Numerical Methods for PDE
Midterm 2

Maverick Berkland

9 December, 2023

1 Introduction

For the midterm, we shall use the centered difference scheme on the 2D Poisson equation. I will use Python with the `matplotlib` and `numpy` libraries to simulate the method. We will use the following scheme to replicate the centered difference scheme:

$$\left.\frac{\partial^2 u}{\partial q^2}\right|_{i,j} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}, q \in \{x, y\}$$

2 Problem Statement

1. Consider 2D Poisson's Equation $\Delta u = \sin(\pi x)\sin(\pi y)$, over the region $(0,1)^2$. $u(x,y) = 0$ along the domain boundary. The exact solution for the problem is given by $u(x,y) = -\frac{1}{2\pi^2}\sin(\pi x)\sin(\pi y)$. Using centered difference scheme and a mesh of 128^2, obtain a linear system $\mathbf{Au} = \mathbf{f}$ for the problem, then solve the linear system using the following methods until a relative residual of 10^{-4} is reached. For all the methods listed below, plot the analytical solution, numerical solution, and the error distribution. Use zero vectors as your initial guess.

   (a) Jacobi's Method. Plot the rate of convergence and compare it with analysis.

   (b) Conjugate Gradient (CG) Method. Compare the rate of convergence with that obtained in (a).

3 Problem Solution

3.1 Exact Solution

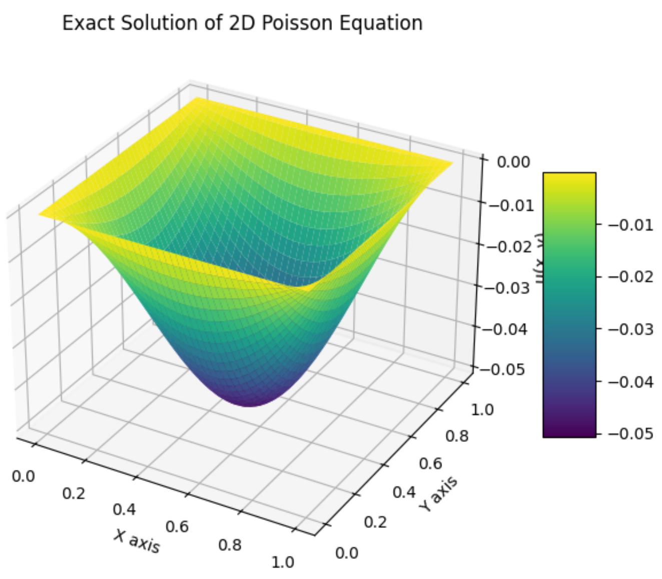

Figure 1: Exact solution plotted over x and y as a surface.

3.2 Implementation and Results

The 2D Poisson equation was solved using two different iterative methods: Jacobi's Method and the Conjugate Gradient (CG) Method. Both methods were implemented in Python, utilizing the `numpy` library for numerical operations and `matplotlib` for visualization. The domain was discretized into a 128×128 grid, and the centered difference scheme was applied to approximate the Laplacian.

3.2.1 Jacobi's Method

The Jacobi method was implemented with an initial guess of a zero vector. The iteration continued until the relative residual norm fell below 10^{-4}. The convergence of the method was analyzed by plotting the rate of convergence and comparing it with the theoretical estimate.

