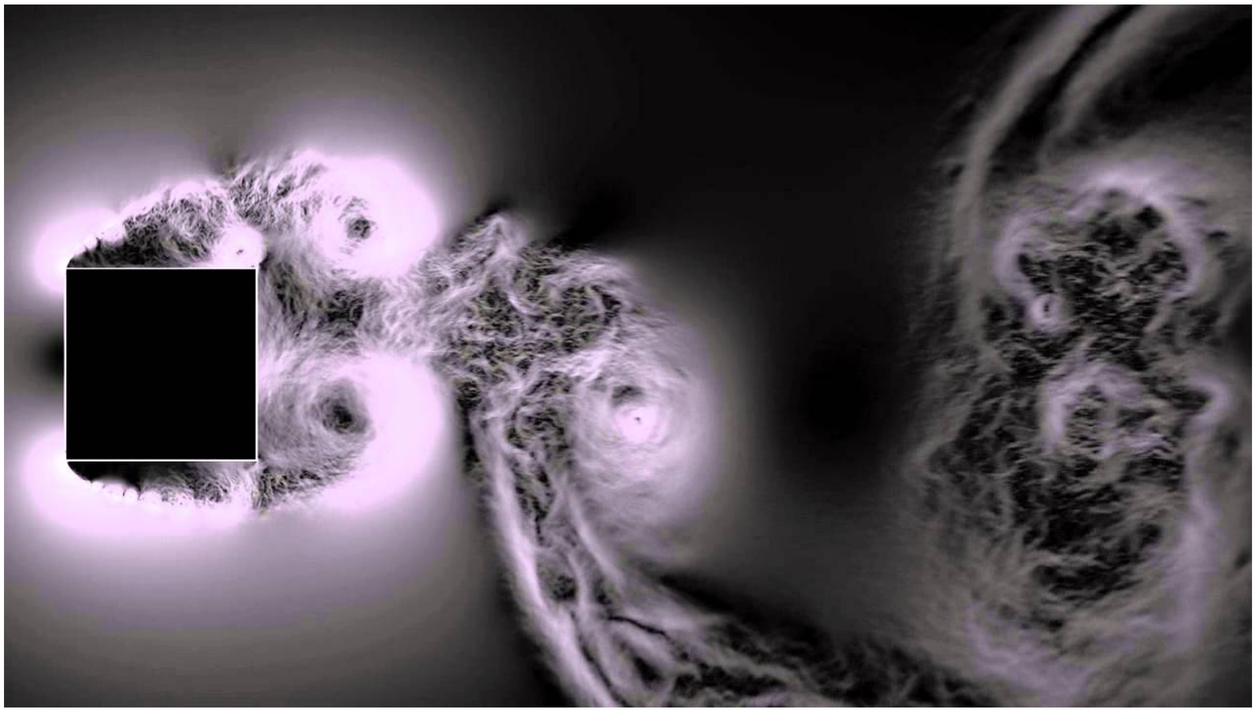
Project 1: Flow Over a Square

Aerospace 523: Computational Fluid Dynamics I
Graduate Aerospace Engineering
University of Michigan, Ann Arbor

By: Dan Card[*], dcard@umich.edu
Date: October 9, 2020



[*]Graduate Aerospace Engineering, AIAA Student Member, Aerospace Honors Society.

COLLEGE OF ENGINEERING
AEROSPACE ENGINEERING
UNIVERSITY OF MICHIGAN

Contents

List of Figures

List of Algorithms

List of Equations

1 Introduction

1.1 Overview

A flow that is incompressible and irrotational can be modeled by Laplace's equation for a scalar velocity potential. In two dimensions, an equivalent formulation is obtained using a stream function, $\psi(x, y)$, from which the velocity components are given by

$$u = \frac{\partial \psi}{\partial y}, \qquad v = -\frac{\partial \psi}{\partial x} \tag{1}$$

With this definition, continuity is automatically satisfied. Requiring the flow to be irrotational leads to Laplace's equation for ψ,

$$\nabla^2 \psi = 0 \tag{2}$$

In this project, I will be solving for two-dimensional potential flow around a square, as illustrated in Figure 1.

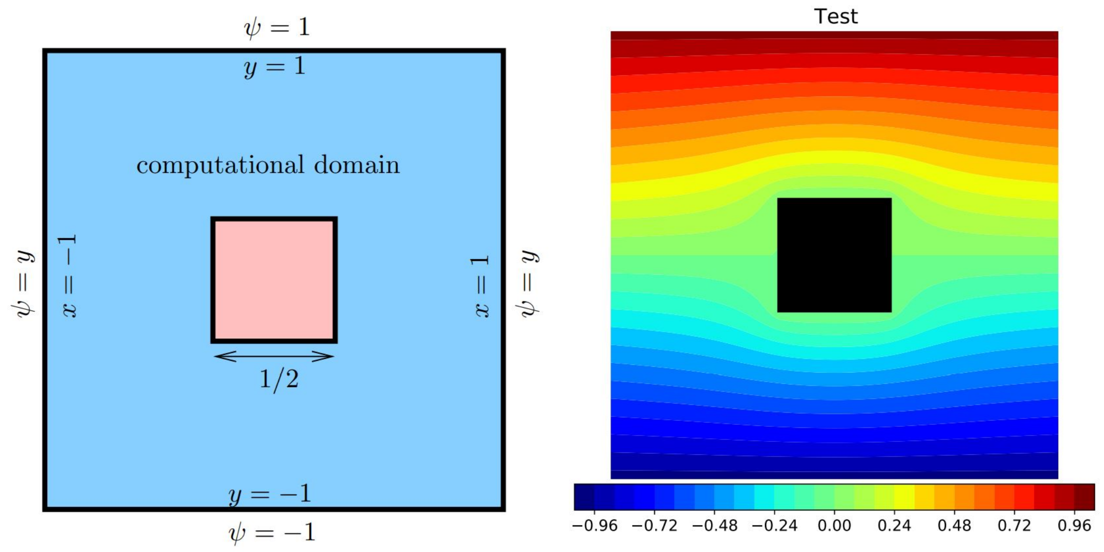


Figure 1: Solution of potential flow around a square, using a stream function, $\psi(x, y)$.

The computational domain is a square, $(x, y) \in [-1, 1]^2$, and the boundary conditions are Dirichlet. On the bottom and top walls, ψ is set to -1 and $+1$, respectively, and on the sides, ψ is set to y. The inner square, of side length $\frac{1}{2}$ and placed in the center of the outer square, presents an obstacle to the flow, and by symmetry it corresponds to a streamline on which $\psi = 0$.

1.2 Discretization

I will use a finite-difference method to solve Equation 2 on the computational domain. The grid consists of a lattice of $(N+1)^2$ points, as shown in Figure 2. Some of these points are on the boundary, and some are inside the inner square, hence outside the computational domain. It is up to me to decide how best to deal with these points. N is the number of intervals across the entire domain, so that the spacing is $\Delta x = \Delta y = h = 2/N$.

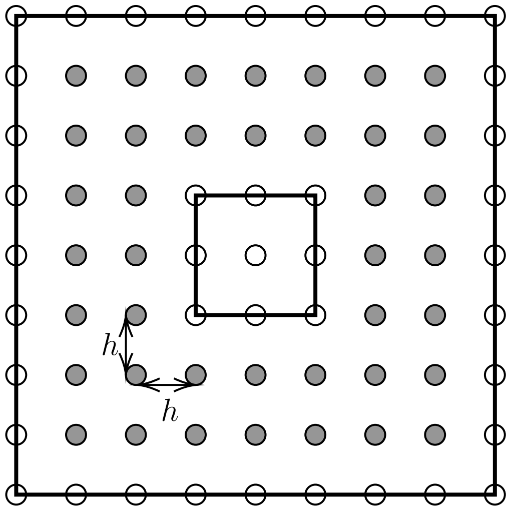


Figure 2: Finite difference grid for $p = 0 \rightarrow N = 8$.

