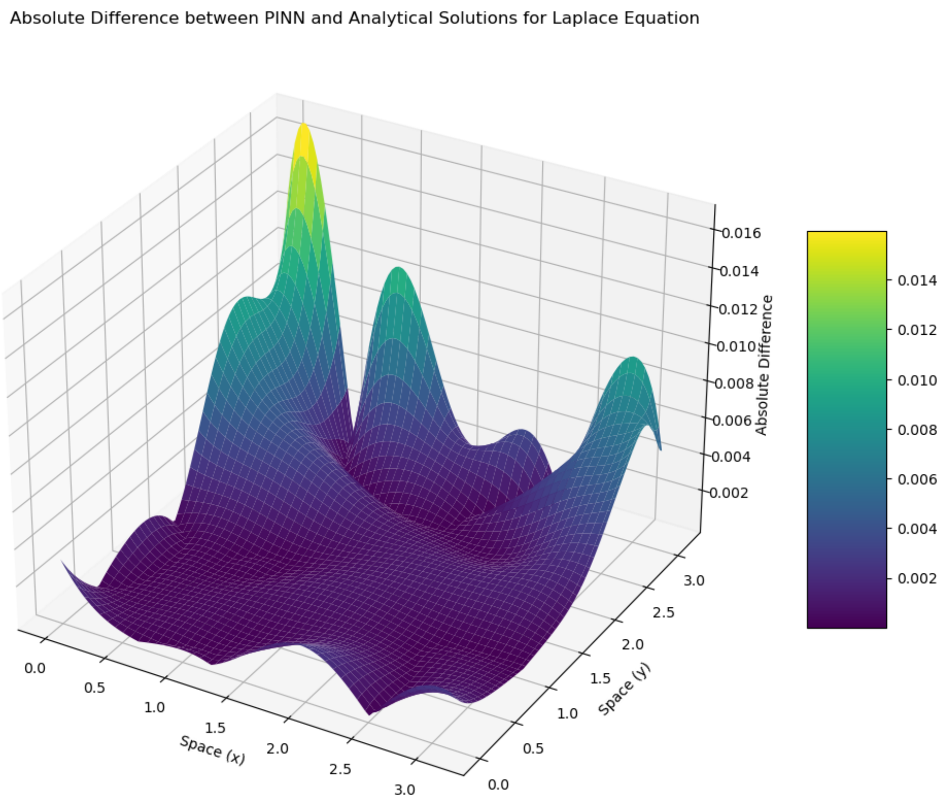


Figure 9: Difference between the analytical solution and the PINN solution $u(x,y)$ using uniform points.

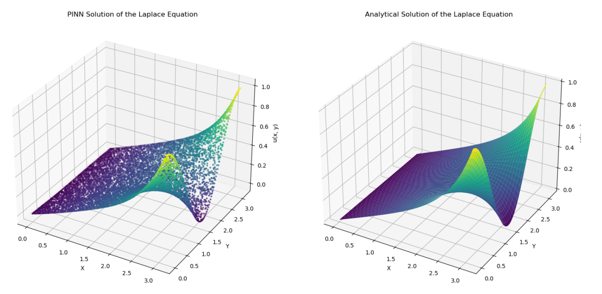


Figure 10: Analytical solution and PINN solution using random points.

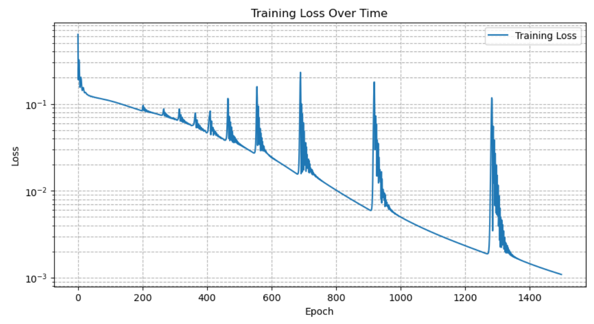


Figure 11: Loss during training for the solution $u(x,y)$ using random points.

as well as analyzing the training loss and the difference error, we can draw the following conclusions:

1. **Accuracy of Solutions**: The PINN solution closely matches the analytical solution, as shown in Figures 7 and 10. This demonstrates that PINNs are capable of accurately approximating the solution to the Laplace equation using both uniform and random sampling points.
2. **Error Analysis**: The difference error plots in Figures 9 and 12 reveal that the PINN solution has low error variations when using both uniform and random points, suggesting that both sampling methods can be effective in capturing the complex dynamics of the solution.

