UNIVERSITY OF CALGARY

PHYSICS 581

COMPUTATIONAL PHYSICS III

Lab #3 & Assignment #3

Authors
Kunal Naidu 30020999
Ruand Hennawi 30024661

Instructor
Dr. Rachid Ouyed

June 5, 2023

Contents

Introduction

In this Lab and Assignment, we will be exploring different aspects of differential equations, including a classification based on the discriminant of the quadratic form, as well as iterative solution methods such as the Relaxation, Jacobi, and Gauss-Seidel methods. Then we will be exploring how these methods are used to solve time-independent partial differential equations, PDEs, which include derivatives of an unknown function with respect to two or more independent variables. Moreover, we will be looking at boundary conditions such as Dirichlet, Neumann, and Mixed boundary conditions.

Methods

Differential Equation

A Partial Differential Equation (PDE) is an equation which includes derivatives of an unknown function with respect to two or more independent variables. On the other hand, an Ordinary Differential Equation (ODE) only has one independent variable.

Direct Method

Time-independent PDEs are classified as elliptic cases. In order to solve an elliptic equation, one needs to define boundary conditions. Elliptic equations are boundary value problems, and so one needs to specify the values of the solution at the boundary of the domain. Moreover, we will be dealing with elliptic PDEs in two space dimensions:

$$a(x,y)u_{xx} + b(x,y)u_{yy} + c(x,y)u = f(x,y) \qquad (1)$$

The boundary conditions can be of three different types: Dirichlet, Neumann, and mixed.

The Dirichlet boundary conditions specify the values of the solution at the boundary of the domain; u is the solution and given on the boundary.

The Neumann boundary conditions specify the normal derivative of the solution at the boundary of the domain; u_x or u_y given on the boundary (or u_n with n outward unit normal).

The mixed boundary conditions is a mix of Dirichlet and Neumann boundary conditions, also called Robin boundary conditions. It specifies both the value and the derivative of the solution at the boundary of the domain; $\alpha u + \beta u_n = \gamma$, where u_n is its normal derivative, α and β are given constants, and γ is a given function. The constant α is usually positive, while β can be positive, negative or zero.

An elliptic equation can be solved numerically or analytically. For an analytical solution, integrate the expression using calculus tricks and then Plug in boundary conditions to get values for integration constants. For a numerical solution, approximate derivatives using numerical tricks and then write all equations into a single matrix and solve.

The direct method solves Time-independent PDEs numerically using

$$u_{i-1,j} + u_{i+1,j} - 4u_{i,j} + u_{i,j-1} + u_{i,j+1} = 0 \qquad (2)$$

Iterative Solution Methods

Once we have defined the boundary conditions, the way we solve time-independent PDEs is based on an iterative scheme called the Relaxation Method.

First, start with an initial guess $u_{ij}^{(0)}$ where all values are known. Then, To obtain the new solution we solve for

$$u_{ij} = 0.25 \cdot \left[u_{i,j+1}^{(0)} + u_{i,j-1}^{(0)} + u_{i+1,j}^{(0)} + u_{i-1,j}^{(0)} \right] \qquad (3)$$

in order to obtain a new solution $u_{ij}^{(1)}$, which is used to obtain a new and improved $u_{ij}^{(2)}$. This process continues until a result which satisfies some specific convergence criteria is obtained.

The most used iterative methods are the Jacobi Iteration Method and the Gauss-Seidel Iteration Method.

Jacobi Iteration Method and the Gauss-Seidel Iteration Method

The Jacobi iteration method and the Gauss-Seidel iteration method are two commonly used iterative methods for solving time-independent PDEs.

The Jacobi method is based on solving for every variable once locally with respect to the other variables; computing the value of the PDE at each grid point using the values of the neighboring grid points from the previous iteration. The update is performed simultaneously for all grid points, and the process is repeated iteratively until the solution converges. The resulting method is easy to understand and implement, but convergence is slow.

The Gauss-Seidel method is like the Jacobi method, but instead of using the values from the previous iteration, it uses the updated values of the neighboring grid points as soon as they become available. In general, if the Jacobi method converges, the Gauss-Seidel method will converge faster than the Jacobi method, though still relatively slowly. We will be exploring this later in this report.

Partial Differential Equations and Classification

A useful classification is based on the discriminant $b^2 - ac$ of the quadratic form $F(x, y) \equiv ax^2 + 2bxy + cy^2$: $b^2 - ac < 0$ describes an **elliptic** equation, $b^2 - ac = 0$ describes a **parabolic** equation, and $b^2 - ac > 0$ describes a **hyperbolic** equation.

1 Poisson's Equation

Poisson's Equation is classified as an elliptic PDE and is given by

$$u_{xx} + u_{yy} = -f(x, y) \tag{4}$$

2 Laplace's Equation

Laplace's Equation is classified as an elliptic PDE and is given by

$$u_{xx} + u_{yy} = 0 \tag{5}$$

3 3-Dimensional Wave Equation

$$u_{tt} - x\, u_{xx} = 0 \tag{6}$$

4 Heat Equation in One Spatial Dimension

The Heat Equation is classified as a parabolic PDE and is given by

$$u_t - u_{xx} = 0 \tag{7}$$

5 Wave Equation in One Spatial Dimension

Wave Equation in one spatial dimension first order is given by

$$u_t = a u_x \tag{8}$$

where a is a constant, and the Wave Equation second order is given by

$$u_{tt} = a u_{xx} \tag{9}$$

and is classified as a hyperbolic PDE, and so $u_{tt} = 2u_{xx}$ is classified as a hyperbolic PDE.

6 Ginzburg-Landau Equation

The Ginzburg-Landau differential equation is given by

$$u_t = (1+ia)u_{xx} + (1+ic)u - (1+id)|u|^2 u \quad (10)$$

where a, c, and d are real constants that depend on the system being modeled, and i is the imaginary unit.

7 Korteweg-de Vries Equation

The Korteweg-de Vries differential equation is given by

$$u_t + u_{xxx} - 6uu_x = 0 \quad (11)$$

and the Korteweg-de Vries-Burger Equation is given by

$$u_t + 2uu_x - nuu_{xx} + muu_{xxx} = 0 \quad (12)$$

where n and m are constants.