To ensures that the grid conforms to the boundary of the inner square, use

$$N = 2^{p+3}, \qquad p = [0,\ 1,\ 2, \ldots] \tag{3}$$

At each interior node, I will use a standard second-order five-point stencil to discretize the Laplacian in Equation 2. Note that the given Dirichlet conditions fully specify ψ on the boundaries.

1.3 Solvers

There are three types of solvers that I will implement to study the flow around the square.

1.3.1 Direct

The direct solver will build a *sparse* linear system of equations to solve for the nodal states. The system will take the form

$$\underline{\underline{A}}\underline{\Psi} = \underline{F} \tag{4}$$

where $\underline{\Psi}$ is the unrolled state vector of unknowns. Solve this system using a sparse direct solver, such as the backslash operator in Matlab or `scipy.sparse.linalg.spsolve()`.

1.3.2 Iterative Smoothers

For the other solver I will implement two iterative smoothers: under-relaxed Jacobi and overrelaxed Gauss-Seidel. For Gauss-Seidel, I will use the "red-black" ordering, in which the nodes are colored in checkerboard fashion and the smoother is applied first to the red nodes and then to the black nodes. The presence of the inner square does not change the red-black ordering: imagine a checkerboard with the center cut out. I will use under-relaxation for Jacobi and over-relaxation for Gauss-Seidel.

1.3.3 Multigrid

The last solver I will use will implement a V-cycle in which successively finer grids are obtained by increasing p in Equation 4. Use full-weighting for the restriction operator, I_{2h}^{h}, and interpolation for the prolongation operator, I_{h}^{2h}. On the down/up sweep of the V-cycle perform $\nu_1 = \nu_2 = 2$ pre/post smoothing iterations. On the coarsest grid, perform $\nu_c = 50$ smoothing iterations. For the initial conditions in all iterative runs, use $\psi = 0$ at the interior nodes. For the smoother, use Gauss-Seidel with an over-relaxation factor of $\omega = 1.5$. *Again, implement these without any matrices whose size scales with the number of unknowns*

1.4 Post-Processing

To visualize the flowfield, plot contours of the stream function, as shown in Figure 1. These contours are streamlines of the flow. The top and bottom of the domain represent impenetrable walls, since ψ is set to a constant there. Acceleration of the fluid around the square changes the pressure of the fluid, and this affects the force on the walls. Of interest will be the pressure coefficient distribution on the bottom wall,

$$c_p(x) = 1 - \frac{u^2}{U_\infty^2}, \tag{5}$$

where the free-stream speed (speed of the flow without the inner square present) is $U_\infty = 1$ based on the boundary conditions. Note that u is obtained by differentiating ψ, according to Equation 1. Use a second-order one-sided finite difference of the ψ data to obtain u at the bottom boundary nodes.

Integrating and normalizing the pressure coefficient gives the lift coefficient on the bottom wall,

$$c_l(x) = \frac{1}{2}\int_0^2 c_p(x)\, dx \tag{6}$$

In this approximation I will use the trapezoidal method to perform this integration.

2 Tasks and Deliverables

In this project I will implement a finite-element solver to approximate the flow field around a square and verify its coefficients of lift and pressure. Additionally I will implement Jacobi and Gauss-Seidel smoothers and determine their convergences to theory. Lastly, I will implement a V-cycle multigrid method and will determine the best level so that the system converges to the appropriate solution.

2.1 Solving the Finite-Difference System

Since the boundary conditions have been specified, I will use a 5-point stencil about the interior nodes to approximate the global $\underline{\underline{A}}$ sparse matrix that will approximate the nodal values. After assembling the $\underline{\underline{A}}$ matrix I will reduce it to solve for the interior $\underline{\psi}$ nodal values. Where the 5-point stencil can be written to be,

$$\frac{1}{h^2}\begin{bmatrix} +u_{i,j+1} \\ +u_{i-1,j} - 4u_{i,j} + u_{i+1,j} \\ u_{i,j-1} \end{bmatrix} = f_{i,j} \tag{7}$$

Using the 5-point stencil shown above in Equation 7, and the coefficients to form the global stiffness matrix $\underline{\underline{A}}$ and using the Dirichlet boundary conditions to form $\underline{F}$ through Python I can solve directly for the approximated solution for ψ. After implementing the finite difference system shown in Algorithm 1 and using the direct solver the 9×9 matrix for $p = 0$, the re-shaped $\underline{\Psi}$ matrix becomes,

$$\boxed{\underline{\psi} = \begin{bmatrix}
1.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 & 1.0000 \\
0.7500 & 0.7336 & 0.7137 & 0.6920 & 0.6849 & 0.6920 & 0.7137 & 0.7336 & 0.7500 \\
0.5000 & 0.4709 & 0.4291 & 0.3692 & 0.3558 & 0.3692 & 0.4291 & 0.4709 & 0.5000 \\
0.2500 & 0.2208 & 0.1625 & 0.0000 & 0.0000 & 0.0000 & 0.1625 & 0.2208 & 0.2500 \\
0.0000 & -0.0000 & -0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\
-0.2500 & -0.2208 & -0.1625 & 0.0000 & 0.0000 & 0.0000 & -0.1625 & -0.2208 & -0.2500 \\
-0.5000 & -0.4709 & -0.4291 & -0.3692 & -0.3558 & -0.3692 & -0.4291 & -0.4709 & -0.5000 \\
-0.7500 & -0.7336 & -0.7137 & -0.6920 & -0.6849 & -0.6920 & -0.7137 & -0.7336 & -0.7500 \\
-1.0000 & -1.0000 & -1.0000 & -1.0000 & -1.0000 & -1.0000 & -1.0000 & -1.0000 & -1.0000
\end{bmatrix}}$$

2.2 Post-Processing Functions

Firstly, to approximate the coefficient of pressure about the bottom of the wall I will use a second-order one-sided finite difference to approximate u given through,

$$u = \frac{\partial \psi}{\partial y} \approx \frac{-\frac{3}{2}\psi_N + 2\psi_{N-1} - \frac{1}{2}\psi_{N-2}}{h^2} \tag{8}$$

Using Equation 8 above I can approximate the horizontal velocity to then approximate the coefficient of pressure using Equation 5 for $p \in [0,\ 2,\ 4]$. Doing this approximation with the aid of Algorithm 2 results in Figure 3 shown below.