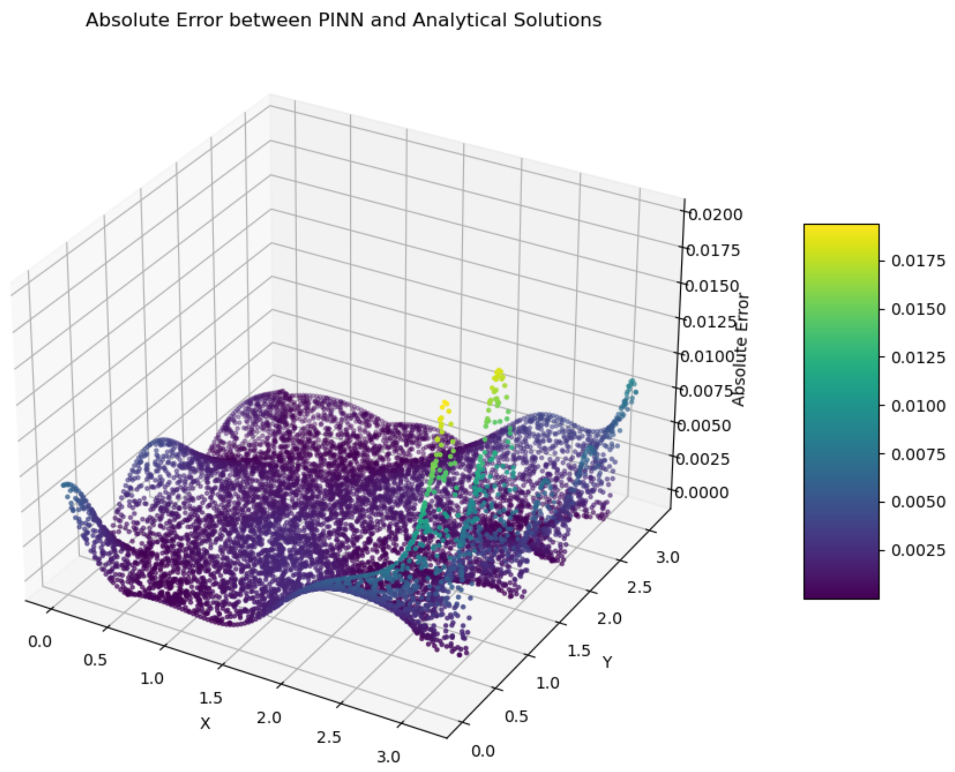


Figure 12: Difference between the analytical solution and the PINN solution $u(x, y)$ using random points.

3. **Convergence Behavior**: From the loss decline chart (Figures 8 and 11), it is evident that the total loss decreases smoothly over the training epochs for both uniform and random sampling, indicating a stable and effective training process.

In conclusion, PINNs provide a robust and flexible approach for solving PDEs like the Laplace equation. The results obtained using random sampling points are accurate and demonstrate the potential of PINNs for solving complex PDEs with various boundary conditions.

References

[1] Raissi, Maziar, Paris Perdikaris, and George Em Karniadakis. ”Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations.” *Journal of Computational Physics*, vol. 378, 2019, pp. 686-707. Elsevier.

[2] Kingma, Diederik P., and Jimmy Ba. ”Adam: A method for stochastic optimization.” *arXiv preprint arXiv:1412.6980*, 2014.

Appendices

In this section, we present the application of Physics-Informed Neural Networks (PINNs) to solve the 2D wave equation, the 3D Laplace equation, and the 10D Laplace equation. As we do not have comparisons with analytical or numerical solutions for these cases, we have included these results in the appendix. This approach allows us to focus on the methodology and visualization of the PINN solutions in the main text while providing the detailed solution results in the appendix for reference.

Besides, the related code using random points for diffusion equation and 2D Laplace equation is available at the appendix.

A Wave Equation

The 2D wave equation with Dirichlet boundary conditions is given by:

$$
\begin{aligned}
&u_{tt} = \nabla^2 u, \quad \text{for} \quad (x, y) \in \Omega, \quad t \in [0, T] \\
&u(t=0, x, y) = \sin(\pi x)\sin(\pi y)e^{-t}, \\
&u(t, x, y) = 0, \quad \text{on} \quad \partial\Omega, \quad \forall t \in [0, T],
\end{aligned}
$$

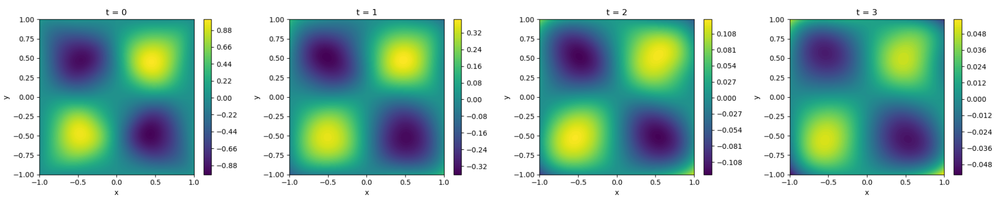


Figure 13: Visualization of the solution $u(t, x, y)$.