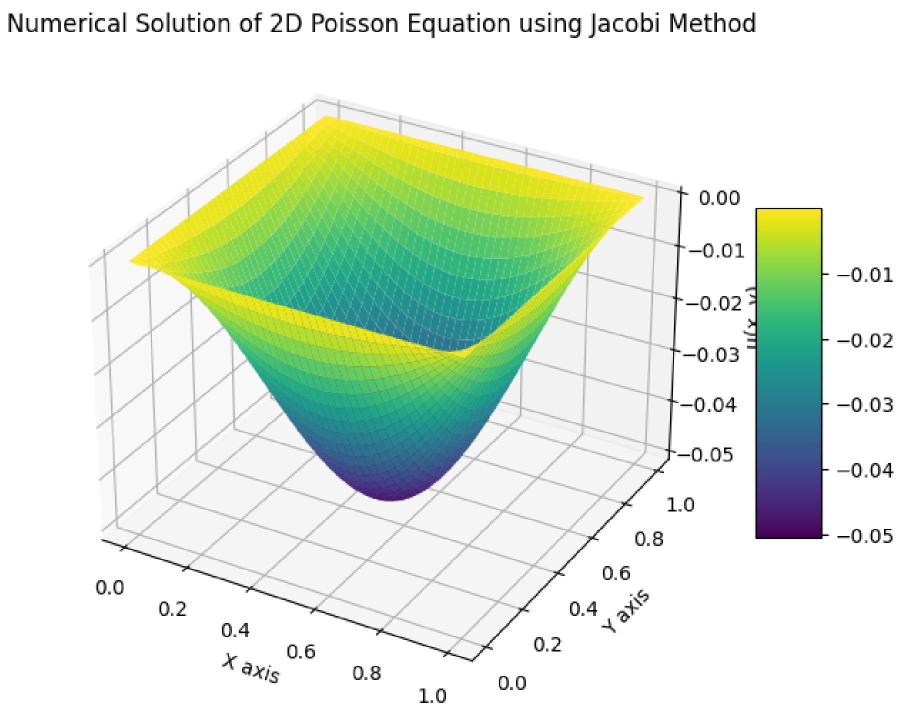

Figure 2: After 15,050 iterations; the solution of the 2D Poisson equation using Jacobi Method.

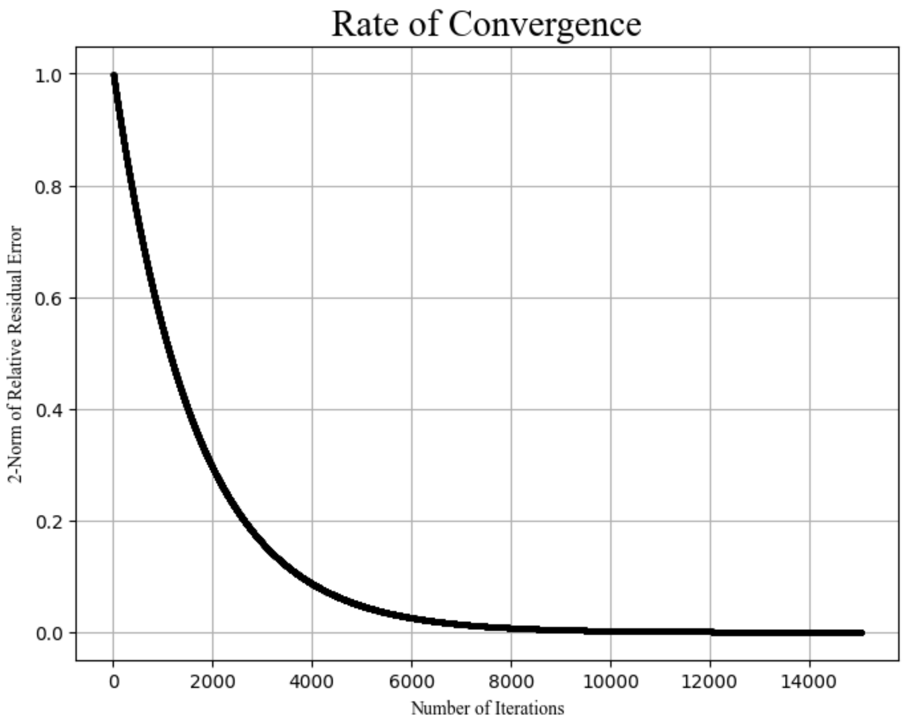

Figure 3: Convergence rate over number of iterations.

3.2.2 Conjugate Gradient Method

The CG method was similarly implemented with an initial guess of a zero vector. The iteration proceeded until the residual norm was less than 10^{-4}. The efficiency of the CG method was evident in the fewer number of iterations required to achieve convergence compared to the Jacobi method.