First Order Forward Finite Difference

The First Order Forward Finite Difference approximation is given by

$$u_x(x) \approx \frac{u(x+h) - u(x)}{h} + O(h) \quad (13)$$

where $O(h)$ is an error of the first derivative using the forward formulation and is of order h. Here, we are going to take the function $u(x) = x^2$ and find the first order forward FD approximation to $u_x(3)$ using step size h = 0.1 and for h = 0.05. So,

$$\begin{aligned} u_x(3) &\approx \frac{u(3+h) - u(3)}{h} \\ &= \frac{(3+h)^2 - 3^2}{h} \\ &= \frac{9 + 6h + h^2 - 9}{h} \\ &= \frac{6h + h^2}{h} \\ &= 6 + h \end{aligned}$$

Therefore, $u_x(3) \approx 6.1$ with step size $h = 0.1$ and $u_x(3) \approx 6.05$ with step size $h = 0.05$. Comparing the results to the exact formula which is $u(x) = x^2 = 3^2 = 6$, for $h = 0.1$, the approximation is off by about 0.1, while for $h = 0.05$, the approximation is off by about 0.05. So we can see that when we decreased the step size, the approximation got closer to the exact value.

Central Finite Difference

The First Order Central Finite Difference approximation is given by

$$u_x(x) \approx \frac{u(x+h) - u(x-h)}{2h} + O(h^2) \quad (14)$$

where $O(h^2)$ is an error of the first derivative using the central formulation and is of order h^2. Again, we are going to take the function $u(x) = x^2$ and find the first order forward FD approximation to $u_x(3)$ using step size h = 0.1 and for h = 0.05. So,

$$\begin{aligned} u_x(3) &\approx \frac{u(3+h) - u(3-h)}{2h} \\ &= \frac{(3+h)^2 - (3-h)^2}{2h} \\ &= \frac{((3+h)(3+h)) - ((3-h)(3-h))}{2h} \\ &= \frac{3((3+x) - (3-x))}{h} \\ &= \frac{6h}{h} \end{aligned}$$

Therefore, $u_x(3) \approx 6$ with step size $h = 0.1$ and $u_x(3) \approx 6$ with step size $h = 0.05$. Comparing the results to the exact formula, we can see that for both step sizes, the central FD approximation is very accurate and produces values very close to the exact formula. This is expected since the central FD is second order accurate.

Second-order accuracy means that the error of a numerical approximation method decreases proportionally to the square of the step size used in the method. And so, from (13) we can see that the error is proportional to h^2 which means that the cen-

tral FD is second order accurate, while the first order forward FD approximation in (12) is only first order accurate.

Implicit FD Methods and Explicit FD Methods for PDEs

Explicit and implicit FD methods are two techniques used for solving PDEs. **Table 1** shows several FD schemes which include: (i) the explicit schemes: Forward Euler, Upwind, Leap-Frog, and Lax-Wendroff; (ii) the implicit schemes: Backward Euler, and Crank-Nicolson.

Explicit schemes are easy to implement and parallelize as they are low cost per time step. They are also a good starting point for the development of CFD software. However, small time steps are required for stability reasons, especially if the velocity and/or mesh size are varying strongly. Moreover, they are extremely inefficient for solution of stationary problems unless local time-stepping i.e. $\Delta t = \Delta t(x)$ is employed.

Implicit schemes are stable over a wide range of time steps, sometimes unconditionally and also constitute excellent iterative solvers for steady-state problems. However, they are difficult to implement and parallelize as they are high cost per time step. Moreover, they are also insufficiently accurate for truly transient problems at large Δt.

Explicit methods are conditionally stable, which means that the size of the time step must be chosen carefully to avoid numerical instability. Specifically, the time step size must be chosen such that it satisfies the CFL (Courant, Friedrichs, Lewy) condition; each mesh point the domain of dependence of the PDE must lie within the discrete domain of dependence. In contrast, implicit methods are unconditionally stable, which means that they can handle larger time steps without losing accuracy or becoming unstable, which means that the stability of the implicit scheme is not affected by the size of the time step.

Von Neumann Stability Analysis

To assess the stability of a scheme, we use Von Neumann stability analysis: Assume that at time t_l, temperature T varies harmonically in x,

$$T_j^l = e^{ikx_j} \tag{15}$$

and consider the wave length k to be a continuous number in the interval

$$0 \leq k \leq k_{Ny}$$

For each of these Fourier modes (characterized by the wave number k), analyze amplification factor A given by

$$T_j^{l+1} = AT_j^l \tag{16}$$

If $|A| < 1$, the Fourier mode is stable, otherwise it is unstable. If at least one Fourier mode is unstable, the scheme is also unstable.

Results and Discussion

Considering the following PDE for $u(x, y)$:

$$u_{xx} + u_{yy} = 0 \tag{17}$$

and applying a second order difference scheme for both x and y direction separately gave

$$u_{xx} = h_x^{-2}\left(u_{i-1,j} - 2u_{i,j} + u_{i+1,j}\right) \tag{18}$$

$$u_{yy} = h_y^{-2}\left(u_{i,j-1} - 2u_{i,j} + u_{i,j+1}\right) \tag{19}$$

where $u_{i,j}$ is the stencil. We then found the stencil for two cases: (i) $\Delta x = \Delta y$; (ii) $\Delta x \neq \Delta y$, considering that the grid spacing is same in both directions.