Upon convergence, the trained neural network provides an approximation to the solution of the 2D wave equation that honors the specified initial and boundary conditions. The learned solution can be visualized at different times to understand the wave propagation within the domain Ω. This approach leverages the power of deep learning to solve complex partial differential equations without requiring a mesh or grid, offering flexibility and efficiency in handling PDEs with complex geometries and conditions.

B 3D Laplace Equation

We try to solve:

$$
\Delta_3 u = u_{xx} + u_{yy} + u_{zz} = 0 \quad \text{in} \quad D
$$

$$
D = \{0 < x < \pi, 0 < y < \pi, 0 < z < \pi\}
$$

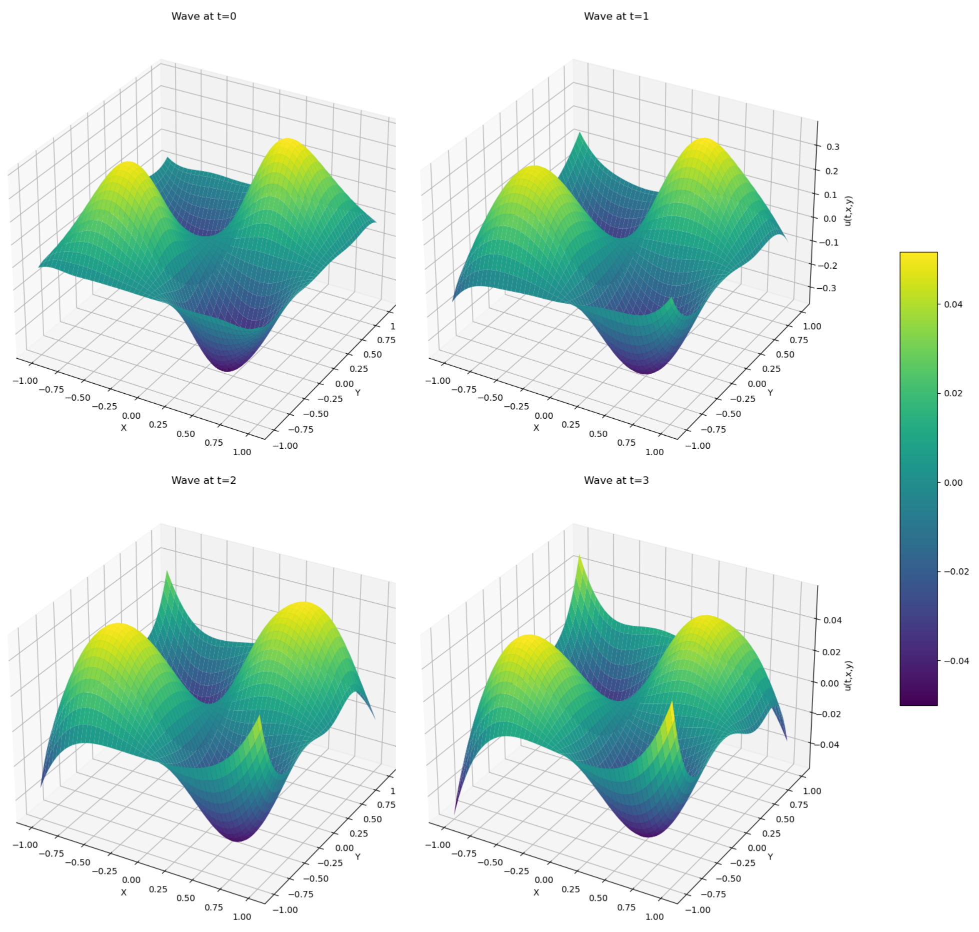
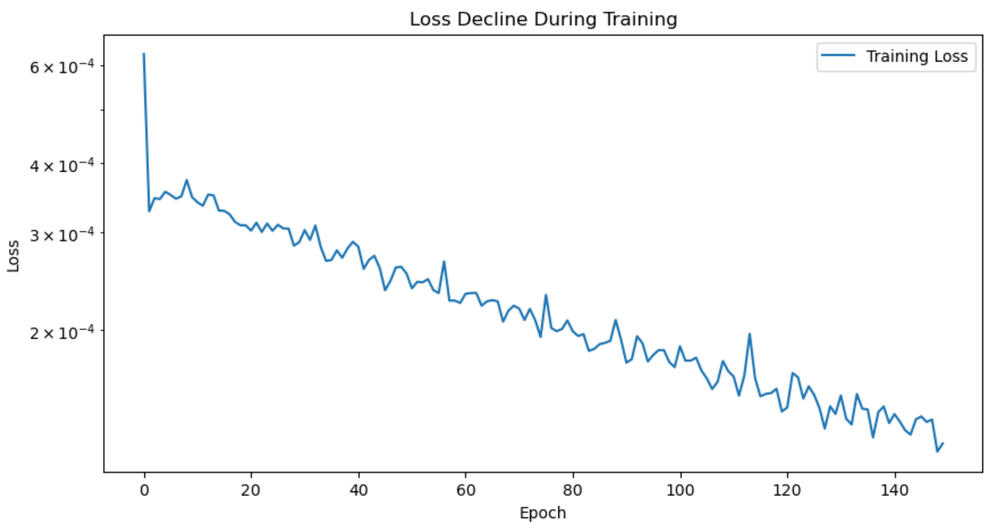


