EN 530.766
Fall 2023
Final Project–Haobo Zhao

March 5, 2024

Abstract

This report is an approach to apply finite difference method to complex domain and boundary condition, design grid generator and SOR Gauss-Seidel solver and line SOR solver, compare convergence and iteration performance.

The solver could be divided by three major parts: Grid Generator, Iteration Solver, and Reuslt Analysist. Grid Genenerator is aiming to create domain, generate grids, and classify different points. Iteration Solver is using iteration information, using deployed formula, to update grid each iteration. Result Analysist is for integrate result and output useful result and basic analysis.

Then, we deployed our solver to solve different size grids, the result shows, the finier the grid as, convergence rate become more slower, and CPU time is larger.

To check the result correctness of the solution, we obtained exact solution for another domain, and use the value of that domain at our domain's boundary, as the new boundary condition we have, to check the correctness of our solution. It shows less error than we expected.

We also checked the overrelaxation parameter performance, found as the finier the grid is, the optimal value moves toward to 2.

We also deployed our solver to 4-cylinder domain, the result is similiar, expected the convergence rate for each grid become much slower as the boundary become more complex.

Deployed line SOR, the convergence rate is much faster than Point Gauss-Seidel.

Finally, to study potential flow pass cylinder, we change the boundary condition to Von-Neuman boundary condition, and calculated the potential flow on large domain.

Contents

1 Review of Project Description

Consider the 2-D Laplace equation on the domain below with $u = 0$ on the outer boundary and the following internal boundary with $u = 1$:

$$u_{xx} + u_{yy} = 0 \tag{1}$$

(a) Use a second-order central-difference scheme to develop an iterative solver to solve the above problem. Use the stair-step method to implement the boundary conditions on the internal boundary. Use point Gauss-Seidel (GS) with Successive Over-Relaxation (SOR). Follow the algorithm shown in class.

(b) Draw a flowchart of your solver. [10 points]

(c) Solve the problem on grids with 32×32, 96×96, 160×160 grid cells, and a 224×224 grid also. Use a random number generator to generate the starting guess for the field in the range

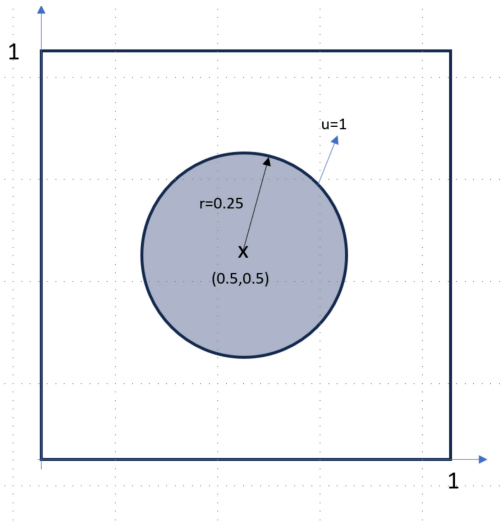


Figure 1: The domain with boundary conditions.

of $[-1, +1]$. Plot the residual versus iteration index and the contour plots of the converged solution for each of the grids for overrelaxation parameter of 1.0. [140 Points]

(d) How can you check the correctness of your solution? Are there some symmetry lines in the solution where you can obtain an exact solution and compare it to the numerical solution? Is there a different boundary condition you could use on the outer boundary for which you could obtain an exact solution and then compare your numerical solution? [25 points]

(e) Discuss the observed performance of iterative solver for each grid simulated here. Consider questions like:

1. How does the convergence change with overrelaxation parameter? What is the optimal value of this parameter? Does this optimal value change with grid size? [25 points]
2. How does the CPU time increase with number of grid points? Is the expected trend observed? [25 points]
3. Does your solution exhibit the expected second-order accuracy? [50 points]

(f) Repeat (e) for the configuration below. Also comment on how the convergence and CPU time is different from that for the previous configuration. [100 Points]

(g) Solve the second configuration with Line-SOR and compare the convergence rate and CPU-time performance with point SOR. [75 points]

(h) Challenge problem for extra points – Modify the solver to compute the potential flow past the 4-cylinder configuration in a much larger computational domain where you can impose free-stream conditions at the outer boundary. Plot streamlines and pressure contours of the flow. [50 points]