For $\Delta x = \Delta y$, meaning $\Delta x = \Delta y = h_x = h_y = h$, so we got

$$u_{xx} = h^{-2}\left(u_{i-1,j} - 2u_{i,j} + u_{i+1,j}\right) \tag{20}$$

$$u_{yy} = h^{-2}\left(u_{i,j-1} - 2u_{i,j} + u_{i,j+1}\right) \tag{21}$$

Plugging (20) and (21) into (17) we get

$$\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} = 0$$
$$u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j} = 0$$
$$\rightarrow u_{i,j} = \frac{1}{4}\left(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}\right)$$

For $\Delta x \neq \Delta y$, meaning $\Delta x = h_x$ and $\Delta y = h_y$. So plugging in (18) and (19) into (17) gave

$$\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h_x^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h_y^2} = 0$$
$$\rightarrow u_{i,j} = \frac{h_y^2\, u_{i-1,j} + h_y^2\, u_{i+1,j} + h_x^2\, u_{i,j-1} + h_x^2\, u_{i,j+1}}{2h_y^2 + 2h_x^2}$$

We wrote a python code that implements the $\Delta x = \Delta y$ stencil $u_{i,j}$ above in a box of size $0 \leq x \leq 20.0$ and $0 \leq y \leq 10.0$ with the following conditions $u(x, 0.0) = u(x, 10.0) = u(0.0, y) = 0$ and $u(20.0, y) = 100.0$. We used the Jacobi method to find the solution for a grid with spatial resolution $\Delta x = \Delta y = h_0 = 5.0$ and then obtained a plot to show the resulting solution as seen in Figure 1.

Figure 1 shows the solution $u(x, y)$ inside the box, where the boundary conditions are $u(x, 0.0) = u(x, 10.0) = u(0.0, y) = 0$ and $u(20.0, y) = 100.0$. The color map indicates the value of u at each point in the x and y axis, with higher values represented by cooler colours (yellows and greens) and lower values represented by warmer colours (blues and purples).

We redid the implementation above but for $h = \frac{5}{2}, \frac{5}{4}, \frac{5}{8}$, and $\frac{5}{16}$ this time, were the resulting solutions are shown in Figure 2.

In the top left panel, corresponding to $h = \frac{5}{2} = 2.5$, we can see that the solution is relatively smooth with spaced out contours. As h decreases the contours become more compact as seen in the panels corresponding to $h = \frac{5}{4} = 1.25$, and $h = \frac{5}{8} = 0.625$. The contours become the most compact at $h = \frac{5}{16} = 0.3125$.

Then, we considered the voltage $u(x, y)$ inside a condenser with parallel plates separated by a distance L obeying the Laplace's equation (5) in the region $0 \leq x \leq L$, $0 \leq y \leq 1$. where the boundary conditions are given by: $u(x, 0) = 100 \sin\left(\frac{2\pi x}{L}\right)$, $u(x, 1) = -100 \sin\left(\frac{2\pi x}{L}\right)$, and $u(0, y) = u(L, y) = 0$. We used the Jacobi and Gauss-Seidel methods to solve for this problem on a 64×64 grid with $L = 1$ and convergence criteria $\epsilon = 1 \times 10^{-8}$ and obtained the plot in Figure 3.

Figure 3 shows the resulting solutions obtained by using Jacobi and Gauss-Seidel methods to solve for the voltage $u(x, y)$ inside a condenser with parallel plates. The left panel shows the solution obtained using Jacobi method, while the right panel shows the solution obtained using Gauss-Seidel method. It can be seen that both methods converge to the same solution. The voltage is highest at the top and bottom edges of the capacitor, and lowest at the center. We found that the Gauss-Seidel method had 1713 iterations while the Jacobi method had 8768 iterations and so it appeared that the Gauss-Seidel method to converges faster than the Jacobi method as expected.

Then, we looked at an equation frequently used as a model for Quantum mechanics, The Schrödinger's equation. We compared two methods to solve this equation: (i) The direct method; (ii) The weighted Jacobi method, using the boundary conditions $u(0, y) = u(1, y) = u(x, 0) = u(x, 1) = 1$. We solved the equation for $0 \leq x \leq 1.0$.

First, for $\alpha = 1.0$, we applied the direct method in a 50×50 grid and found the solution and obtained the graph in Figure 4, where we plotted $u(x, y)$ vs (x, y).

Second, we used the weighted Jacobi method in a 50×50 grid to explore the effect of the weighting factor ω on the solution and so we did runs for $\omega = 0.1, 0.25, 0.5, 0.75$, and 0.9, and obtained five plots shown in Figure 5. We found that the effect of the weighting factor *omega* on the solution can be observed by comparing the convergence rate of the weighted Jacobi method for different values of ω. A smaller value resulted in a slower convergence

rate, while a larger value of resulted in instability in the solution. However, the plots appeared to be the same because they converge to the same solution and the range of values for the solution $u(x, y)$ is fixed for all the plots.

Now instead of $\alpha = 1$, we redid the above for $\alpha = 1000$ for the direct method, where we obtained the plot in Figure 6, and for the Weighted Jacobi method, were we obtained the plot in Figure 7. The plots also appeared to be the same in this case because they converge to the same solution and the range of values for the solution $u(x, y)$ is fixed for all the plots, however, We have found that a larger value of α required more iterations to converge.

We looked at the Advection equation as representing a concentration $u(x, t)$ and c the speed of the flow so that in 1-D we have $u_t = cu_x$ with $c > 0$ or $c < 0$. The exact analytical solution of the 1-D advection equation using the method of characteristics is $u(x, t) = u_0(x + ct)$ where $u_0(x) = u(x, 0)$. We have found that if one sets a pulse as the initial function u_0, if c > 0, the flow will move to the right meaning it will move in the positive x-direction, and so the pulse will move in that direction with speed c. If c < 0, the flow will move to the left meaning it will move in the negative x-direction, and so the pulse will move in that direction with speed c.

Conclusion

In conclusion, partial differential equations are equations that include derivatives of an unknown function with respect to two or more independent variables. Time-independent PDEs are classified as elliptic cases and require defining boundary conditions to solve. The three types of boundary conditions are Dirichlet, Neumann, and mixed.

One method to solve time-independent PDEs numerically is the direct method, which involves approximating derivatives using numerical tricks and then writing all equations into a single matrix to solve. The Jacobi and Gauss-Seidel methods are two commonly used iterative methods for solving time-independent PDEs. We have explored both methods and found that Gauss-Seidel method to converges faster than the Jacobi method. We also implemented the direct method the Weighted Jacobi method for the Schrödinger's equation and plotted their solutions.