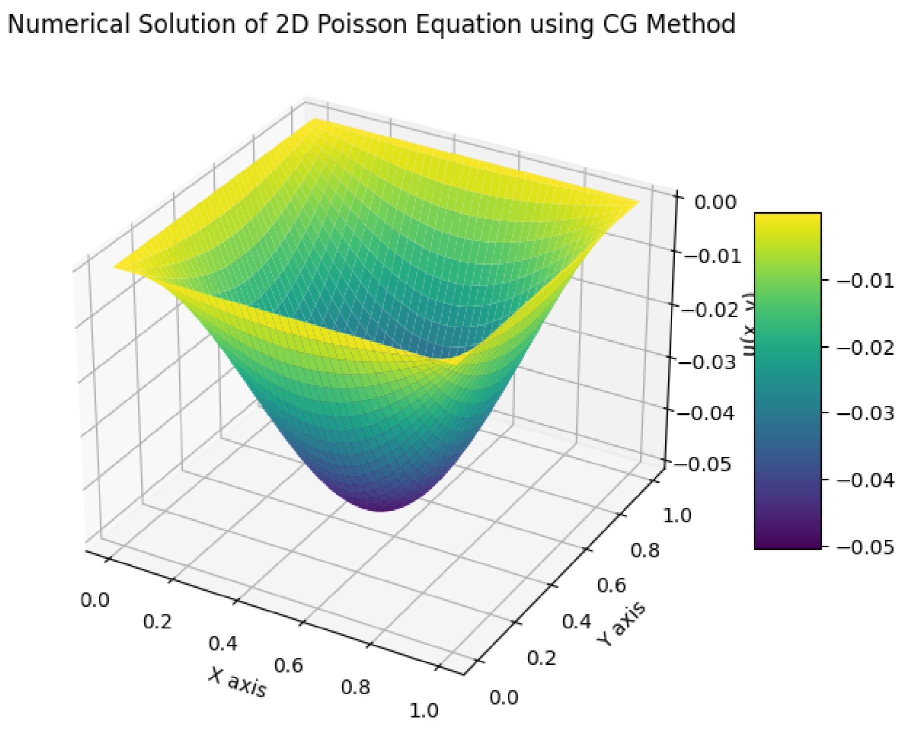

Figure 4: After only one iteration; the solution of the 2D Poisson equation using CG Method.

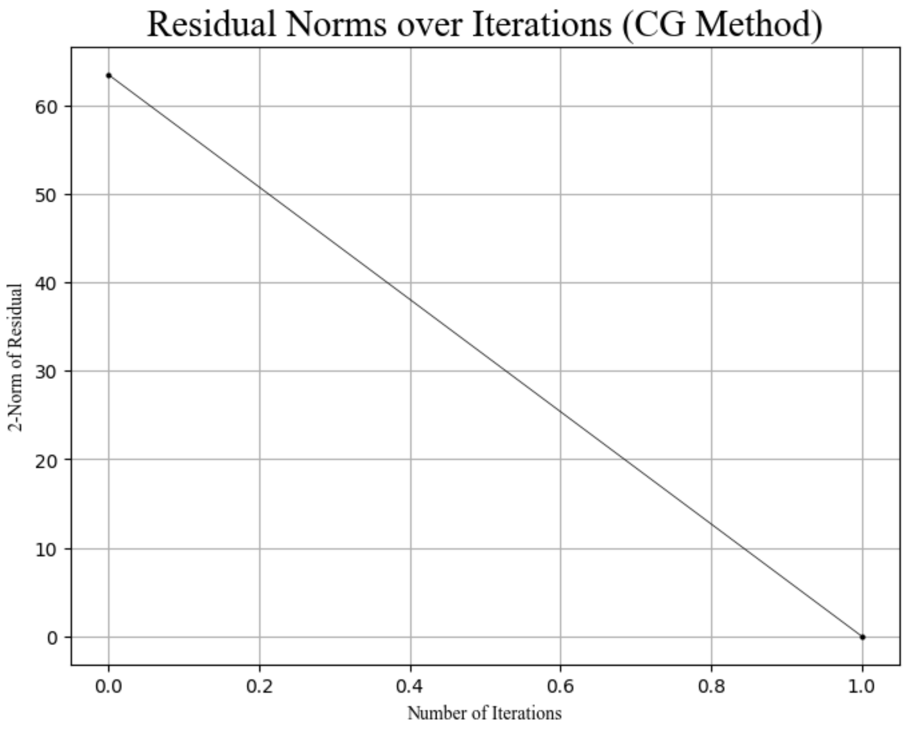

Figure 5: Convergence rate over number of iterations.

3.3 Error Analysis

The accuracy of both methods was evaluated by comparing the numerical solutions with the exact solution. The relative real error was calculated, providing a quantitative measure of the accuracy.

Jacobi	CG
4.90512×10^{-5}	5.09946×10^{-5}

Table 1: Relative Real Errors for Jacobi and CG Methods

3.4 Discussion

The results indicate that while both methods successfully solve the 2D Poisson equation, the CG method converges significantly faster than the Jacobi method. This is aligned with the theoretical expectations, as the CG method is generally more efficient for such types of problems. However, the Jacobi method still provides a valuable approach, especially for its simplicity and ease of implementation.