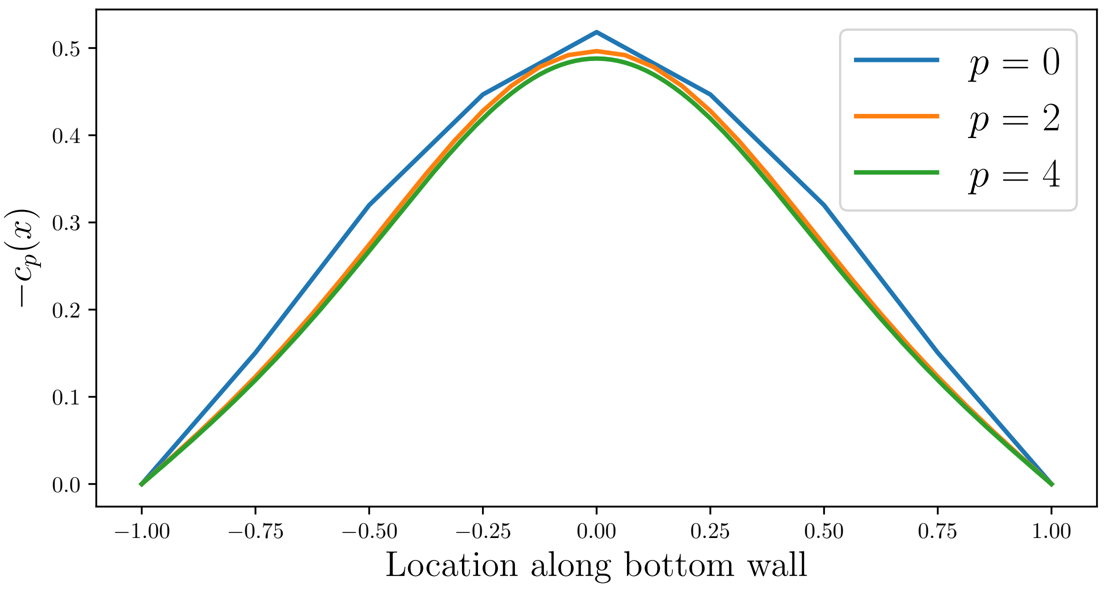


Figure 3: P-scaling coefficient of pressure $-c_p(x)$ along bottom wall.

Looking above to Figure 3, increasing p will simply "*smooth*" out the coefficient of pressure as it varies along x. Notable is that $-c_p(x)$ increases as it gets to the middle, where the middle of the square lies about the x-axis. This increase in $-c_p(x)$ indicates that the flow is increasing here as the coefficient of pressure indicates how the local flow is changing with respect to the ambient flow U_∞.

Nextly, after having approximated u, is to solve for the coefficient of lift c_l using Equation 6. However, to approximate this integral I will use the trapezoidal integration method shown below in Equation 9.

$$\int_a^b f(x)\ dx \approx \frac{\Delta x}{2}\left(f(x_N) + f(x_0)\right) + \Delta x \sum_{i=0}^{N} f(x_i) \tag{9}$$

Using Equation 9 to numerically approximate the coefficient of lift through Equation 6, I can approximate the coefficient of lift per specific p value. Through this method I will run a simulation with $p = 7$ to approximate an "*exact*" solution. Using this method I get the following results in Table 1 below.

Table 1: Coefficient of lift c_l values at p.

p	c_l
$p = 0$	-0.2940765
$p = 1$	-0.2795586
$p = 2$	-0.2706653
$p = 3$	-0.2665109
$p = 4$	-0.2647269
$p = 5$	-0.2639878

Conducting this simulation again with $p = 7$ approximates the "*exact*" solution to be,

$$\boxed{c_{l,exact}(p = 7) = -0.2635660}$$

Using this value to be the exact solution, I can then approximate the convergence rate of the simulations by taking the absolute value of the difference and comparing it to the p value and obtaining the rate of convergence as $r = \frac{\log_{10} \tau_{i+1}/\tau_i}{\log_{10} \Delta h_{i+1}/\Delta h_i}$. Conducting this convergence study gives that the approximated error is,

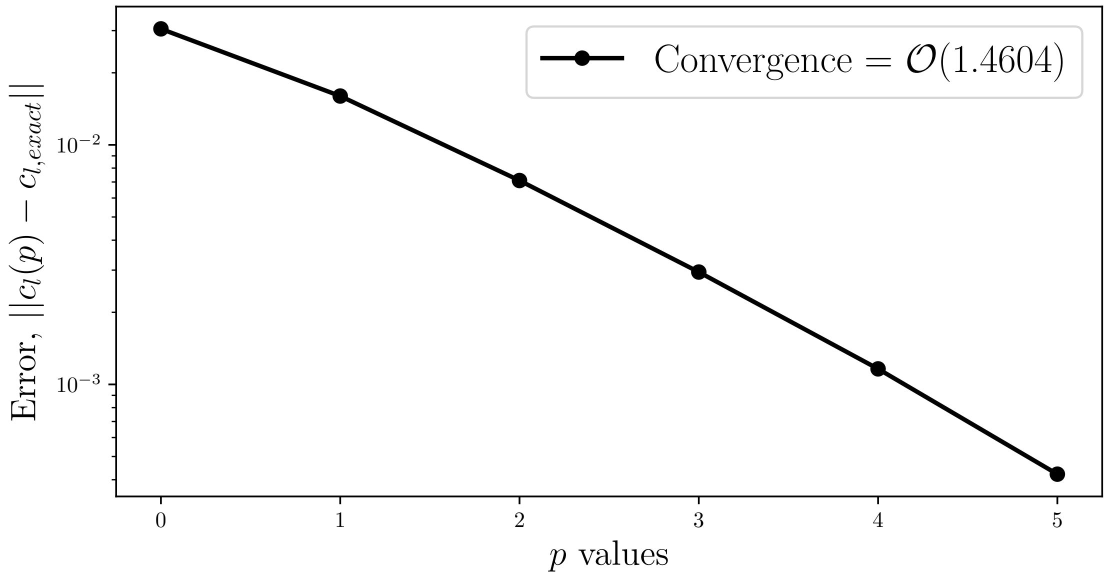


Figure 4: Approximated error and convergence while varying p.

Looking above to Figure 4 the slope of the convergence is $\mathcal{O}(\Delta h^2)$ hence second-order accuracy. This order of accuracy is consistent with the second-oder one-sided finite difference and the 5-point stencil and is the expected slope. This graph confirms the order of accuracy.

2.3 Implementing Jacobi and Gauss-Seidel Smoothers

In this task, I will implement both a Jacobi iterative smoother and a Gauss-Seidel smoother. Taking an iterative solution approach allows for easier implementation, but as a result will display a slower convergence. This convergence history will be highlighted here.

2.3.1 Jacobi Iteration Smoother

In order to implement the Jacobi Iteration Smoother I will use the expression for the next iteration state approximation shown below as,

$$u_{i,j}^{n+1} = \frac{1}{4}\left(u_{i-1,j}^{n} + u_{i+1,j}^{n} + u_{i,j-1}^{n} + u_{i,j-1}^{n} + \Delta h^2 f_i\right) \tag{10}$$

Using Equation 10 above, and implementing into Python code shown in Appendix 3, I can use this to approximate the next iteration state values and iterate several times until the approximation matches that of the analytical. Choosing the iteration values I choose to pick a high value of iterations until the approximated solution matched that of Figure 1. Then from here I conducted an L_2 residual norm and its convergence shown below.

