Kazakh-British Technical University

Report

Made by Shahnoza Nurubloeva

Almaty, 2024

Assignment #9

1) Boundary and Initial Conditions

No time dependancy → No initial Consitions

Dirichlet Boundary Conditions

BC for OX: $T(0, y) = 100$ and $T(1, y) = 100$

BC for OY: $T(x, 0) = 100$ and $T(x, 1) = 100$

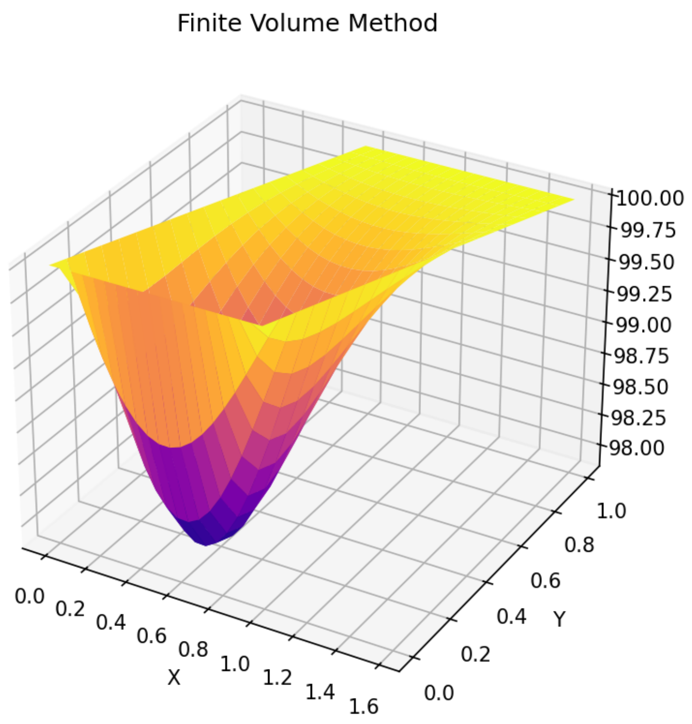
2) Computational Grid

Grid Type: nodes are nonevenly spaced in a rectangular shapes in XOY (which is linearly transformed by 30 degrees to a parallelogram)

Spacings (Δx, Δy) are chosen randomly

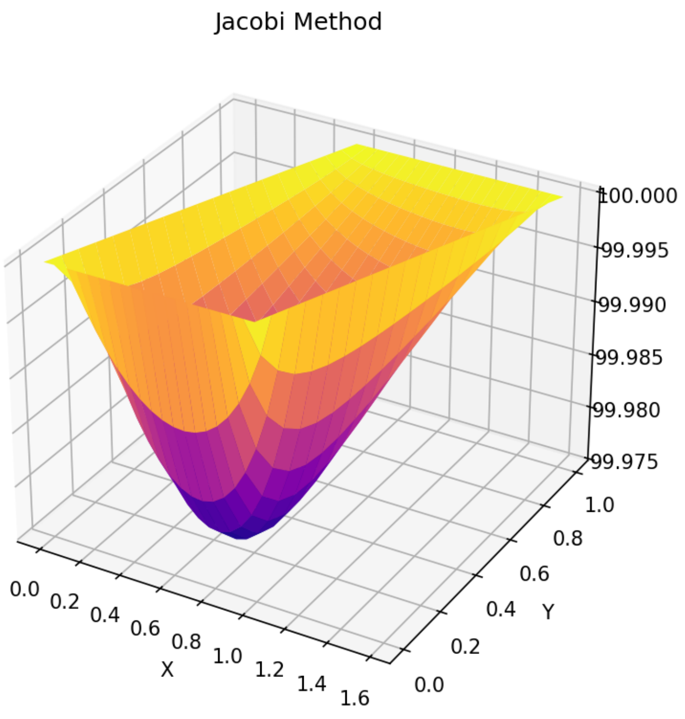
3) FVM (Finite Volume Method):

Similar to the Jacobi method, but specifically incorporates volume integrals to approximate flux and source terms, ensuring conservation laws are satisfied within each control volume.