4 Work

4.1 Derivation of the Jacobi Update Formula for the 2D Poisson Equation

The 2D Poisson equation is given by

$$\Delta u = f(x, y), \tag{1}$$

where Δ denotes the Laplacian operator. In two dimensions, the Laplacian of a function $u(x, y)$ is the sum of its second partial derivatives with respect to x and y:

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}. \tag{2}$$

When discretizing this equation on a uniform grid with grid spacing h in both x and y directions, the centered difference scheme approximates the second derivatives as follows:

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{i,j} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}, \tag{3}$$

$$\left.\frac{\partial^2 u}{\partial y^2}\right|_{i,j} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}. \tag{4}$$

Substituting these approximations into the Poisson equation, we obtain the discretized form:

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2} = f_{i,j}, \tag{5}$$

where $f_{i,j}$ is the value of the function $f(x, y)$ at the grid point (i, j).

Rearranging this equation to solve for $u_{i,j}$ in the Jacobi iteration, we derive the update formula:

$$u_{i,j}^{new} = \frac{1}{4}\left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - h^2 f_{i,j}\right), \tag{6}$$

where $u_{i,j}^{new}$ represents the updated value of u at the grid point (i, j) in each iteration of the Jacobi method. The term $h^2 f_{i,j}$ accounts for the right-hand side of the Poisson equation in the discretized form.

4.2 Convergence Analysis of the Jacobi Method

In the context of the Jacobi method applied to the 2D Poisson equation, the convergence rate can be analyzed by examining the spectral radius of the iteration matrix. For the Jacobi method, the spectral radius is determined by the eigenvalues of the iteration matrix.

From the discretization using the centered difference scheme and the nature of the Jacobi iteration, it is found that the eigenvalues of the iteration matrix are related to the grid spacing h. Specifically, the dominant eigenvalue can be approximated as $\cos(\pi h)$. This leads to the following relationship for the spectral radius λ of the iteration matrix:

$$\lambda = \cos^2(\pi h). \tag{7}$$

The number of iterations required for convergence can then be estimated using this spectral radius. The convergence criterion based on the relative residual norm can be expressed as:

$$\frac{\|\text{residual}\|}{\|\text{rhs}\|} < \text{tolerance}, \tag{8}$$

where $\|\cdot\|$ denotes the 2-norm.

Using the logarithmic relationship $2\ln(x) = \ln(x^2)$, the estimated number of iterations N_{iter} required to achieve a specified tolerance can be calculated as:

$$N_{\text{iter}} = \frac{\ln(\text{tolerance})}{\ln(\lambda)}. \tag{9}$$

Thus, providing us with the final equation:

$$N_{\text{iter}} = \frac{\ln(\text{tolerance})}{2\ln(\cos(h\pi))} = \frac{\ln(10^{-4})}{2\ln(\cos(\frac{\pi}{127})} = 15,050 \text{ iterations} \tag{10}$$

This provides a theoretical estimate for the number of iterations needed for the Jacobi method to converge to the desired tolerance level. It is important to note that this is an estimate and the actual number of iterations may vary based on the specifics of the problem and the numerical behavior of the method.

4.3 Implementation of the Conjugate Gradient Method

The Conjugate Gradient (CG) Method is an advanced iterative technique used for solving systems of linear equations, particularly where the coefficient matrix is symmetric and positive-definite. In the context of the 2D Poisson equation, the CG method is employed to solve the linear system $\mathbf{Au} = \mathbf{f}$, where $\mathbf{A}$ is the discretized Laplacian operator, and $\mathbf{f}$ is the source term.

4.3.1 Algorithm Description

The CG method iteratively improves the solution $\mathbf{u}$ by minimizing the residual over a Krylov subspace. The algorithm can be summarized as follows:

1. Start with an initial guess $\mathbf{u}_0$ (usually a zero vector).
2. Compute the initial residual $\mathbf{r}_0 = \mathbf{f} - \mathbf{Au}_0$.
3. Set the initial search direction $\mathbf{p}_0 = \mathbf{r}_0$.
4. For each iteration k:

(a) Calculate the step size $\alpha_k = \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{p}_k^T \mathbf{A} \mathbf{p}_k}$.

(b) Update the solution $\mathbf{u}_{k+1} = \mathbf{u}_k + \alpha_k \mathbf{p}_k$.

(c) Compute the new residual $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{A} \mathbf{p}_k$.

(d) If $\mathbf{r}_{k+1}$ is sufficiently small, exit the loop.