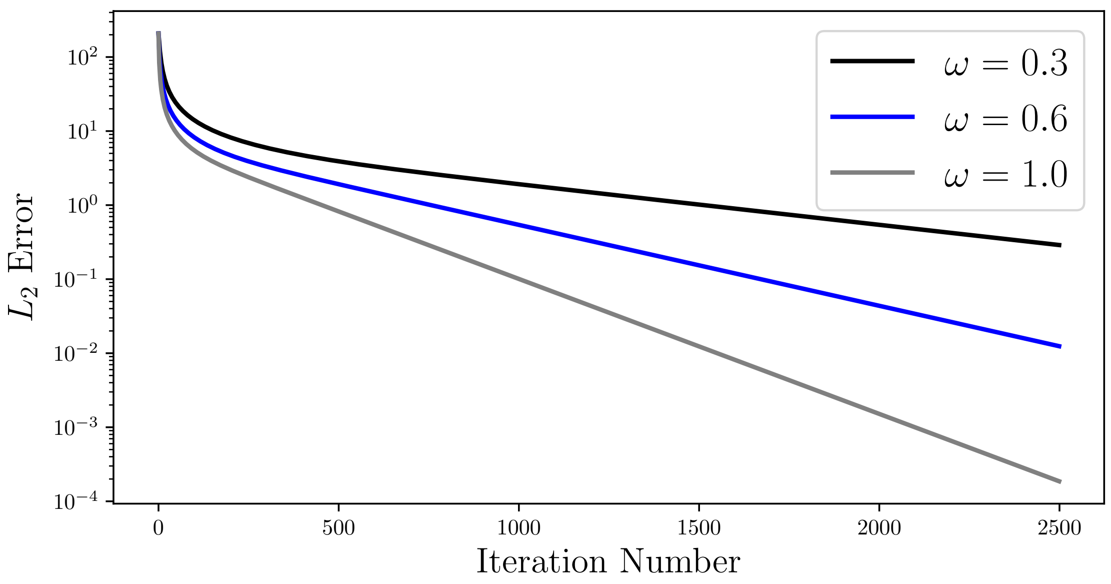


Figure 5: L_2 residual norm convergence history for varying ω.

As shown above in Figure 5, the over-relaxation factor plays a large role into how fast the solution converges. Further analysis on the *y-log* plot shows that it appears as if the multiple of the over-relaxation factor of ω denotes what its L_2 error will be as $\omega = 0.6$ has an L_2 residual norm that is approximately twice the magnitude that of $\omega = 0.3$ residual norm.

2.3.2 Gauss-Seidel Smoother

The Gauss-Seidel smoother is very similar in implementation to the Jacobi iteration smoother, however it differs in how the state updates. Gauss-Seidel will update half of the state first and then use the updated state to further smooth the approximated states in that given iteration. This method is called the "*checker-board*" or black-red update. This essentially means that every other node/adjacent node will get updated and then use these updated nodes to make a better approximation. Implementing this in Python can be shown in Appendix 4, but with the L_2 residual norm error shown below for differing values of over-relaxation.

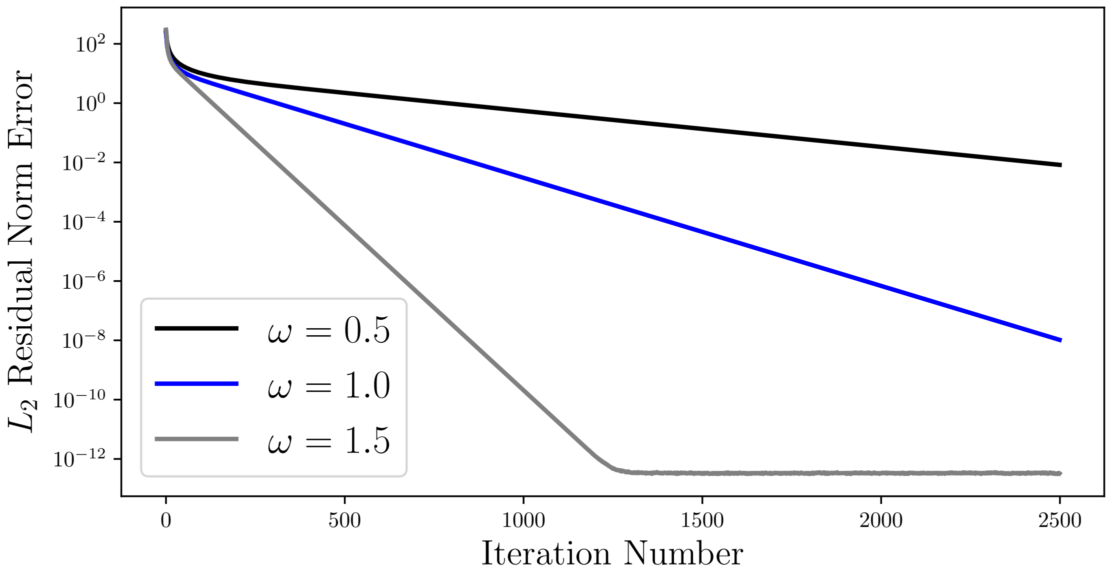


Figure 6: L_2 residual norm convergence history for varying ω.

Shown above in Figure 6, is the L_2 residual norm error through its iteration values. This shows again the the over-relaxation value does converge faster with a higher value. However, this may be an artifact of machine precision but it does appear that $\omega = 1.5$ will converge to a given solution whereas $\omega = 0.5$ or $\omega = 1.0$ may be able to converge to a more precise solution but would take longer to do so.

Across both the Jacobi iteration smoother and the Gauss-Seidel smoother, the rates at which they converge depend on the over-relaxation value. Having a higher over-relaxation value will allow the iterative solver to coverge faster. However, notable is that the Gauss-Seidel smoother can have a over-relaxation value greater than 1 or $\omega > 1$ and still converge. When Jacobi iteration smoother has an over-relaxation value greater than 1 it will diverge from the analytical solution whereas Gauss-Seidel continues to converge. Running Jacobi iteration with $\omega = 1.1$ gave that the final value was $\mathcal{O}(10^{140})$ which is in agreement that an over-relaxation value greater than 1 will diverge.

2.4 Implementing V-Cycle Multigrid Method

The method of the V-Cycle multigrid is computing the residual at a fine mesh, restricting it to a smaller grid then repeating until reaching the coarsest mesh grid size. At this coarsest meshgrid the restricted residual will be smoothed and then the prolongation scheme will start. In this prolongation the error is passed up through each prolongation and smoother and ultimately added to the initial guess of the state matrix.

2.4.1 V-Cycle Multigrid Method Convergence

Restriction Restricting the multigrid can be done by simply iterating by multiples of 2 throughout the x and y directions and taking a weighted average of the surrounding nodes such that the restriction can be written below in Equation 11,

$$r_{i,j} = \frac{1}{4} \underbrace{r_{i,j}}_{\text{Center node}} + \frac{1}{8} \underbrace{(r_{i\pm1,j} + r_{i,j\pm1})}_{\text{Up/Down nodes}} + \frac{1}{16} \underbrace{(r_{i\pm1,j\pm1} + r_{i\pm1,j\mp1})}_{\text{Corner nodes}} \tag{11}$$

Prolongation Prolongation is similar to restriction except that it is acting in the opposite direction. Instead of weighting adjacent nodes to a center node it applies one node to several. In my implementation I would iterate fully through the finer mesh but I would conduct integer division to use values from every other node and then weight them accordingly. The expression that I used for prolongation can be shown below in Equation 12

$$\begin{bmatrix} e_{i-1_h,j+1_h} & e_{i_h,j+1_h} & e_{i+1_h,j+1_h} \\ e_{i-1_h,j_h} & e_{i_h,j_h} & e_{i+1_h,j_h} \\ e_{i-1_h,j-1_h} & e_{i_h,j-1_h} & e_{i+1_h,j+1_h} \end{bmatrix} = \begin{bmatrix} \frac{1}{4}e_{i_{2h},j_{2h}} & \frac{1}{2}e_{i_{2h},j_{2h}} & \frac{1}{4}e_{i_{2h},j_{2h}} \\ \frac{1}{2}e_{i_{2h},j_{2h}} & e_{i_{2h},j_{2h}} & \frac{1}{2}e_{i_{2h},j_{2h}} \\ \frac{1}{4}e_{i_{2h},j_{2h}} & \frac{1}{2}e_{i_{2h},j_{2h}} & \frac{1}{4}e_{i_{2h},j_{2h}} \end{bmatrix} \tag{12}$$

Implementing Multigrid Using Equations 11, 12 and implementing into the Python environment with Algorithm 5, I ultimately arrive to find the convergence rates for the V-Cycle multigrid. Running several iterations with differing p values I get that the L_2 residual norms with $\nu_1 = \nu_2 = 2$ and $\nu_c = 50$ are shown in Figure 7 on the following page.