Figure 14: Visualization of the solution $u(t, x, y)$.



with the boundary condition:

$$u(\pi, y, z) = \sin y \cos z$$

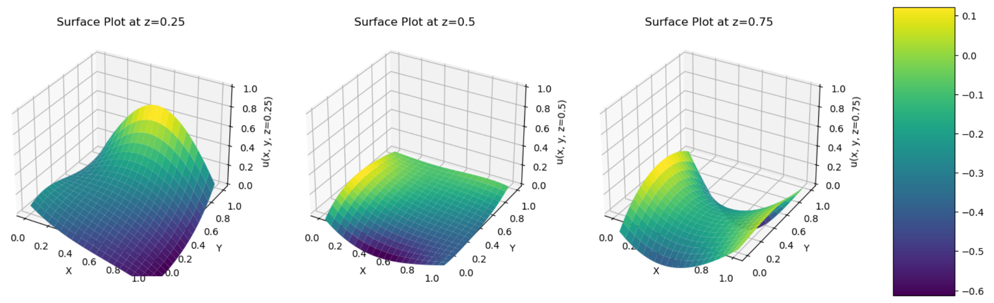


Figure 15: Visualization of the solution $u(x, y, z)$.

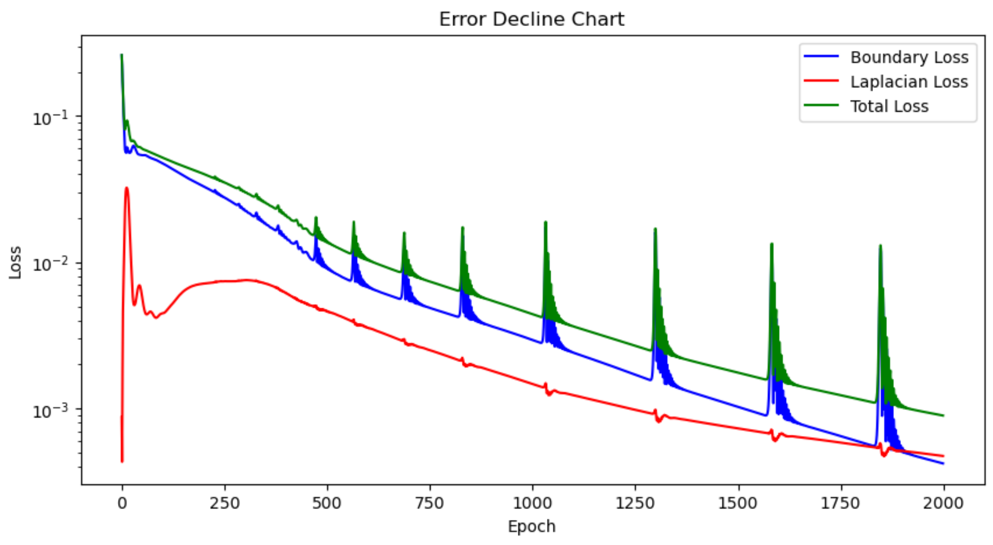
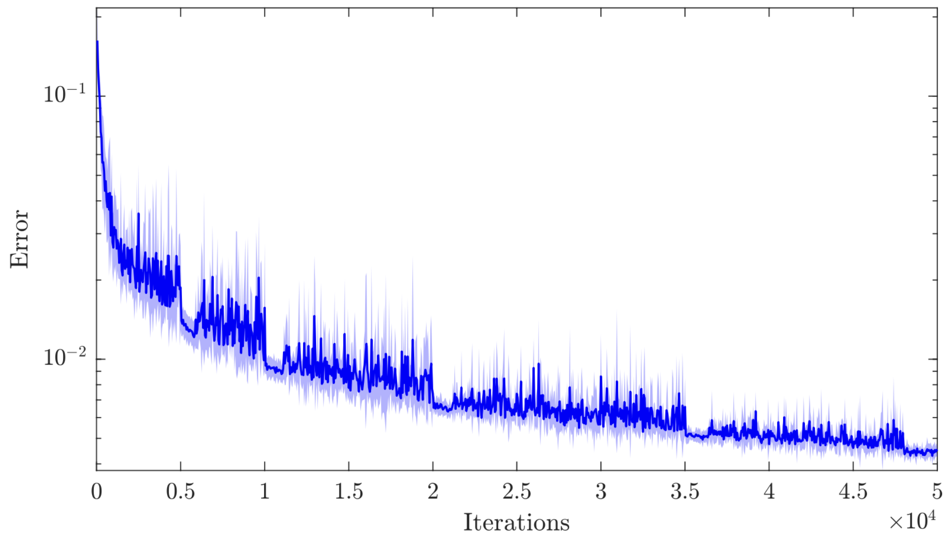