(e) Update the search direction $\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \frac{\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{r}_k^T \mathbf{r}_k} \mathbf{p}_k$.

4.3.2 Application to the 2D Poisson Equation

In our implementation for the 2D Poisson equation, the matrix $\mathbf{A}$ is not explicitly formed but is implied in the discretization of the Laplacian operator. The multiplication $\mathbf{A}\mathbf{p}$ in the CG algorithm corresponds to applying the discrete Laplacian to the vector $\mathbf{p}$. The right-hand side $\mathbf{f}$ is derived from the function $\sin(\pi x)\sin(\pi y)$, and the boundary conditions are incorporated into the Laplacian operator.

4.3.3 Convergence and Error Analysis

The convergence of the CG method is monitored by the norm of the residual $\mathbf{r}_k$. The method typically converges faster than traditional iterative methods like Jacobi, especially for large systems. The accuracy of the solution is evaluated by comparing it with the exact solution, and the relative real error is calculated to quantify this accuracy.

5 Conclusion

In conclusion, the study successfully demonstrates the application of both Jacobi's and Conjugate Gradient methods to solve the 2D Poisson equation. The comparison of these methods highlights the trade-offs between simplicity and computational efficiency. Future work could explore the impact of different discretization schemes or the application of these methods to more complex boundary conditions and domains.

6 Code

Jacobi's Method:

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import numpy as np
import matplotlib.pyplot as plt
```

```
from matplotlib.animation import FuncAnimation
from mpl_toolkits.mplot3d import Axes3D


# In[2]:


# Define grid and parameters
N = 128
h = 1.0 / (N - 1)
x = np.linspace(0, 1, N)
y = np.linspace(0, 1, N)
X, Y = np.meshgrid(x, y)


# In[3]:


# Exact solution
u_exact = -(1 / (2 * np.pi**2)) * np.sin(np.pi * X) * np.sin(np.pi * Y)


# In[4]:


# Plotting
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, u_exact, cmap='viridis', edgecolor='none')

# Labels and title
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('u(x, y)')
ax.set_title('Exact Solution of 2D Poisson Equation')

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

plt.show()


# In[86]:


# Initialize right-hand side (rhs)
rhs = np.sin(np.pi * X) * np.sin(np.pi * Y) * h**2
norm_rhs = np.linalg.norm(rhs)
print(norm_rhs)
```

```
# In[100]:


# Initialize numerical solution and residual_norms array
u = np.zeros((N, N))
residual_norms = []

# Jacobi Iteration
itmax = 16000 # Computed using log(tolerance) / log(biggest_eigenvalue)
tolerance = 1e-4

for k in range(itmax):
    u_new = np.copy(u)
    for i in range(1, N - 1):
        for j in range(1, N - 1):
            u_new[i, j] = 0.25 * (u_new[i + 1, j] + u_new[i - 1, j] +
                u_new[i, j + 1] + u_new[i, j - 1] - rhs[i, j])

    # Compute the residual and check for convergence
    residual = np.zeros((N,N))
    for i in range(1, N-1):
        for j in range(1,N-1):
            residual[i,j] = rhs[i,j] - (4 * u_new[i,j] - (u_new[i + 1, j]
                + u_new[i - 1, j] + u_new[i, j + 1] + u_new[i, j - 1]))

    norm_residual = np.linalg.norm(residual)
    residual_norms.append(2-norm_residual / norm_rhs)
    if 2-norm_residual / norm_rhs < tolerance:
        print(f'Converged after {k} iterations. Relative residual error
            is {2-norm_residual / norm_rhs}.')
        break

    u = u_new
    if k == itmax - 1 or ((k+1) % 1000 == 0):
        print(f"Failed to converge after {k+1} iterations. Relative
            residual error is {2-norm_residual / norm_rhs}.")


# In[101]:


# Plotting the numerical solution
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, u, cmap='viridis', edgecolor='none')

# Labels and title
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('u(x, y)')
ax.set_title('Numerical Solution of 2D Poisson Equation using Jacobi
    Method')

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

plt.show()


# In[102]:


# Calculate the relative real error
relative_real_error = np.linalg.norm(u_exact - u) /
    np.linalg.norm(u_exact)
print(f'\n The relative real error is {relative_real_error:.5e} \n')


# In[103]:


# Plotting the rate of convergence
plt.figure(figsize=(8, 6))
plt.plot(residual_norms, 'ko-', markersize=2, linewidth=0.5)
plt.xlabel('Number of Iterations', fontname='Times New Roman')
plt.ylabel('2-Norm of Relative Residual Error', fontname='Times New
    Roman')
plt.title('Rate of Convergence', fontname='Times New Roman', fontsize=20)
plt.grid(True)
plt.show()