Appendix

Tables

Name	FD scheme	Order of accuracy	CFL stability restriction
	Explicit		
Forward Euler	$u_j^{n+1} = u_j^n + \frac{1}{2}r\left(u_{j+1}^n - u_{j-1}^n\right)$	1	$r \leq 1$
Upwind	$u_j^{n+1} = u_j^n + r\left(u_{j+1}^n - u_j^n\right)$	1	$r \leq 1$
Leap-Frog	$u_j^{n+1} = u_j^{n-1} + r\left(u_{j+1}^n - u_{j-1}^n\right)$	2	$r \leq 1$
Lax-Wendroff	$u_j^{n+1} = u_j^n + \frac{1}{2}r\left(u_{j+1}^n - u_{j-1}^n\right) + \frac{1}{2}r^2\left(u_{j+1}^n - 2u_j^n + u_{j-1}^n\right)$	2	$r \leq 1$
	Implicit		
Backward Euler	$u_j^{n+1} = u_j^n + \frac{1}{2}r\left(u_{j+1}^{n+1} - u_{j-1}^{n+1}\right)$	1	None
Cranck-Nicolson	$u_j^{n+1} = u_j^n + \frac{1}{4}r\left(u_{j+1}^n - u_{j-1}^n\right) + \frac{1}{4}r\left(u_{j+1}^{n+1} - u_{j-1}^{n+1}\right)$	2	None

Table 1: PDEs: Explicit and Implicit methods

Plots

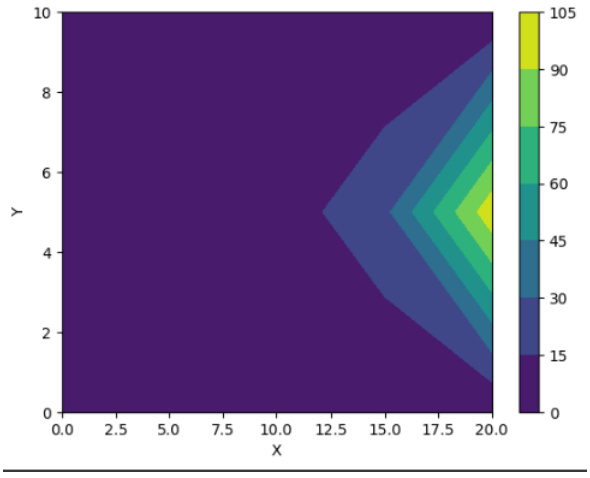

Figure 1: $u(x, y)$ inside the box using Jacobi Method

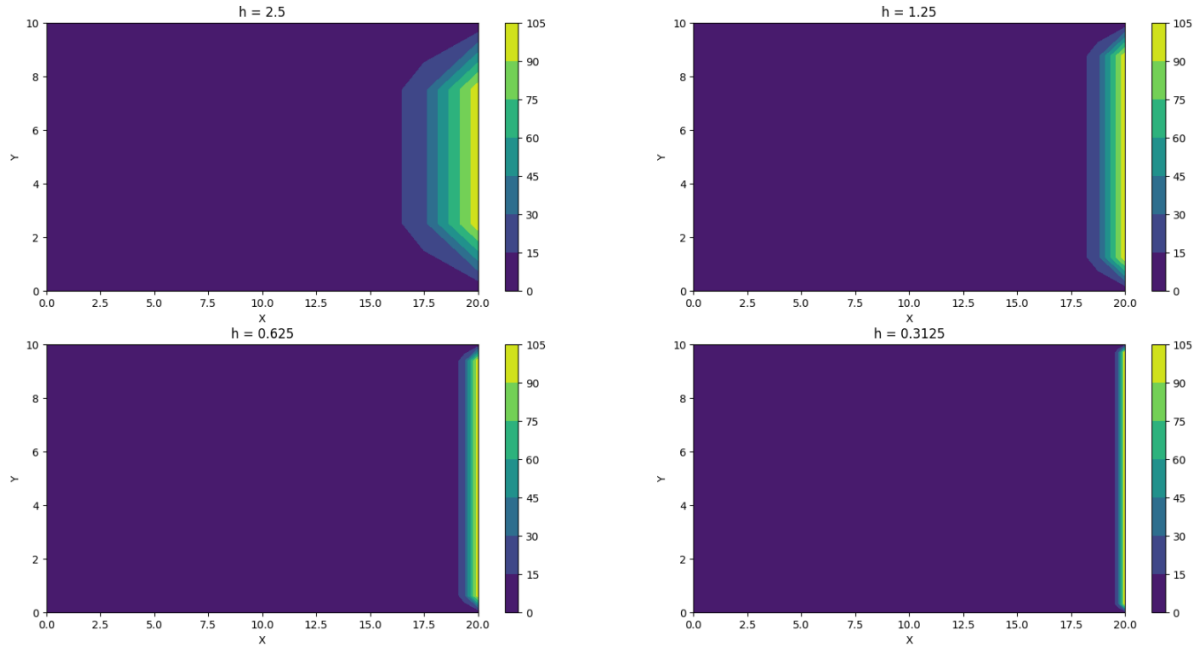

Figure 2: $u(x, y)$ inside the box using Jacobi Method for $h = 2.5, 1.25, 0.625, and 0.3125$