Figure 16: Error of the predicted solution $u(x, y, z)$.

C 10D Laplace Equation

$$
\begin{aligned}
-\Delta u = 0, \qquad & x \in (0,1)^{10} \\
u(x) = \textstyle\sum_{k=1}^{5} x_{2k-1}x_{2k}, \quad & x \in \partial(0,1)^{10}.
\end{aligned}
$$

use deep ritz method to solve the 10D Laplace equation.



D Code for Diffusion Equation

```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from torch.autograd import grad

# Check if CUDA is available and set the device to GPU or CPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

class PINN(nn.Module):
    def __init__(self):
        super(PINN, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(2, 128),  # Increase the number of neurons
            nn.Tanh(),
            nn.Linear(128, 128),
            nn.Tanh(),
            nn.Linear(128, 1)
        )

    def forward(self, x, t):
        u = self.net(torch.cat([x, t], dim=1))  # Concatenate x and t
        return u
```

```
def pde_residual(net, x, t):
    u = net(x, t)
    u_x = grad(u.sum(), x, create_graph=True)[0]
    u_xx = grad(u_x.sum(), x, create_graph=True)[0]
    u_t = grad(u.sum(), t, create_graph=True)[0]
    residual = u_t - u_xx
    return residual

def boundary_condition(net, x, t):
    return net(x, t)

def initial_condition(net, x, t):
    # Ensure that t is zero tensor of the same shape as x for initial condition
    t = torch.zeros_like(x)
    u = net(x, t)
    return u - torch.sin(np.pi * x)



# Domain points
# Adjusting x and t tensor definitions to have requires_grad=True
# Use more points or advanced sampling techniques
x = torch.tensor(np.random.uniform(0, 1, size=(1000, 1)), dtype=torch.float32,
    requires_grad=True)
t = torch.tensor(np.random.uniform(0, 1, size=(1000, 1)), dtype=torch.float32,
    requires_grad=True)

# Define the domain points with required gradients
x_domain = torch.linspace(0, 1, 100).view(-1, 1).requires_grad_(True)
t_domain = torch.linspace(0, 1, 100).view(-1, 1).requires_grad_(True)

# Apply initial conditions (t=0) across the x domain
initial_condition_points = torch.cat((x_domain, torch.zeros_like(x_domain)), dim
    =1)

# Apply boundary conditions (x=0 and x=1) across the time domain
boundary_condition_points_0 = torch.cat((torch.zeros_like(t_domain), t_domain),
    dim=1)
boundary_condition_points_1 = torch.cat((torch.ones_like(t_domain), t_domain), dim
    =1)

# Boundary and initial condition points
# Correct approach to prepare x_bc and t_bc for boundary conditions
# Assuming t_bc is defined correctly as:
t_bc = torch.linspace(0, 1, 100).view(-1, 1)[1:].requires_grad_(True)  # Skip t=0
    to avoid duplication with IC
x_bc_0 = torch.zeros_like(t_bc)  # Boundary at x=0 for t>0
x_bc_1 = torch.ones_like(t_bc)   # Boundary at x=1 for t>0
# Combine these to form the complete x_bc tensor for both boundaries
x_bc = torch.cat((x_bc_0, x_bc_1), dim=0)

# Similarly, prepare t_bc to be used with both sets of x_bc
t_bc_repeated = torch.cat((t_bc, t_bc), dim=0)  # Repeat t_bc for both x=0 and x=1
     boundaries

model = PINN()
model.to(device)  # Move the model to the appropriate device

# Include weight decay in the optimizer (if not already present)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-5)
# Use a learning rate scheduler
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience
    =500, factor=0.5, min_lr=1e-6)



# Lists to keep track of errors
total_errors = []
boundary_errors = []

for epoch in range(5000):  # Number of epochs
    optimizer.zero_grad()

    # Random sample domain points for PDE loss
    random_index = torch.randperm(x.size(0))
    random_x = torch.rand(500, 1, device=device).requires_grad_(True)
    random_t = torch.rand(500, 1, device=device).requires_grad_(True)
    pde_loss = pde_residual(model, random_x, random_t).pow(2).mean()
    # Initial condition loss
    ic_loss = initial_condition(model, initial_condition_points[:, 0:1],
        initial_condition_points[:, 1:2]).pow(2).mean()

    # Boundary condition loss