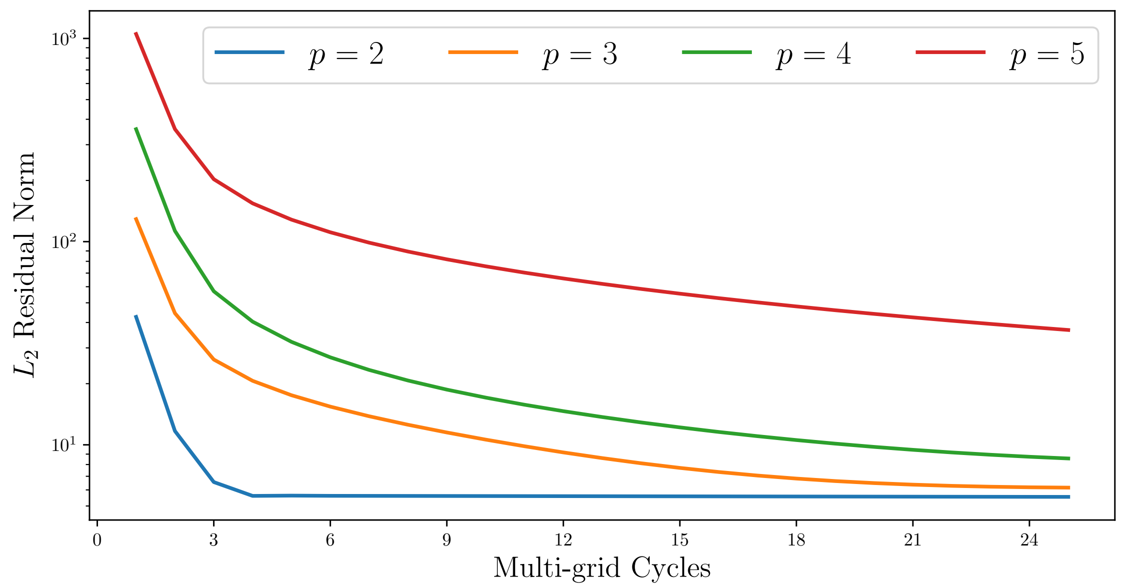


Figure 7: The V-Cycle multigrid method L_2 convergence while varying p.

Shown above in Figure 7, is the convergence for the V-cycle multigrid L_2 residual norm. Looking to the convergence rates, as you increase the size of the meshgrid (increase of p) the longer it will take to converge to the solution. This can be shown as the horizontal asymptote reached first from $p = 2$. Noting that the L_2 residual norm is relatively large this is most likely due to the fact that the smoothing values ν_1, ν_2, ν_c are all relatively small compared to the actual smoothing iterations that I used for Jacobi and Gauss-Seidel with $\mathcal{O}(2500)$ iterations. These small smoothing values result in a larger residual but a converged solution nonetheless.

2.4.2 V-Cycle Multigrid Smoothing Iterations

In order to choose an efficient V-cycle setting to create accurate solutions that require lower computing power relative to other solutions will require "*tweaking*" of the smoothing iterations until values are found that result in accurate solutions that require less computing than other alternatives. To relate V-Cycle multigrid to Gauss-Seidel I will be using Work Units to compare the two. The units for "work units" can be defined below in Equation 13,

$$1 \text{ V Cycle} = \sum_{l=0}^{n_{\text{level}}-1} (\nu_1 + \nu_2)\, 2^{-l} \text{ Work Units} \tag{13}$$

After some trial and error and adjusting ν_1, ν_2, ν_c, and the number of iterations plotting V-Cycle multigrid and Gauss-Seidel against their work units arrives to Figure 8 shown below.

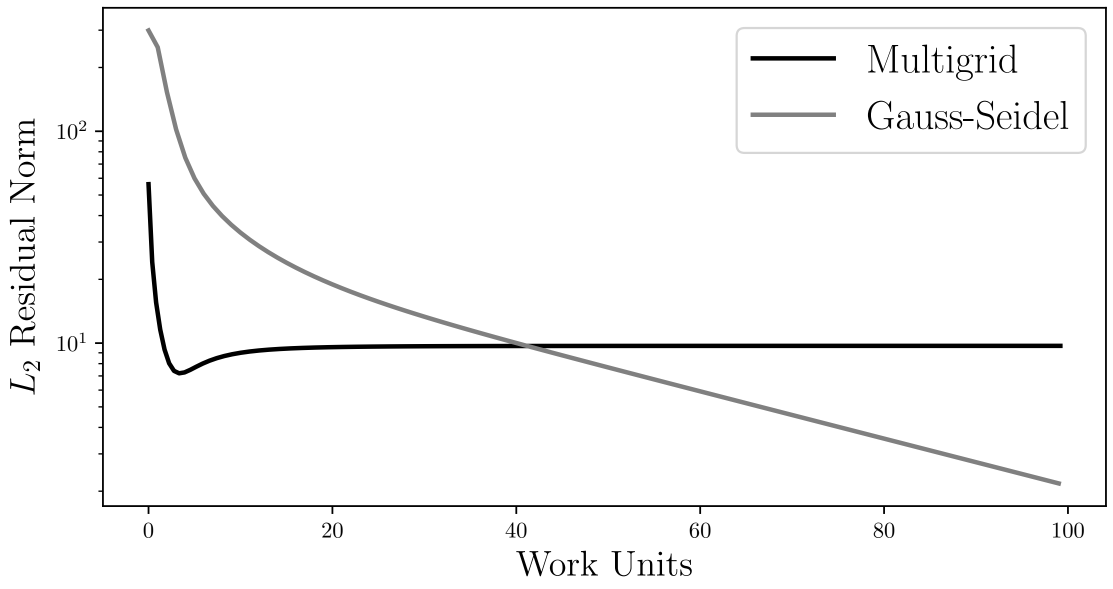


Figure 8: Adjusting V-Cycle multigrid parameters to compare computational costs between V-Cycle multigrid and Gauss-Seidel smoother.