4) Jacobi Method

$$T_{i,j}^{(k+1)} = \frac{S_{21}T_{i,j-1}^{(k)} + S_{31}T_{i-1,j}^{(k)} + S_{41}T_{i,j+1}^{(k)} + S_{51}T_{i+1,j}^{(k)} - f(x_{i,j}, y_{i,j}) \cdot \gamma}{S_{21} + S_{31} + S_{41} + S_{51}}$$



```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

def f(x, y):
    return np.sin(x)*np.cos(y)

def volume(neighbour1, neighbour2):
    return (neighbour1+neighbour1)*(neighbour2+neighbour2)/10

def Sij(neighbour1, neighbour2, neighbour3, neighbour4, neigh
    S_31 = 1/neighbour5*(neighbour1+neighbour3)/2
    s_21 = 1/(2*neighbour1)*(neighbour2+neighbour4)/2
    S_41 = 1/(2*neighbour3)*(neighbour2+neighbour4)/2
    S_51 = 1/neighbour2*(neighbour1+neighbour3)/2
    return s_21, S_31, S_41, S_51

def FVM(X, Y, N, T, max_step=1000):
    for it in range(max_step):
        for i in range(1, N[1] - 1):
            for j in range(1, N[0] - 1):
                neighbour1 = X[i,j]-X[i,j-1]
                neighbour2 = Y[i,j]-Y[i-1,j]
                neighbour3 = X[i,j+1]-X[i,j]
                neighbour4 = Y[i+1,j]-Y[i,j]
                neighbour5 = X[i+1,j]-X[i,j-1]

                gamma = volume(neighbour1, neighbour2)
                S21, S31, S41, S51 = Sij(neighbour1, neighbou
                                                    neighbour
                f_xy = f(X[i, j], Y[i, j])
                T[i,j]=(S21*T[i,j-1]+S31*T[i-1,j]+S41*T[i,j+1
                                                 f_xy*gamma)/
    return T

def Jacobi(X, Y, N, T, max_step=1000):
    for it in range(max_step):
        for i in range(1, N[1] - 1):
            for j in range(1, N[0] - 1):
                dx_2 = X[i,j]-X[i,j-1]
                dx_4 = X[i,j+1]-X[i,j]
                dy_5 = Y[i,j]-Y[i-1,j]
                dy_3 = Y[i+1,j]-Y[i,j]
                dx_3 = X[i+1,j]-X[i,j-1]

                S21, S31, S41, S51 = Sij(dx_2, dx_4, dy_5, dy
                gamma = volume(dx_2, dy_5)

                f_xy = f(X[i, j], Y[i, j])

                T[i, j] = (S21*T[i,j-1]+S31*T[i-1,j]+S41*T[i,
                                                    f_xy*gamma
    return T



N = [20, 20]
L = [1, 1]
initial_temp = 100
```

```
angle = 30
inrad = np.radians(angle)

x_discrete = np.linspace(0, L[0], N[0])
y_discrete = np.linspace(0, L[1], N[1])
X, Y = np.meshgrid(x_discrete, y_discrete)
X += Y * np.tan(inrad)

T = np.zeros((N[0], N[1]))
T[:, 0] = initial_temp
T[:, -1] = initial_temp
T[0, :] = initial_temp
T[-1, :] = initial_temp


fvm_matrix = FVM(X, Y, N, T)
# plot FVM
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, fvm_matrix, cmap='plasma', edgec
plt.colorbar(surf, label='Temperature')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_title('Finite Volume Method')
plt.show()


jacobi_matrix = Jacobi(X, Y, N, T)
# plot Jacobi
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X, Y, jacobi_matrix, cmap='plasma', ed
plt.colorbar(surf, label='Temperature')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_title('Jacobi Method')
plt.show()
```

Conclusion:

- **Jacobi Method:**
  - Simple and easy to implement for **linear** systems but might struggle with complex problems.
  - More useful for **steady-state solutions** and **linear equations**.
- **Finite Volume Method:**
  - More complex but suitable for **PDEs**, especially with **conservation laws**, and for systems with **non-linear behavior** or **complicated boundaries**.
  - Likely to be more accurate in your case since it's specifically designed to handle physical conservation principles.