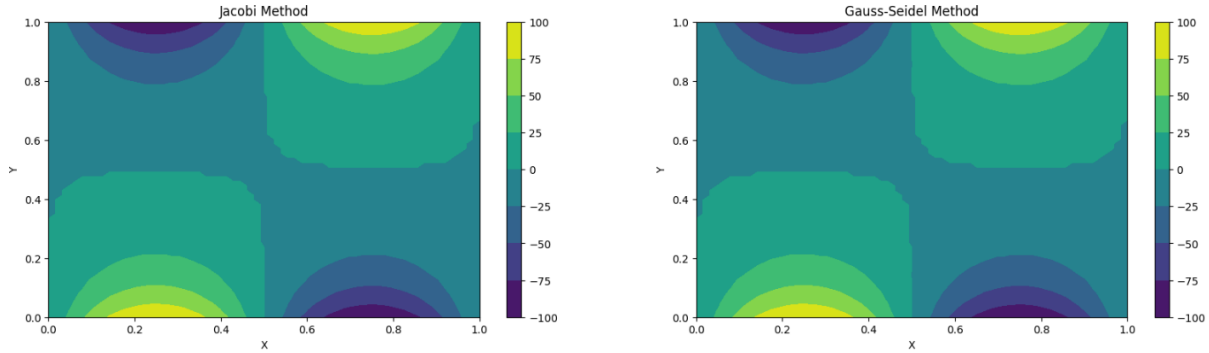

Figure 3: Jacobi vs Gauss-Seidel Methods

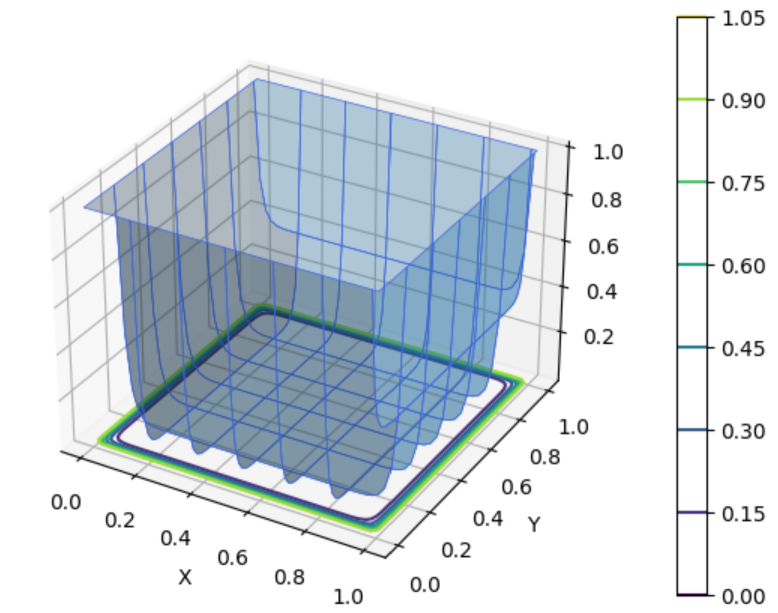

Figure 4: Direct Method for $\alpha = 1$

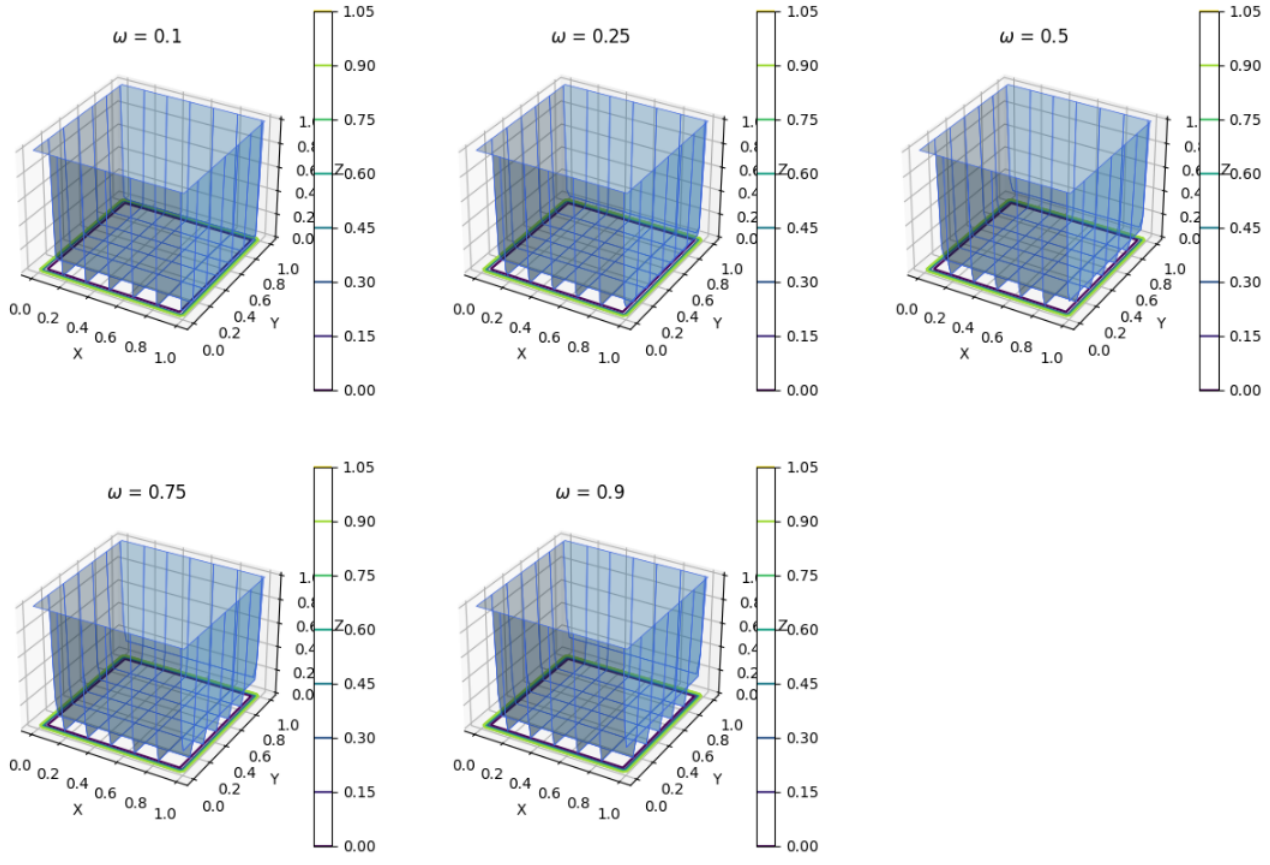

Figure 5: Weighted Jacobi Method for $\alpha = 1$ for different ω

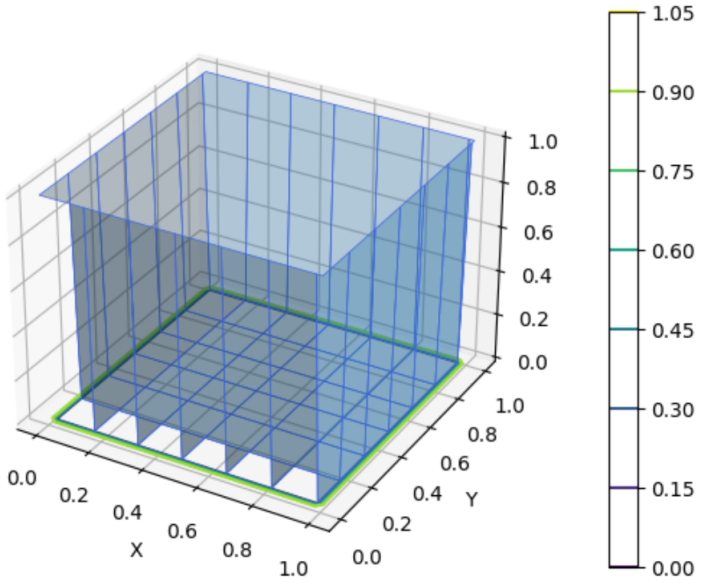

Figure 6: Direct Method for $\alpha = 1000$

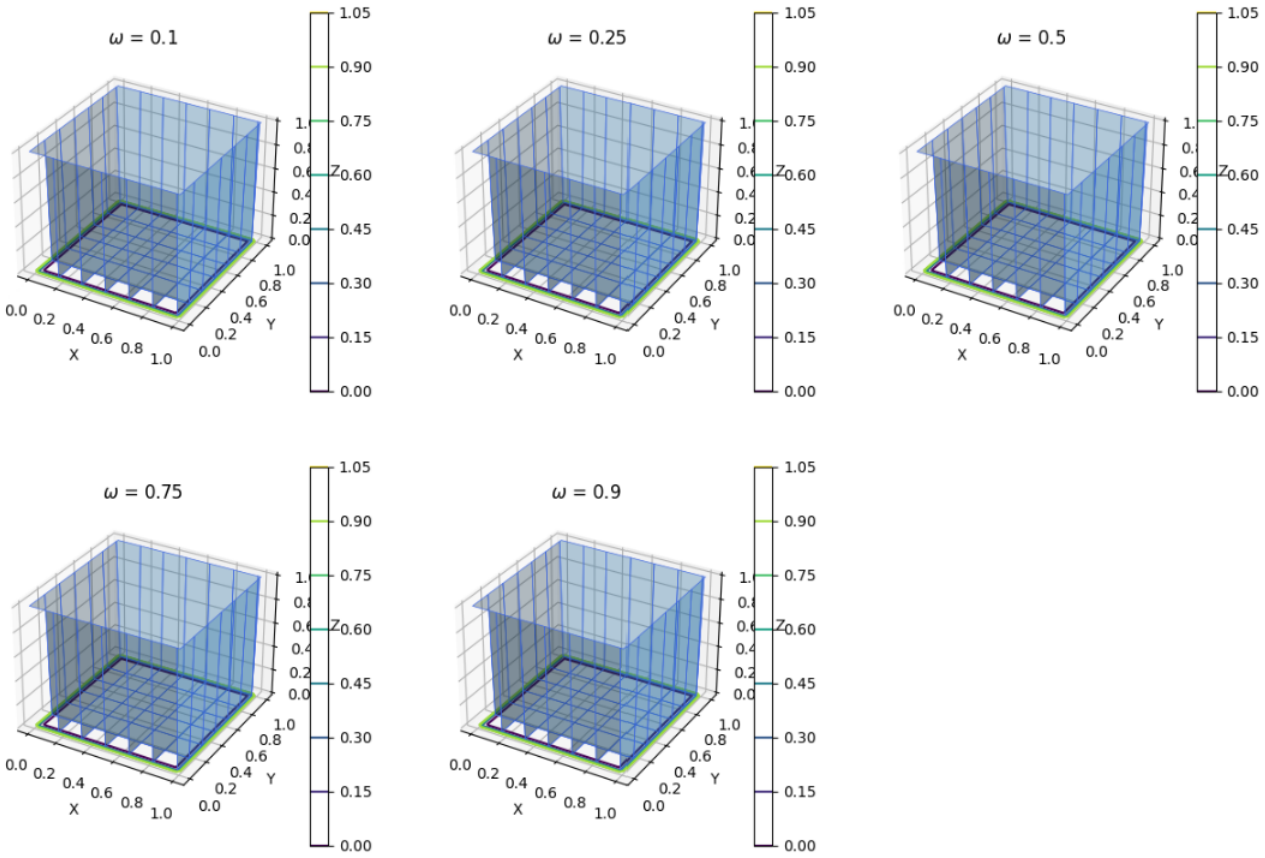

Figure 7: Weighted Jacobi Method for $\alpha = 1000$ for different ω

Code

```
    # -*- coding: utf-8 -*-
"""Lab 3.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1pzESEAF1PyRJ0ZMsRXupYcv8e72ngZhT
"""

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import scipy


def matrix_Ab(grid):

  u_numbery = (len(grid)-2)
  u_numberx = (len(grid[0])-2)
  A = np.full((u_numberx*u_numbery, u_numberx * u_numbery), 0)
  b = np.full(u_numberx*u_numbery, 0)

  u_range_y = np.arange(1, u_numbery + 1)
  u_range_x = np.arange(1, u_numberx + 1)

  k = 0
  for i in u_range_y:
    for j in u_range_x:

      A[k][k] = 4

      for l in [-1,1]:
        shift_x = j + l

        if shift_x in u_range_x:
          A[k][k + l]  = -1
        else:
          b[k] += grid[i][shift_x]

      for l in [-1,1]:
        shift_y = i + l
```

```
        if shift_y in u_range_y:
          A[k][k + u_numbery*l]  = -1
        else:
          b[k] += grid[shift_y][j]

      k += 1
  return A, b

h = 5
x_size = 20 // 5 + 1
y_size = 10 // 5 + 1

stencil = np.full((y_size, x_size), 0)

for i in range(1,y_size-1):
  stencil[i][-1] = 100

A, b = matrix_Ab(stencil)


L, U, D = np.tril(A, -1), np.triu(A, 1), np.diag(np.diag(A))


invD = scipy.linalg.inv(D)

LU_sub = -L-U
C = np.matmul(invD, LU_sub)
d = invD * b

def jacobiMethod2(grid, m = 10, error = 1e-8, error_find = False):
  A, b = matrix_Ab(grid)
  L, U, D = np.tril(A, -1), np.triu(A, 1), np.diag(np.diag(A))

  invD = scipy.linalg.inv(D)

  LU_sub = -L-U
  C = np.matmul(invD, LU_sub)
  d = np.matmul(invD,b)

  u = np.zeros(len(b))

  count = 0
  if error_find:
```

```
    current_error = 1
    while current_error > error:
      u = np.matmul(C, u) + d
      current_error = np.max(np.matmul(A, u) - b)
      count += 1

  else:
    for i in range(m):
      u = np.matmul(C, u) + d
    count = m

  solved_grid = grid.copy()
  k = 0
  for i in range(1, len(grid)-1):
    for j in range(1, len(grid[0])-1):
      solved_grid[i][j] = u[k]
      k += 1

  return solved_grid, count

solution_2 = jacobiMethod2(stencil, m=40)
print(solution_2)

x = np.arange(0, 21, 5)
y = np.arange(0, 11, 5)

X, Y = np.meshgrid(x,y)

plt.contourf(X,Y, solution_2[0])
plt.colorbar()
plt.xlabel("X")
plt.ylabel("Y")

solutions = []

h = [5/2, 5/4, 5/8, 5/16]