    bc_loss_0 = boundary_condition(model, boundary_condition_points_0[:, 0:1],
        boundary_condition_points_0[:, 1:2]).pow(2).mean()
    bc_loss_1 = boundary_condition(model, boundary_condition_points_1[:, 0:1],
        boundary_condition_points_1[:, 1:2]).pow(2).mean()
    boundary_loss = bc_loss_0 + bc_loss_1

    # Total loss
    loss = pde_loss + ic_loss + boundary_loss

    # Log the errors
    total_errors.append(loss.item())
    boundary_errors.append(boundary_loss.item())

    # Perform the backward pass and optimization step
    loss.backward()
    torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
    optimizer.step()
    scheduler.step(loss)

    if epoch % 100 == 0:
        print(f'Epoch␣{epoch},␣Loss:␣{loss.item()}')



# Define parameters for the analytical solution
x_np = np.linspace(0, 1, 100)
t_np = np.linspace(0, 1, 100)
X, T = np.meshgrid(x_np, t_np)
U_analytical = np.exp(-np.pi**2 * T) * np.sin(np.pi * X)

# Prepare for PINN solution
x_test = torch.linspace(0, 1, 100, device=device)
t_test = torch.linspace(0, 1, 100, device=device)
t_grid, x_grid = torch.meshgrid(t_test, x_test, indexing='ij')  # Correct indexing
tx_grid = torch.stack([x_grid.flatten(), t_grid.flatten()], dim=-1)

model.eval()  # Set the model to evaluation mode
with torch.no_grad():
    # Ensure tx_grid is on the same device as the model
    u_pred = model(tx_grid[:, 0:1], tx_grid[:, 1:2]).reshape(100, 100).cpu().numpy
        ()

# Plot the analytical solution and the PINN solution
fig = plt.figure(figsize=(20, 7))

# Analytical solution plot
ax1 = fig.add_subplot(121, projection='3d')
```

```
surf1 = ax1.plot_surface(X, T, U_analytical, cmap='viridis', edgecolor='none')
ax1.set_title('Analytical␣Solution')
ax1.set_xlabel('Space␣(x)')
ax1.set_ylabel('Time␣(t)')
ax1.set_zlabel('Temperature␣(u)')
fig.colorbar(surf1, ax=ax1, shrink=0.5, aspect=5)

# PINN solution plot
ax2 = fig.add_subplot(122, projection='3d')
surf2 = ax2.plot_surface(X, T, u_pred, cmap='viridis', edgecolor='none')
ax2.set_title('PINN␣Solution')
ax2.set_xlabel('Space␣(x)')
ax2.set_ylabel('Time␣(t)')
ax2.set_zlabel('Temperature␣(u)')
fig.colorbar(surf2, ax=ax2, shrink=0.5, aspect=5)

# Plotting the error decline chart
plt.figure(figsize=(12, 6))

# Calculate a moving average of the error for smoothing
def moving_average(a, n=100):
    ret = np.cumsum(a, dtype=float)
    ret[n:] = ret[n:] - ret[:-n]
    return ret[n - 1:] / n

smooth_total_errors = moving_average(total_errors, n=500)  # Adjust the window
    size as needed
smooth_boundary_errors = moving_average(boundary_errors, n=500)

plt.plot(smooth_total_errors, label='Total␣Error␣(smoothed)', color='red')
plt.plot(smooth_boundary_errors, label='Boundary␣Error␣(smoothed)', color='blue')
# ... rest of plotting code ...


plt.yscale('log')  # Set the y-axis to a logarithmic scale
plt.xlabel('Iteration')
plt.ylabel('Log␣Error')
plt.title('Error␣Decline␣Chart')
plt.legend()
plt.grid(True)

# Save the figure if needed
plt.savefig('/Users/zhouyixiaoxiao/Desktop/error_decline_chart.png')
```

```
# Sample data for the demonstration
# Let's assume we already have u_pred and U_analytical which are the solutions
   from PINN and the analytical method
# They are both 2D arrays with shape (100, 100), which you would replace with your
    actual data
# Calculate the absolute difference
u_difference = np.abs(u_pred - U_analytical)

# Define parameters for plotting
x = np.linspace(0, 1, 100)
t = np.linspace(0, 1, 100)
X, T = np.meshgrid(x, t)

# Plot the absolute difference as a 3D surface
fig = plt.figure(figsize=(14, 10))
ax = fig.add_subplot(111, projection='3d')

# Surface plot
surf = ax.plot_surface(X, T, u_difference, cmap='viridis', edgecolor='none')

# Labels and title
ax.set_title('Absolute␣Difference␣between␣PINN␣and␣Analytical␣Solutions')
ax.set_xlabel('Space␣(x)')
ax.set_ylabel('Time␣(t)')
ax.set_zlabel('Absolute␣Difference')

# Colorbar
fig.colorbar(surf, shrink=0.5, aspect=5)

plt.show()
```