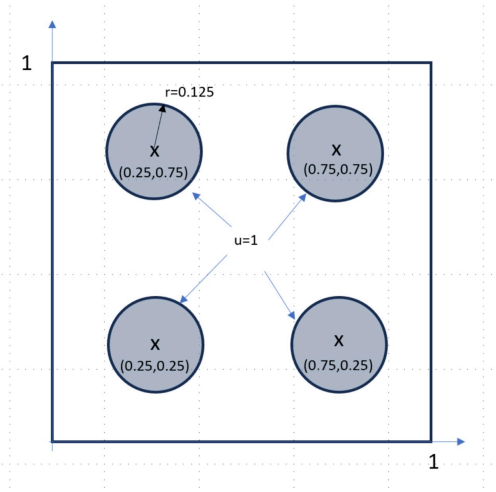


Figure 2: The domain with New boundary conditions.

2 (a) Iterative GS-SOR Solver

2.0.1 Grid Generator

The first step to build a solver is to bulid is Grid Generator, whatever its boundary or platforms are. Grid generstor's responsibility is to set platform with boundary, which means it needs to:

(1) Create grid
(2) identify points apply boundary condition
(3) identify points affected by boundary points directly (we call these Effected Points (EP))

According to our problem, identify part (2) need been done by ourself. However, we need to solve inner boundary first.

Thus, we can divide our Grid Generator to three main functional Creator: Grid-Creator, Inner Boundary Counter (IBcboundary), and Inner Boundary Conditionor (IBconditionor).

2.0.1.1 Grid-Creator

For Grid-Creator, we just using the size of platform, and grid cell size.

2.0.1.2 Inner Boundary Counter (IBcboundary)

For Inner Boundary Counter, the idea is using the formula of inner Boundary, firstly to find the inner boundary limit in x and y direction.

In the limit, by using Stair-Step Method, for the real boundary intersected whith horizontal line j (vertical coordinate), we can identify the i (horizontal coordinate) for each Effective points.

Thus, we can find i_{low} and i_{high} for each j.
Similiar, we can find j_{low} and j_{high} for each i.

The simplest EPs can been divided by 4 kinds:

Boundary intersected with horizontal line(j line) on (i_{low}[j],j)

Boundary intersected with horizontal line(j line) on (i_{high}[j],j)
Boundary intersected with vertical line(i line) on (i,j_{low}[i])
Boundary intersected with vertical line(i line) on (i,j_{high}[i])

However, if just use these information, is hard to apply our control equation for th EPs, especially the formulas are different.

That's why we are going to transfor our information to Inner Boundary Classifier (IBclassfiier), let it help us classify different kinds of EP points.

2.0.1.3 Inner Boundary Classifier (IBclassifier)

We already get columns like i_{low}[j], this classifier is going to generate conditions for each kind of EP:

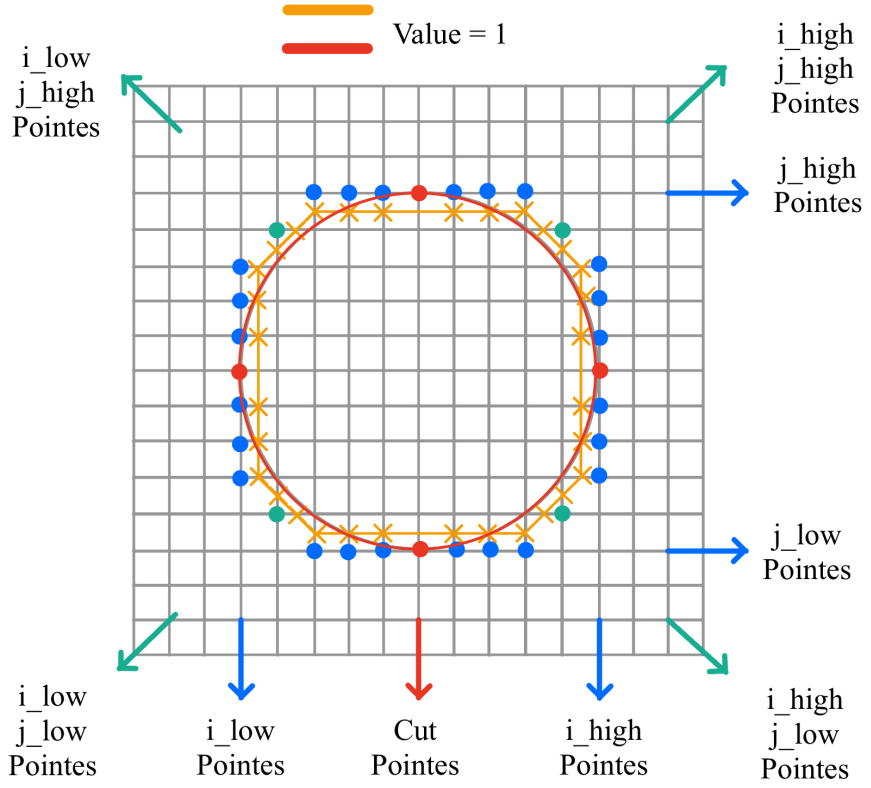


Figure 3: Classify Different kinds of Effected Points

In this diagram, the Red line is circle boundary, while it cut through grids, we use the orange line at each half grid as its step boundary, where is value equal to 1.

However, when we employ this method in to solver, what we calculate is not the orange lines directly, we calculate the bule and green points, applying boundary modified control equation to these points, to replace the actual boundary effect. These points we called Effected Points. The Effected Points could be classified into 9 groups:

Blue:
(1) i_{low}: circle boundary only cuts grid at x-low lines.

(2) i_{high}: circle boundary only cuts grid at x-high lines.
(3) j_{low}: circle boundary only cuts grid at y-low lines.
(4) j_{high}: circle boundary only cuts grid at y-high lines.

Green:
(5) $i_{low\_j\_low}$: circle boundary cuts grid at x-low lines and y-low lines.
(6) $i_{low\_j\_high}$: circle boundary cuts grid at x-low lines and y-high lines.
(7) $i_{high\_j\_low}$: circle boundary cuts grid at x-high lines and y-low lines.
(8) $i_{high\_j\_high}$: circle boundary cuts grid at x-high lines and y-high lines.

Red:
(9) Cut Points: Where circle cut grid at grid points, Value=1

The iteration formula for these special points is showing below:

For i_{low}, we have:

$$U_{xx} = \frac{4}{3\Delta^2}U_{i-1} - \frac{4}{\Delta^2}U_i + \frac{8}{3\Delta^2}$$

Then, the iteration formula for i_{low} is:

$$U_i = \frac{1}{6}\left[\frac{4}{3}U_{i-1,j} + \frac{8}{3} + U_{i,j-1} + U_{i,j+1}\right]$$

For i_{high}, we have:

$$U_{xx} = \frac{4}{3\Delta^2}U_{i+1} - \frac{4}{\Delta^2}U_i + \frac{8}{3\Delta^2}$$

$$U_i = \frac{1}{6}\left[\frac{4}{3}U_{i+1,j} + \frac{8}{3} + U_{i,j-1} + U_{i,j+1}\right]$$

Similarly, for For j_{low}::

$$U_i = \frac{1}{6}\left[\frac{4}{3}U_{i,j-1} + \frac{8}{3} + U_{i-1,j} + U_{i+1,j}\right]$$

Similarly, for For j_{high}::

$$U_i = \frac{1}{6}\left[\frac{4}{3}U_{i,j+1} + \frac{8}{3} + U_{i-1,j} + U_{i+1,j}\right]$$

Similarly, for $i_{low\_j\_low}$:

$$U_i = \frac{1}{8}\left[\frac{4}{3}U_{i-1,j} + \frac{4}{3}U_{i,j-1} + \frac{16}{3}\right]$$

Similarly, for $i_{low\_j\_high}$:

$$U_i = \frac{1}{8}\left[\frac{4}{3}U_{i-1,j} + \frac{4}{3}U_{i,j+1} + \frac{16}{3}\right]$$

Similarly, for $i_{high\_j\_low}$:

$$U_i = \frac{1}{8}\left[\frac{4}{3}U_{i+1,j} + \frac{4}{3}U_{i,j-1} + \frac{16}{3}\right]$$

Similarly, for $i_{high\_j\_high}$:

$$U_i = \frac{1}{8}\left[\frac{4}{3}U_{i+1,j} + \frac{4}{3}U_{i,j+1} + \frac{16}{3}\right]$$