for i in h:
  x_size = round(20 // i + 1)
  y_size = round(10 // i + 1)

  stencil = np.full((y_size, x_size), 0)
```

```
  for i in range(1,y_size-1):
    stencil[i][-1] = 100

  solutions.append(jacobiMethod2(stencil, error_find=True)[0])

fig = plt.figure(figsize=(20, 10))

for i, j in enumerate(h):
  x = np.arange(0, 20.1, j)
  y = np.arange(0, 10.1, j)

  X, Y = np.meshgrid(x,y)

  ax = fig.add_subplot(2, 2, i+1)
  contour = ax.contourf(X,Y, solutions[i])
  fig.colorbar(contour)
  ax.set_xlabel("X")
  ax.set_ylabel("Y")
  ax.set_title("h = " + str(j))

def gaussSeidel2(grid, m = 10, error = 1e-8, error_find = False):
  A, b = matrix_Ab(grid)
  L, U, D = np.tril(A, -1), np.triu(A, 1), np.diag(np.diag(A))


  DL_inv = scipy.linalg.inv(D+L)

  u = np.zeros(len(b))

  count = 0
  if error_find:
    current_error = 1
    while current_error > error:
      u = np.matmul(DL_inv,np.matmul(-U, u)+b)
      current_error = np.max(np.matmul(A, u) - b)
      count += 1

  else:
    for i in range(m):
      u = np.matmul(DL_inv,np.matmul(-U, u)+b)
    count = m

  solved_grid = grid.copy()
```

```
  k = 0
  for i in range(1, len(grid)-1):
    for j in range(1, len(grid[0])-1):
      solved_grid[i][j] = u[k]
      k += 1

  return solved_grid, count

x_size = 64
y_size = 64

x = np.linspace(0, 1, 64)
y = np.linspace(0, 1, 64)

stencil = np.full((x_size, y_size), 0)

for i in range(1,x_size-1):
  stencil[0][i] = 100 * np.sin(2 * np.pi * x[i])

for i in range(1,x_size-1):
  stencil[-1][i] = - 100 * np.sin(2 * np.pi * x[i])

data_jac, itteration_jac = jacobiMethod2(stencil, error_find=True)
data_gaus, itteration_gaus = gaussSeidel2(stencil, error_find=True)

fig = plt.figure(figsize=(20, 5))



X, Y = np.meshgrid(x,y)

ax = fig.add_subplot(1, 2, 1)
contour = ax.contourf(X,Y, data_jac)
fig.colorbar(contour)
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_title("Jacobi Method")

ax = fig.add_subplot(1, 2, 2)
contour = ax.contourf(X,Y, data_gaus)
fig.colorbar(contour)
ax.set_xlabel("X")
ax.set_ylabel("Y")
```

```
ax.set_title("Gauss-Seidel Method")

print("Number of Itterations till convergence for Jacobi Method " + str(itteration_jac))
print("Number of Itterations till convergence for Gauss-Seidel Method " + str(itteration_gaus))

def matrix_Ab_2(grid, alpha):

  u_numbery = (len(grid)-2)
  u_numberx = (len(grid[0])-2)
  A = np.full((u_numberx*u_numbery, u_numberx * u_numbery), 0)
  b = np.full(u_numberx*u_numbery, 0)

  u_range_y = np.arange(1, u_numbery + 1)
  u_range_x = np.arange(1, u_numberx + 1)

  k = 0
  for i in u_range_y:
    for j in u_range_x:

      A[k][k] = 4 + alpha

      for l in [-1,1]:
        shift_x = j + l