E Code for 2D Laplace Equation

```
  import torch
  import numpy as np
  import matplotlib.pyplot as plt
  from mpl_toolkits.mplot3d import Axes3D
  # CUDA support
  if torch.cuda.is_available():
      device = torch.device('cuda')
  else:
      device = torch.device('cpu')
```

```
# Adjust the domain for generating training data
pi = np.pi
num_train_samples = 10000

# Assuming a square root is a suitable way to allocate boundary points for a
    square domain
boundary_points_per_edge = int(np.sqrt(num_train_samples))

# Generate random points for the interior
interior_points = num_train_samples - (4 * boundary_points_per_edge)
interior_x = torch.rand(interior_points) * pi
interior_y = torch.rand(interior_points) * pi
interior_xy = torch.stack((interior_x, interior_y), dim=1)

# Generate uniform points for the boundaries at y=0 and y=pi
boundary_y0 = torch.zeros(boundary_points_per_edge).view(-1, 1)
boundary_y1 = pi * torch.ones(boundary_points_per_edge).view(-1, 1)
boundary_x = torch.linspace(0, pi, boundary_points_per_edge).view(-1, 1)

# Combine x and y for the boundary at y=0 and y=pi
boundary_xy_y0 = torch.cat((boundary_x, boundary_y0), dim=1)
boundary_xy_y1 = torch.cat((boundary_x, boundary_y1), dim=1)

# Generate uniform points for the boundaries at x=0 and x=pi
boundary_x0 = torch.zeros(boundary_points_per_edge).view(-1, 1)
boundary_x1 = pi * torch.ones(boundary_points_per_edge).view(-1, 1)
boundary_y = torch.linspace(0, pi, boundary_points_per_edge).view(-1, 1)

# Combine x and y for the boundary at x=0 and x=pi
boundary_xy_x0 = torch.cat((boundary_x0, boundary_y), dim=1)
boundary_xy_x1 = torch.cat((boundary_x1, boundary_y), dim=1)

# Stack interior and boundary points
xy_grid = torch.cat((interior_xy, boundary_xy_y0, boundary_xy_y1, boundary_xy_x0
    , boundary_xy_x1), dim=0)


# Define the neural network architecture
class LaplaceNet(torch.nn.Module):
    def __init__(self):
        super(LaplaceNet, self).__init__()
        self.fc1 = torch.nn.Linear(2, 128)
        self.fc2 = torch.nn.Linear(128, 128)
        self.fc3 = torch.nn.Linear(128, 128)
        self.fc4 = torch.nn.Linear(128, 1)

    def forward(self, x):
        x = torch.tanh(self.fc1(x))
        x = torch.tanh(self.fc2(x))
        x = torch.tanh(self.fc3(x))
        return self.fc4(x)

# Define the network and optimizer
network = LaplaceNet()
optimizer = torch.optim.Adam(network.parameters(), lr=0.001)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=100, gamma=0.9)

# Function to compute Laplacian (for clarity)
def laplacian(u, xy):
    u_sum = u.sum()
    grad_u = torch.autograd.grad(u_sum, xy, create_graph=True, retain_graph=True
        )[0]
    u_x, u_y = grad_u[:, 0], grad_u[:, 1]
    u_xx = torch.autograd.grad(u_x.sum(), xy, create_graph=True)[0][:, 0]
    u_yy = torch.autograd.grad(u_y.sum(), xy, create_graph=True)[0][:, 1]
    return u_xx + u_yy

# Custom loss for Neumann BC (u_y = 0 at y=0 and y=pi)
def neumann_bc_loss(network, xy):
    points = xy[xy[:, 1] == 0]  # Apply for y=0 and y=pi, this line for y=0
    points.requires_grad_(True)
    u = network(points)
    u_y = torch.autograd.grad(u, points, torch.ones_like(u), create_graph=True)
        [0][:, 1]
    loss = (u_y ** 2).mean()

    points = xy[xy[:, 1] == pi]  # Now for y=pi
    points.requires_grad_(True)
    u = network(points)
    u_y = torch.autograd.grad(u, points, torch.ones_like(u), create_graph=True)
        [0][:, 1]
    loss += (u_y ** 2).mean()

    return loss

# List to store loss values
loss_history = []
```

```python
# Adjusting the training loop
# Assuming you have defined your network, optimizer, and other setup as before
for epoch in range(1500):