Looking above to Figure 8, I found that the V-cycle was ultimately more accurate for lower work units but at higher work units Gauss-Seidel would become more accurate as V-cycle would start to bring diminishing returns at higher work units. I believe that this is due to the fact that this V-cycle multigrid would converge quickly to its solvers solution whereas the Gauss-Seidel methods solution is more accurate but for better accuracy costs significant more computational power(multigrid converged to its solution at ≈ 10 work units whereas it took Gauss-Seidel ≈ 40 work units). For quick convergence these are the V-Cycle multigrid smoothing parameters that I recommend.

$$\boxed{\mathbf{k = 50}, \quad \boldsymbol{\nu_1 = 10}, \quad \boldsymbol{\nu_2 = 10}, \quad \boldsymbol{\nu_c = 1000}}$$

Appendices

A Python Direct Solver Implementation

Algorithm 1: Driving Code to Implement Direct Solver.

```python
def directsol(p):
    import numpy as np
    from scipy import sparse
    from scipy.sparse import linalg

    N = 2**(p + 3) # P-value scaling
    h = float(2/N) # Step size
    xlin = np.linspace(-1, 1, N+1, endpoint=True) # linspace over domain
    xlin, ylin = np.meshgrid(xlin, np.flip(xlin)) # Meshgrid values

    A = sparse.lil_matrix(((N+1)**2,(N+1)**2)) # Pre-allocate sparse Matrix
    q = np.zeros((N+1)**2)                     # Pre-allocate q vector
    for iy in range(N+1):
        for ix in range(N+1):
            i = iy*(N+1) + ix                  # Iteration index
            iL = i - 1; iR = i + 1             # Left/Right indices
            iD = i - (N+1); iU = i + (N+1)     # Top/Bottom indices

            if ylin[iy,ix] == 1:   # Top Boundary
                q[i] = 1
                A[i,i] = 1
            elif ylin[iy,ix] == -1: # Bottom Boundary
                q[i] = -1
                A[i,i] = 1
            elif abs(xlin[iy,ix]) == 1: # Left/Right Boundary
                q[i] = ylin[iy,ix]
                A[i,i] = 1
            elif abs(xlin[iy,ix]) <= 0.25 and abs(ylin[iy,ix]) <= 0.25: # Interior
                 Boundary
                q[i] = 0
                A[i,i] = 1
            else:   # Interior Domain
                q[i] = 0
                A[i,i] = -4*h**-2
                A[i,iL] = h**-2
                A[i,iR] = h**-2
                A[i,iD] = h**-2
                A[i,iU] = h**-2
    phiv = linalg.spsolve(sparse.csr_matrix(A), q) # Solve for Phi
    phi = np.reshape(phiv, (N+1, N+1))         # Re-shape for plotting

    return xlin, ylin, phi
```

B Coefficient of Pressure Implementation

Algorithm 2: Python Implemention to Approximate Coefficient of Pressure.

```python
def calc_cp(ylin, phi):
    import numpy as np

    h = ylin[0,0] - ylin[1,0]
    num = max(np.shape(ylin)) - 1
    cp = np.zeros(num + 1)

    for i in range(num+1):
        u = (-3/2*phi[num,i] + 2*phi[num-1,i] - 1/2*phi[num-2,i])*h**-1
        cp[i] = 1 - u**2

    return cp

def calc_cl(ylin, cp):
    import numpy as np

    h = ylin[0,0] - ylin[1,0]
    num = max(np.shape(ylin)) - 1
    cl = 0

    for i in range(1, num):
        cl += h*cp[i]
    cl += h/2*(cp[0] + cp[num])
    cl *= 1/2

    return cl
```

C Jacobi Iteration Smoother

Algorithm 3: Python Implemention of Jacboi Iteration Smoother.

```python
def jacobisol(omega):
    import numpy as np
    import math

    p = 3
    N = 2**(p + 3) # P-value scaling
    h = float(2/N) # Step size
    xlin = np.linspace(-1, 1, N+1, endpoint=True) # linspace over domain
    xlin, ylin = np.meshgrid(xlin, np.flip(xlin)) # Meshgrid values

    # Variables for Jacobi Iteration
    resid_mat = np.zeros((N+1, N+1))
    num_iters = 2500
    resid = np.zeros(num_iters)

    # Boundary Conditions / Intial Guess
    U = np.zeros((N+1, N+1))
    U[0,:] = 1; U[N,:] = -1
    U[:,0] = ylin[:,0]; U[:,N] = ylin[:,0]
    u_temp = U.copy()

    for k in range(num_iters):
        for j in range(1, N):
            for i in range(1, N):
                if abs(xlin[j,i]) <= 0.25 and abs(ylin[j,i]) <= 0.25: # Interior
                     Boundary
                    u_temp[j,i] = 0
                    resid_mat[j,i] = 0
                else:   # Interior Domain
                    u_temp[j,i] = (U[j+1, i] + U[j-1, i] + U[j,i+1] + U[j,i-1])/4
                    rij = 0 - h**-2*(4*u_temp[j,i] - 4*U[j,i])
                    resid_mat[j,i] = rij

        U[1:N, 1:N] = U[1:N, 1:N]*(1.-omega) + u_temp[1:N, 1:N]*omega
        for i in range(N+1):
            for j in range(N+1):
                resid[k] += resid_mat[j,i]**2
        resid[k] = math.sqrt(resid[k]/(N+1)**2)

    return resid
```

D Gauss-Seidel Iteration Smoother

Algorithm 4: Python Implemention of Gauss-Seidel Iteration Smoother.

```python
def gausssol(num_iters, p, omega):
    import numpy as np
    import math
    # Variables for Jacobi Iteration
    N = 2**(p + 3) # P-value scaling
    h = float(2/N) # Step size
    xlin = np.linspace(-1, 1, N+1, endpoint=True) # linspace over domain
    xlin, ylin = np.meshgrid(xlin, np.flip(xlin)) # Meshgrid values
    resid_mat = np.zeros((N+1, N+1))
    resid = np.zeros(num_iters)

    # Boundary Conditions / Intial Guess
    U = np.zeros((N+1, N+1))
    U[0,:] = 1; U[N,:] = -1; U[:,0] = ylin[:,0]; U[:,N] = ylin[:,0]
    for k in range(num_iters):
        for j in range(1, N): # Red Nodes
            ------------------------------------------
            if j%2 == 0:
                for i in range(1, N, 2):
                    if abs(xlin[j,i]) <= 0.25 and abs(ylin[j,i]) <= 0.25: # Interior
                        Boundary
                        resid_mat[j,i] = 0
                    else:  # Interior Domain
                        u_unew = (U[j+1, i] + U[j-1, i] + U[j,i+1] + U[j,i-1])/4
                        rij = 0 - h**-2*(4*u_unew - 4*U[j,i])
                        resid_mat[j,i] = rij
                        U[j,i] = U[j,i]*(1.-omega) + u_unew*omega
            else:
                for i in range(2, N-1, 2):
                    if abs(xlin[j,i]) <= 0.25 and abs(ylin[j,i]) <= 0.25: # Interior
                        Boundary
                        resid_mat[j,i] = 0
                    else:  # Interior Domain
                        u_unew = (U[j+1, i] + U[j-1, i] + U[j,i+1] + U[j,i-1])/4
                        rij = 0 - h**-2*(4*u_unew - 4*U[j,i])
                        resid_mat[j,i] = rij
                        U[j,i] = U[j,i]*(1.-omega) + u_unew*omega
        for j in range(1, N): # Black Nodes
            ------------------------------------------
            if j%2 == 0:
                for i in range(2, N-1, 2):
                    if abs(xlin[j,i]) <= 0.25 and abs(ylin[j,i]) <= 0.25: # Interior
                        Boundary
                        resid_mat[j,i] = 0
                    else:  # Interior Domain
                        u_unew = (U[j+1, i] + U[j-1, i] + U[j,i+1] + U[j,i-1])/4
                        rij = 0 - h**-2*(4*u_unew - 4*U[j,i])
                        resid_mat[j,i] = rij
                        U[j,i] = U[j,i]*(1.-omega) + u_unew*omega
            else:
                for i in range(1, N, 2):
                    if abs(xlin[j,i]) <= 0.25 and abs(ylin[j,i]) <= 0.25: # Interior
                        Boundary
                        resid_mat[j,i] = 0
                    else:  # Interior Domain
                        u_unew = (U[j+1, i] + U[j-1, i] + U[j,i+1] + U[j,i-1])/4
                        rij = 0 - h**-2*(4*u_unew - 4*U[j,i])
                        resid_mat[j,i] = rij
                        U[j,i] = U[j,i]*(1.-omega) + u_unew*omega
        for i in range(N+1):
            for j in range(N+1):
                resid[k] += resid_mat[j,i]**2
        resid[k] = math.sqrt(resid[k]/(N+1)**2)
    return resid
```