        if shift_x in u_range_x:
          A[k][k + l]  = -1
        else:
          b[k] += grid[i][shift_x]

      for l in [-1,1]:
        shift_y = i + l
        if shift_y in u_range_y:
          A[k][k + u_numbery*l]  = -1
        else:
          b[k] += grid[shift_y][j]

      k += 1
  return A, b

x_size = 50
y_size = 50

grid_schro = np.zeros((50,50))
```

```
for i in range(0,x_size):
  grid_schro[0][i] = 1

for i in range(0,x_size):
  grid_schro[-1][i] = 1

for i in range(1,y_size-1):
  grid_schro[i][-1] = 1

for i in range(1,y_size-1):
  grid_schro[i][0] = 1

A_schro, b_schro = matrix_Ab_2(grid_schro, 1)

solved_grid_schro = grid_schro.copy()

x_schro  = np.linalg.solve(A_schro, b_schro)
k = 0
for i in range(1, len(grid_schro)-1):
  for j in range(1, len(grid_schro[0])-1):
    solved_grid_schro[i][j] = x_schro[k]
    k += 1

x = np.linspace(0,1, x_size)
y = np.linspace(0, 1, y_size)

X, Y = np.meshgrid(x,y)

fig = plt.figure(figsize=(14,5))
ax = fig.add_subplot(111, projection='3d')

ax.plot_surface(X, Y, solved_grid_schro,edgecolor='royalblue', lw=0.5, rstride=8, cstride=8,
                alpha=0.3)

p = ax.contour(X, Y, solved_grid_schro, offset=0)


# Add color bar
cb = fig.colorbar(p)


# Set labels and title
```

```
ax.set_xlabel('X')
ax.set_ylabel('Y')


# Show plot
plt.show()

def weightedJacobiMethod2(grid, w, a,m = 10, error = 1e-8, error_find = False):
  A, b = matrix_Ab_2(grid, a)
  L, U, D = np.tril(A, -1), np.triu(A, 1), np.diag(np.diag(A))

  invD = scipy.linalg.inv(D)

  LU_sub = -L-U
  C = np.matmul(invD, LU_sub)
  d = np.matmul(invD,b)

  u = np.zeros(len(b))

  count = 0
  if error_find:
    current_error = 1
    while current_error > error:
      u = w * (np.matmul(C, u) + d) + (1- w) * u
      current_error = np.max(np.matmul(A, u) - b)
      count += 1

  else:
    for i in range(m):
      u = w * (np.matmul(C, u) + d) + (1- w) * u
    count = m

  solved_grid = grid.copy()
  k = 0
  for i in range(1, len(grid)-1):
    for j in range(1, len(grid[0])-1):
      solved_grid[i][j] = u[k]
      k += 1

  return solved_grid, count

w = [0.1,0.25,0.5,0.75, 0.9]
schro_data = []
```

```
for i in w:
  schro_data.append(weightedJacobiMethod2(grid_schro, i, 1,error_find = True)[0])

fig = plt.figure(figsize=(15, 10))

for i in range(len(schro_data)):
  ax = fig.add_subplot(2, 3, i+1, projection='3d')
  ax.plot_surface(X, Y, schro_data[i],edgecolor='royalblue', lw=0.5, rstride=8, cstride=8,
                alpha=0.3)
  p = ax.contour(X, Y, schro_data[i], offset=0)


  # Add color bar
  cb = fig.colorbar(p)

  ax.set_xlabel("X")
  ax.set_ylabel("Y")
  ax.set_zlabel("Z")
  ax.set_title("$\omega$ = " + str(w[i]))



plt.show()