    network.zero_grad()
    xy_grid.requires_grad_(True)
    u_pred = network(xy_grid)

    # Loss for Dirichlet boundary conditions
    loss_dirichlet = torch.tensor(0., device=u_pred.device)
    # u = 0 for x = 0
    mask_x0 = xy_grid[:, 0] == 0
    loss_dirichlet += (u_pred[mask_x0] ** 2).mean()
    # u = 0.5 * (1 + cos(2y)) for x = pi
    mask_xpi = xy_grid[:, 0] == pi
    target = 0.5 * (1 + torch.cos(2 * xy_grid[mask_xpi][:, 1]))
    loss_dirichlet += ((u_pred[mask_xpi].squeeze() - target) ** 2).mean()

    # Laplacian loss (PDE constraint)
    laplacian_u = laplacian(u_pred, xy_grid)
    loss_pde = torch.mean(laplacian_u ** 2)

    # Neumann BC loss
    loss_neumann = neumann_bc_loss(network, xy_grid)

    # Total loss
    loss = loss_dirichlet + loss_pde + loss_neumann

    loss.backward()
    optimizer.step()
    loss_history.append(loss.item())
    if epoch % 100 == 0:
        print(f'Epoch␣{epoch},␣Loss:␣{loss.item()}')

# Define the function u(x, y)
def u_analytical(x, y):
    return (1 / (2 * np.pi)) * x + (1 / (2 * (np.exp(2 * np.pi) - np.exp(-2 * np
        .pi)))) * np.cos(2 * y) * (np.exp(2 * x) - np.exp(-2 * x))

# Visualization for PINN solution
network.eval()
with torch.no_grad():
    # Predict the solution with the neural network
    u_pred = network(xy_grid).detach().numpy()
```

```python
# The points used for training are already in 'xy_grid', so we use them directly
x_points = xy_grid[:, 0].detach().numpy()
y_points = xy_grid[:, 1].detach().numpy()

# Create a meshgrid for the analytical solution
X_analytical, Y_analytical = np.meshgrid(np.linspace(0, np.pi, 100), np.linspace
    (0, np.pi, 100))
Z_analytical = u_analytical(X_analytical, Y_analytical)

# Initialize figure
fig = plt.figure(figsize=(15, 7))

# Plot PINN solution using a scatter plot
ax1 = fig.add_subplot(1, 2, 1, projection='3d')
scatter1 = ax1.scatter(x_points, y_points, u_pred, c=u_pred, cmap='viridis',
    marker='.')
ax1.set_xlabel('X')
ax1.set_ylabel('Y')
ax1.set_zlabel('u(x,␣y)')
ax1.set_title('PINN␣Solution␣of␣the␣Laplace␣Equation')

# Plot analytical solution
ax2 = fig.add_subplot(1, 2, 2, projection='3d')
surf2 = ax2.plot_surface(X_analytical, Y_analytical, Z_analytical, cmap='viridis
    ', edgecolor='none')
ax2.set_xlabel('X')
ax2.set_ylabel('Y')
ax2.set_zlabel('u(x,␣y)')
ax2.set_title('Analytical␣Solution␣of␣the␣Laplace␣Equation')

# Show the figure
plt.tight_layout()
plt.show()

# Plotting the training loss history
plt.figure(figsize=(10, 5))
plt.plot(loss_history, label='Training␣Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training␣Loss␣Over␣Time')
plt.legend()
plt.show()

# Assuming 'u_analytical' function is defined to work with flattened arrays
```

```
u_analytical_pred = u_analytical(x_points, y_points)

# Flatten the arrays if they're not already
u_pred_flat = u_pred.flatten()
u_analytical_pred_flat = u_analytical_pred.flatten()

# Compute the error
error = np.abs(u_pred_flat - u_analytical_pred_flat)
# Make sure the lengths match before plotting
assert len(x_points) == len(y_points) == len(error), "The␣point␣arrays␣and␣error
    ␣must␣have␣the␣same␣length."

# Now plotting the error
fig = plt.figure(figsize=(15, 7))
ax = fig.add_subplot(1, 1, 1, projection='3d')
scatter_error = ax.scatter(x_points, y_points, error, c=error, cmap='viridis',
    marker='.')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Absolute␣Error')
ax.set_title('Absolute␣Error␣between␣PINN␣and␣Analytical␣Solutions')
fig.colorbar(scatter_error, shrink=0.5, aspect=5)
plt.tight_layout()
plt.show()
```