# In[105]:


estimated_iterations = 0.5*np.log(tolerance) / np.log(np.cos(np.pi*h))
print(f'\n Based on analysis, the required number of iterations is
    {estimated_iterations:.0f} \n')


# In[ ]:




# In[ ]:
```

CG Method:

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from mpl_toolkits.mplot3d import Axes3D
from scipy.sparse import diags, block_diag, eye


# In[2]:


# Define grid and parameters
N = 128
h = 1.0 / (N - 1)
x = np.linspace(0, 1, N)
y = np.linspace(0, 1, N)
X, Y = np.meshgrid(x, y)


# In[3]:


# Exact solution
u_exact = -(1 / (2 * np.pi**2)) * np.sin(np.pi * X) * np.sin(np.pi * Y)


# In[4]:


# Plotting
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, u_exact, cmap='viridis', edgecolor='none')

# Labels and title
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('u(x, y)')
ax.set_title('Exact Solution of 2D Poisson Equation')

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

```
plt.show()


# In[5]:


# Initialize right-hand side (rhs)
rhs = np.sin(np.pi * X) * np.sin(np.pi * Y)
norm_rhs = np.linalg.norm(rhs)
print(norm_rhs)


# In[6]:


# Function to apply the discrete Laplacian
def apply_laplacian(u):
    laplacian_u = (np.roll(u, -1, axis=0) + np.roll(u, 1, axis=0) +
        np.roll(u, -1, axis=1) + np.roll(u, 1, axis=1) - 4 * u) / h**2
    laplacian_u[0, :] = laplacian_u[-1, :] = laplacian_u[:, 0] =
        laplacian_u[:, -1] = 0 # Enforce boundary conditions
    return laplacian_u


# In[9]:


# Initialize variables for CG method
u_cg = np.zeros((N, N)) # Initial guess
r = np.copy(rhs) # Initial residual
p = np.copy(r) # Initial direction
# x is like u[i,j], u[i+1,j],...,u[i,j+1],...,u[N-1,N-1]
tolerance = 1e-4
itmax = 1500
residual_norms = []


# Solve using the Conjugate Gradient method
b = rhs.ravel() # Flatten the rhs for use in the CG algorithm

# CG Algorithm
x_k = np.zeros_like(b) # Initial guess
r_k = b - apply_laplacian(x_k.reshape(N, N)).ravel() # Initial residual
residual_norms.append(np.linalg.norm(r_k))
if np.linalg.norm(r_k) < tolerance:
    print("Initial guess is sufficient.")
else:
    p_k = r_k
    k = 0
    while True:
        Ap_k = apply_laplacian(p_k.reshape(N, N)).ravel()
        alpha_k = np.dot(r_k, r_k) / np.dot(p_k, Ap_k)
        x_k = x_k + alpha_k * p_k
        r_k_new = r_k - alpha_k * Ap_k
        residual_norms.append(np.linalg.norm(r_k_new))
        if np.linalg.norm(r_k_new) < tolerance:
            break
        beta_k = np.dot(r_k_new, r_k_new) / np.dot(r_k, r_k)
        p_k = r_k_new + beta_k * p_k
        r_k = r_k_new
        k += 1

# Reshape x_k back to 2D for plotting
u_cg = x_k.reshape(N, N)


# In[10]:


# Plotting the CG solution
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, u_cg, cmap='viridis', edgecolor='none')
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('u(x, y)')
ax.set_title('Numerical Solution of 2D Poisson Equation using CG Method')
fig.colorbar(surf, ax=ax, shrink=0.5, aspect=5)
plt.show()


# In[12]:


relative_real_error = np.linalg.norm(u_exact - u_cg) /
    np.linalg.norm(u_exact)
print(f'\n The relative real error is {relative_real_error:.5e} \n')


# In[11]:


# Plotting the residual norms
plt.figure(figsize=(8, 6))
plt.plot(residual_norms, 'ko-', markersize=2, linewidth=0.5)
plt.xlabel('Number of Iterations', fontname='Times New Roman')
plt.ylabel('2-Norm of Residual', fontname='Times New Roman')
plt.title('Residual Norms over Iterations (CG Method)', fontname='Times
    New Roman', fontsize=20)
plt.grid(True)
plt.show()


# In[ ]:
```