E V-Cycle Multigrid

Algorithm 5: Python Implemention of V-Cycle Multigrid.

```python
import numpy as np
import math
from direct_sol import directsol

def residual(U, F):
    N = U.shape[0] - 1; h = 2.0/N
    R = U.copy()

    for iy in range(1, N):
        for ix in range(1, N):
            R[iy, ix] = F[iy, ix] - (U[iy+1, ix] + U[iy-1, ix] + U[iy, ix+1] + U[iy,
                ix-1] - 4*U[iy, ix])*h**-2
    return R
def restrict(r, N):
    rc = np.zeros([int(N/2)+1, int(N/2)+1])

    ix = 0; iy = 0
    for j in range(1, N+1, 2):
        for i in range(1, N+1, 2):
            rc[iy,ix] = 1/8.*(r[j+1,i] + r[j-1,i] + r[j,i+1] + r[j,i-1]) + 1/16.*(r[j
                +1,i+1] + r[j-1,i-1] + r[j-1,i+1] + r[j+1,i-1]) + 1/4.*r[j,i]
            ix += 1
            if ix >= int(N/2):
                ix = 0
        iy += 1
    return rc
def prolongate(e2h, N):
    eh = np.zeros([int(2*N)+1, int(2*N)+1])
    xlin = np.linspace(-1, 1, int(2*N)+1, endpoint=True) # linspace over domain
    xlin, ylin = np.meshgrid(xlin, np.flip(xlin)) # Meshgrid values

    for j in range(1, int(2*N), 2):
        for i in range(1, int(2*N), 2):
            if abs(xlin[j,i]) <= 0.25 and abs(ylin[j,i]) <= 0.25: # Interior
                Boundary
                pass
            else:   # Interior Domain
                # Self Weight
                eh[j, i] = e2h[math.floor(j/2.), math.floor(i/2.)]

                # Up/Down Nodes
                eh[j+1, i] += 0.5*e2h[math.floor(j/2.), math.floor(i/2.)]
                eh[j-1, i] += 0.5*e2h[math.floor(j/2.), math.floor(i/2.)]
                eh[j, i+1] += 0.5*e2h[math.floor(j/2.), math.floor(i/2.)]
                eh[j, i-1] += 0.5*e2h[math.floor(j/2.), math.floor(i/2.)]

                # Corner Nodes
                eh[j+1, i+1] += 0.25*e2h[math.floor(j/2.), math.floor(i/2.)]
                eh[j-1, i-1] += 0.25*e2h[math.floor(j/2.), math.floor(i/2.)]
                eh[j-1, i+1] += 0.25*e2h[math.floor(j/2.), math.floor(i/2.)]
                eh[j+1, i-1] += 0.25*e2h[math.floor(j/2.), math.floor(i/2.)]
    return eh

def multigrid(U, F, p, pmax, viter, nu1, nu2, nuc):
    l2err = np.zeros(viter)
    fh = F

    print("V-Cycle␣Method(p=", pmax, ")\n---------------------")
    for k in range(viter):
        print("Iteration:␣", k)

        N = U.shape[0] - 1
        griditer = 0
        fmat = np.zeros([U.shape[0], U.shape[0], pmax-p+1])

        # Sweep Down
        utemp = smooth(U, fh, nu1)
        while N > 2**(p + 3):
            rh = residual(utemp, fh)
            f2h = restrict(rh, N)

            griditer += 1
            N = int(N/2)
            fmat[0:(N+1), 0:(N+1), griditer] = f2h
            utemp = smooth(np.zeros([N+1, N+1]), f2h, nu1)

        # Coarsest Mesh
        utemp = smooth(np.zeros([N+1, N+1]), fmat[0:(N+1), 0:(N+1), griditer], nuc)

        # Sweep Up
        while N <= 2**(pmax + 2):
            utemp = prolongate(utemp, N)

            N = int(2*N)
            griditer -= 1
            utemp = smooth(utemp, fmat[0:(N+1), 0:(N+1), griditer], nu2)

        U += utemp
        resid = residual(U, F)

        for j in range(N + 1):
            for i in range(N + 1):
                l2err[k] += resid[j,i]**2
        l2err[k] = np.sqrt(l2err[k]/(N + 1)**2)

    return U, l2err
def vcyclesol(p, pmax, viter, nu1, nu2, nuc):
    N = 2**(pmax + 3)
    U = np.zeros([N+1, N+1])
    F = U.copy()
    U[:, 0] = np.flip(np.linspace(-1, 1, N+1))
    U[:, N] = np.flip(np.linspace(-1, 1, N+1))
    U[0,:] = 1; U[N,:] = -1

    U, l2err = multigrid(U, F, p, pmax, viter, nu1, nu2, nuc)

    return U, l2err
def smooth(U, F, nu):
    N = U.shape[0]-1; h = 2.0/N
    omega = 1.5

    xlin = np.linspace(-1, 1, N+1, endpoint=True) # linspace over domain
    xlin, ylin = np.meshgrid(xlin, np.flip(xlin)) # Meshgrid values
    for k in range(nu):
        for iy in range(1, N): # Red Nodes --------------------------------------------
            if iy%2 == 0:
                for ix in range(1, N, 2):
                    if abs(xlin[iy,ix]) <= 0.25 and abs(ylin[iy,ix]) <= 0.25: # Interior Boundary
                        U[iy, ix] = 0
                    else:  # Interior Domain
                        unew = 0.25*(U[iy+1, ix] + U[iy-1, ix] + U[iy, ix-1] + U[iy, ix+1] - F[iy, ix]*h**2)
                        U[iy, ix] = U[iy, ix]*(1.0 - omega) + unew*omega
            else:
                for ix in range(2, N-1, 2):
                    if abs(xlin[iy,ix]) <= 0.25 and abs(ylin[iy,ix]) <= 0.25: # Interior Boundary
                        U[iy, ix] = 0
                    else:  # Interior Domain
                        unew = 0.25*(U[iy+1, ix] + U[iy-1, ix] + U[iy, ix-1] + U[iy, ix+1] - F[iy, ix]*h**2)
                        U[iy, ix] = U[iy, ix]*(1.0 - omega) + unew*omega
        for iy in range(1, N): # Black Nodes --------------------------------------------
            if iy%2 == 0:
                for ix in range(2, N-1, 2):
                    if abs(xlin[iy,ix]) <= 0.25 and abs(ylin[iy,ix]) <= 0.25: # Interior Boundary
                        U[iy, ix] = 0
                    else:   # Interior Domain
                        unew = 0.25*(U[iy+1, ix] + U[iy-1, ix] + U[iy, ix-1] + U[iy, ix+1] - F[iy, ix]*h**2)
                        U[iy, ix] = U[iy, ix]*(1.0 - omega) + unew*omega
            else:
                for ix in range(1, N, 2):
                    if abs(xlin[iy,ix]) <= 0.25 and abs(ylin[iy,ix]) <= 0.25: # Interior Boundary
                        U[iy, ix] = 0
                    else:   # Interior Domain
                        unew = 0.25*(U[iy+1, ix] + U[iy-1, ix] + U[iy, ix-1] + U[iy, ix+1] - F[iy, ix]*h**2)
                        U[iy, ix] = U[iy, ix]*(1.0 - omega) + unew*omega
    return U
```