A_schro, b_schro = matrix_Ab_2(grid_schro, 1000)

solved_grid_schro = grid_schro.copy()

x_schro  = np.linalg.solve(A_schro, b_schro)
k = 0
for i in range(1, len(grid_schro)-1):
  for j in range(1, len(grid_schro[0])-1):
    solved_grid_schro[i][j] = x_schro[k]
    k += 1

fig = plt.figure(figsize=(14,5))
ax = fig.add_subplot(111, projection='3d')

ax.plot_surface(X, Y, solved_grid_schro,edgecolor='royalblue', lw=0.5, rstride=8, cstride=8,
                alpha=0.3)

p = ax.contour(X, Y, solved_grid_schro, offset=0)
```

```
# Add color bar
cb = fig.colorbar(p)


# Set labels and title
ax.set_xlabel('X')
ax.set_ylabel('Y')


# Show plot
plt.show()

w = [0.1,0.25,0.5,0.75, 0.9]
schro_data = []
for i in w:
  schro_data.append(weightedJacobiMethod2(grid_schro, i, 1000,error_find = True)[0])

fig = plt.figure(figsize=(15, 10))

for i in range(len(schro_data)):
  ax = fig.add_subplot(2, 3, i+1, projection='3d')
  ax.plot_surface(X, Y, schro_data[i],edgecolor='royalblue', lw=0.5, rstride=8, cstride=8,
                alpha=0.3)
  p = ax.contour(X, Y, schro_data[i], offset=0)


  # Add color bar
  cb = fig.colorbar(p)

  ax.set_xlabel("X")
  ax.set_ylabel("Y")
  ax.set_zlabel("Z")
  ax.set_title("$\omega$ = " + str(w[i]))



plt.show()

def forwardEuler(u, r):
  A = np.zeros((len(u), len(u)))
  for i in range(len(A)):
    if i > 0 and i < len(u)-1:
```

```
      A[i][i+1] = 1/2 * r
      A[i][i-1] = -1/2 * r
      A[i][i] = 1
    elif i == 0:
      A[i][i] = 1
      A[i][i+1] = 1/2 * r
    elif i == len(u) - 1:
      A[i][i] = 1
      A[i][i-1] = -1/2 * r

  b = np.zeros(len(u))

  b[0] = - 1 / 2 * r
  b[-1] = 1 / 2 * r

  return np.matmul(A, u) + b

dt_list = [0.02, 0.02, 0.02, 0.02, 0.02, 0.01, 0.04]
dx_list = [0.04, 0.02, 0.0137, 0.0101, 0.099, 0.02, 0.02]
r_list = [0.25, 0.5, 0.728, 0.99, 1.11, 0.25, 1]

def sin_initial(x):
  return np.sin(2 * x)

def exp_initial(x):
  return np.exp(-4*x**2)

data_1 = []
data_2 = []

for dt, dx, r in zip(dt_list, dx_list, r_list):
  t_range = np.arange(0, 2, dt)
  x_range = np.arange(-2, 2, dx)

  data_temp1 = np.zeros((len(t_range), len(x_range)))

  data_temp1[0] = sin_initial(x_range)

  for i in range(1, len(data_temp1)):
    data_temp1[i] = forwardEuler(data_temp1[i-1], r)

  data_1.append(data_temp1)
```

```
  data_temp2 = np.zeros((len(t_range), len(x_range)))

  data_temp2[0] = exp_initial(x_range)

  for i in range(1, len(data_temp2)):
    data_temp2[i] = forwardEuler(data_temp2[i-1], r)

  data_2.append(data_temp2)

t_range = np.arange(0, 2, 0.02)
x_range = np.arange(-2, 2, 0.04)

X, T = np.meshgrid(x_range,t_range)

plt.contourf(X,Y, data_1[0])
plt.colorbar()
plt.xlabel("X")
plt.ylabel("Y")
```