F Main Python Driving Code

Algorithm 6: Python Implemention of V-Cycle Multigrid.

```
from matplotlib import pyplot as plt
from matplotlib.ticker import MaxNLocator
import numpy as np
import math
from direct_sol import directsol
from calc_coefs import calc_cp, calc_cl
from jacobi_sol import jacobisol
from gauss_sol import gausssol
from vcycle_sol import vcyclesol

plt.rc('text', usetex=True)
plt.rc('font', family='serif')

def export_phi(phi):
    f = open('9by9_mat',"w") # Filename
    output = ''
    for j in range(9):
        for i in range(9):
            output += str.format('{0:.4f}',phi[j,i]) + r'&␣'
        if i == 8:
            output += r'\\' # Output results to LaTeX environment
    f.write(output)
    f.close()

def export_cl(cl):
    f = open('cl_vals',"w") # Filename
    output = ''
    for i in range(6):
        output += r'$p␣=␣$␣' + str.format('{0:.0f}',i)+ r'&␣' + str.format('{0:.7f}'
            ,cl[i]) + r'\\'
    f.write(output)
    f.close()

def gen_grids(p):
    N = 2**(p + 3) # P-value scaling
    xlin = np.linspace(-1, 1, N+1, endpoint=True) # linspace over domain
    xlin, ylin = np.meshgrid(xlin, np.flip(xlin)) # Meshgrid values

    return xlin, ylin

def run_q1():
    xlin, ylin, phi = directsol(0)
    export_phi(phi)

def run_q2():
    plt.figure(figsize=(8,4))
    for i in np.array([0, 2, 4]):
        xlin, ylin, phi = directsol(i)
        cp = calc_cp(ylin, phi)

        plot_label = r'$p$␣=␣' + str(i)
        plt.plot(xlin[0,:], -cp, lw = 2, label = plot_label)
    plt.xlabel(r'Location␣along␣bottom␣wall', fontsize = 16)
    plt.ylabel(r'$-c_p(x)$', fontsize = 16)
    plt.legend(fontsize = 18)
    plt.savefig('figs/cp_runs.pdf', bbox_inches = 'tight')
    plt.show()

    xlin, ylin, phi = directsol(7)
    cl_exact = calc_cl(ylin, calc_cp(ylin, phi))
    f = open('cl_exact',"w")
    output = str.format('{0:.7f}',cl_exact)
    f.write(output)
    f.close()

    cl_vals = np.zeros(6); h_vals = np.zeros(6); cl_err = np.zeros(6)
```

```python
    for i in range(6):
        xlin, ylin, phi = directsol(i)
        cl = calc_cl(ylin, calc_cp(ylin, phi))
        cl_vals[i] = cl
        h_vals[i] = ylin[0,0] - ylin[1,0]
        cl_err[i] = abs(cl - cl_exact)
    export_cl(cl_vals)

    rate = math.log10(cl_err[4]/cl_err[5])/math.log10(h_vals[4]/h_vals[5])
    plot_label = r'Convergence␣=␣$\mathcal{O}($' + str.format('{0:.4f}',rate) + R')'
    plt.figure(figsize=(8,4))
    plt.plot(range(6), cl_err, color = 'black', marker = 'o', lw = 2, label = plot_label)
    plt.xlabel(r'$p$␣values', fontsize = 16)
    plt.ylabel(r'Error,␣$||␣c_l(p)␣-␣c_{l,␣exact}||$', fontsize = 16)
    plt.yscale('log')
    plt.legend(fontsize = 18)
    plt.savefig('figs/cl_err.pdf', bbox_inches = 'tight')
    plt.show()

def run_q3():
    plt.figure(figsize=(8,4))
    plt.plot(range(2500), jacobisol(0.3), color = 'black', lw = 2, label = r'$\omega␣=␣0.3$'); print('Omega␣=␣0.3␣-␣Done')
    plt.plot(range(2500), jacobisol(0.6), color = 'blue', lw = 2, label = r'$\omega␣=␣0.6$'); print('Omega␣=␣0.6␣-␣Done')
    plt.plot(range(2500), jacobisol(1.0), color = 'gray', lw = 2, label = r'$\omega␣=␣1.0$'); print('Omega␣=␣1.0␣-␣Done')
    plt.xlabel(r'Iteration␣Number', fontsize = 16)
    plt.ylabel(r'$L_2$␣Residual␣Norm␣Error', fontsize = 16)
    plt.yscale('log')
    plt.legend(fontsize = 18)
    plt.savefig('figs/jacobi_l2.pdf', bbox_inches = 'tight')
    plt.show()

    plt.figure(figsize=(8,4))
    plt.plot(range(2500), gausssol(2500, 3, 0.5), color = 'black', lw = 2, label = r'$\omega␣=␣0.5$'); print('Omega␣=␣0.5␣-␣Done')
    plt.plot(range(2500), gausssol(2500, 3, 1.0), color = 'blue', lw = 2, label = r'$\omega␣=␣1.0$'); print('Omega␣=␣1.0␣-␣Done')
    plt.plot(range(2500), gausssol(2500, 3, 1.5), color = 'gray', lw = 2, label = r'$\omega␣=␣1.5$'); print('Omega␣=␣1.5␣-␣Done')
    plt.xlabel(r'Iteration␣Number', fontsize = 16)
    plt.ylabel(r'$L_2$␣Residual␣Norm␣Error', fontsize = 16)
    plt.yscale('log')
    plt.legend(fontsize = 18)
    plt.savefig('figs/gauss_l2.pdf', bbox_inches = 'tight')
    plt.show()

def run_q4():
    p = 1; viter = 25
    ax = plt.figure(figsize=(10,5)).gca()
    ax.xaxis.set_major_locator(MaxNLocator(integer=True))
    for pval in range(2, 6):
        U, l2_2 = vcyclesol(p, pval, viter, 2, 2, 50)
        plotlabel = r'$p$␣=␣' + str(pval)
        plt.plot(range(1, viter+1), l2_2, lw = 2, label = plotlabel)
    plt.xlabel(r'Multi-grid␣Cycles', fontsize = 16)
    plt.ylabel(r'$L_2$␣Residual␣Norm', fontsize = 16)
    plt.yscale('log')
    plt.legend(fontsize = 18, ncol = 4)
    plt.savefig('figs/vcyc_l2_err.pdf', bbox_inches = 'tight')
    plt.show()


    p = 0; pmax = 4; viter = 50
    nu1 = 10; nu2 = 10; nuc = 1000
    U, resid_norm = vcyclesol(p, pmax, viter, nu1, nu2, nuc)
    workunits = np.zeros(viter)
    for l in range(viter-1):
        workunits[l+1] = workunits[l] + 2**(pmax*l/viter + 3)/(nu1 + nu2)

    plt.figure(figsize=(8,4))
    plt.plot(workunits, resid_norm, color = 'black', lw = 2, label = r'Multigrid')
    plt.plot(range(100), gausssol(100, 3, 1.5), color = 'gray', lw = 2, label = r'
        Gauss-Seidel')
    plt.xlabel(r'Work␣Units', fontsize = 16)
    plt.ylabel(r'$L_2$␣Residual␣Norm', fontsize = 16)
    plt.yscale('log')
    plt.legend(fontsize = 18)
    plt.savefig('figs/vcyc_gs.pdf', bbox_inches = 'tight')
    plt.show()
def main():
    run_q1()
    run_q2()
    run_q3()
    run_q4()
if __name__ == "__main__":
    main()
```