2.0.2 Iteration Formula

For General Point's calculation, the formula is showing below:

$$(u_{xx} + u_{yy}) = 0 \quad \text{for } 0 < x, y < 2\pi$$

Or in other notation:

$$\nabla^2 P = 0$$

Add sudo-time part, obtain:

$$\nabla^2 P = \frac{\partial P}{\partial t}$$

Where is also can be shown as:

$$(u_{xx} + u_{yy}) = u_t$$

Transfer Partial Difference Equartion (PDE) to Finite Difference equation (FDE), use central difference method for spacial derivative transform, and use forward difference for "time" transformk, obtained:

$$\frac{P_{ij}^{k+1} - P_{ij}^{k}}{\Delta t} = \frac{1}{\Delta^2}\left[P_{i-1,j}^{k} + P_{i+1,j}^{k} + P_{i,j-1}^{k} + P_{i,j+1}^{k} - 4P_{ij}^{k}\right]$$

For stability, in 1-D scheme, r =<1/2, in 2-D scheme, r =< 1/4. To obatin the biggest dt, let r= 1/4, which is $\frac{\Delta t}{\Delta^2} = \frac{1}{4}$, where the FDE transfer to:

$$P_{ij}^{k+1} = \frac{1}{4}\left[P_{i-1,j}^{k} + P_{i+1,j}^{k} + P_{i,j-1}^{k} + P_{i,j+1}^{k}\right]$$

The formula shown above is Jacobi iteration method. For Gauss Seidel method, the equation is showing below:

$$P_{ij}^{k+1} = \frac{1}{4}\left[P_{i-1,j}^{k+1} + P_{i+1,j}^{k} + P_{i,j-1}^{k+1} + P_{i,j+1}^{k}\right]$$

For residual, it could be calculated by compare our simulation result with the source term, which is 0 in this scenario:

$$r^k = \frac{1}{\Delta^2}\left[P_{i-1,j}^{k} + P_{i+1,j}^{k} + P_{i,j-1}^{k} + P_{i,j+1}^{k} - 4P_{ij}^{k}\right]$$

2.1 Grid Generator

The Grid Genersator's function is already been explained on the subsection before, the flow chart of Grid Generator is showing below:

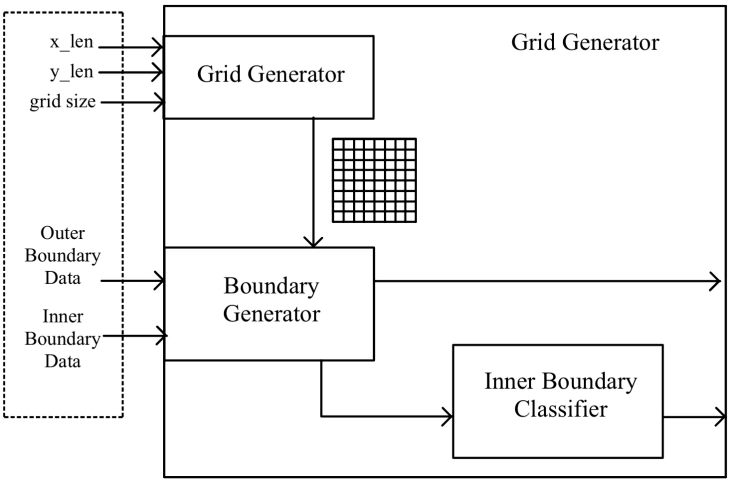


Figure 4: Grid Generator

2.2 Iteration Solver

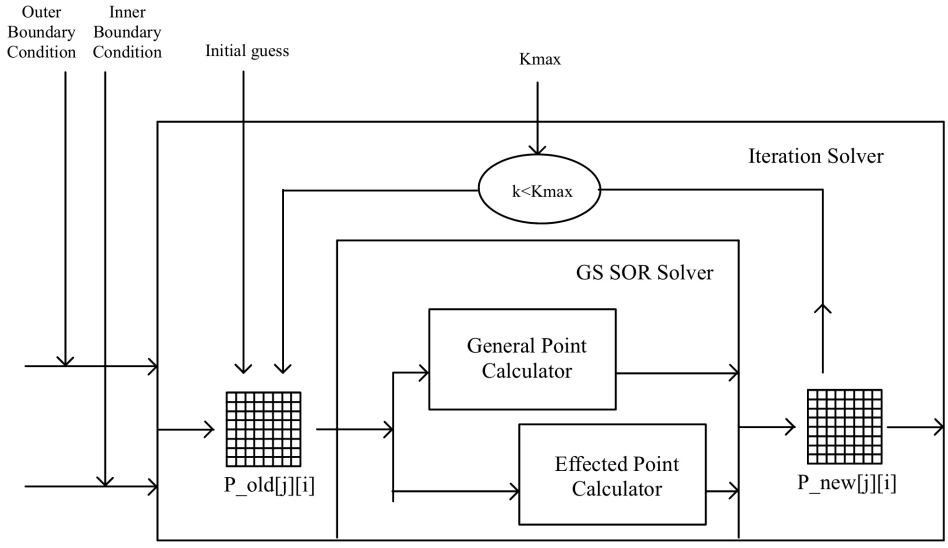


Figure 5: Iteration Solver

The Sover diagram is showing above. For iteration Solver, we already classified our points to General Points and Effected Points.

For General Points, the algorithm is showing below:

For Effected Points, the algorithm is showing below:

C is the output of Inner Boundary Classifier, which value is 1 if the point we are using belong its class, the value become 0 if the point is not belong this class.

Algorithm 1 Point Gauss-Seidel SOR Calculator

procedure POINT_GSSSOR($P_{\text{in}}, j_{\text{low}}, j_{\text{high}}, i_{\text{low}}, i_{\text{high}}, \omega$)
  $P_{\text{out}} \leftarrow \text{deepcopy}(P_{\text{in}})$
  for $j \leftarrow j_{\text{low}}$ to j_{high} **do**
    for $i \leftarrow i_{\text{low}}$ to i_{high} **do**
      $P_{\text{out}}[j][i] \leftarrow \frac{1}{4} \times (P_{\text{out}}[j][i-1] + P_{\text{out}}[j-1][i] + P_{\text{in}}[j][i+1] + P_{\text{in}}[j+1][i])$
      $P_{\text{out}}[j][i] \leftarrow \omega \times P_{\text{out}}[j][i] + (1-\omega) \times P_{\text{in}}[j][i]$
    end for
  end for
  return P_{out}
end procedure

Algorithm 2 Point Calculator for Effected Point using Gauss-Seidel SOR Method

procedure IBGSSOR($P_{\text{out}}, i, j, \omega, C_{\text{inner}}, C_{\text{total}}, C_{\text{xbl}}, C_{\text{ybl}}, C_{\text{ybh}}, C_{\text{edge}}, C_{\text{xyl}}, C_{\text{xyh}}, C_{\text{xyh}}$)
  $P_{\text{temp}} \leftarrow \frac{1}{4} (P_{\text{out}}[i-1][j] + P_{\text{out}}[i][j-1] + P_{\text{in}}[i+1][j] + P_{\text{in}}[i][j+1])$
  $P_{\text{corr}} \leftarrow C_{\text{xbl}}[i][j] \times P_{\text{out}}[i-1][j] + C_{\text{ybl}}[i][j] \times P_{\text{out}}[i][j-1] + C_{\text{ybh}}[i][j] \times P_{\text{out}}[i+1][j] + C_{\text{edge}}[i][j] \times P_{\text{out}}[i][j+1]$
  $P_{\text{new}} \leftarrow (1-\omega) \times P_{\text{out}}[i][j] + \omega \times P_{\text{temp}} - \omega \times P_{\text{corr}}$
  $P_{\text{out}}[i][j] \leftarrow P_{\text{new}}$
  return $P_{\text{out}}[i][j]$
end procedure

2.3 Result Abnalysist

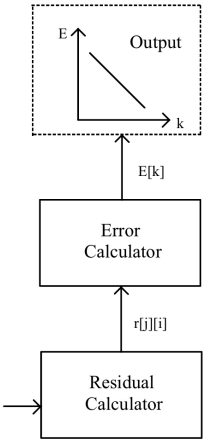


Figure 6: Result Analysist

The result analysist is going to capture each step's result, and calculate its resuidual, then add the absolute resudial value at each point, find out the Error. Then, it could plot the relationship of Error with iteration step.

3 (b) Solver General Strture

The whole solver diagram is showing below:

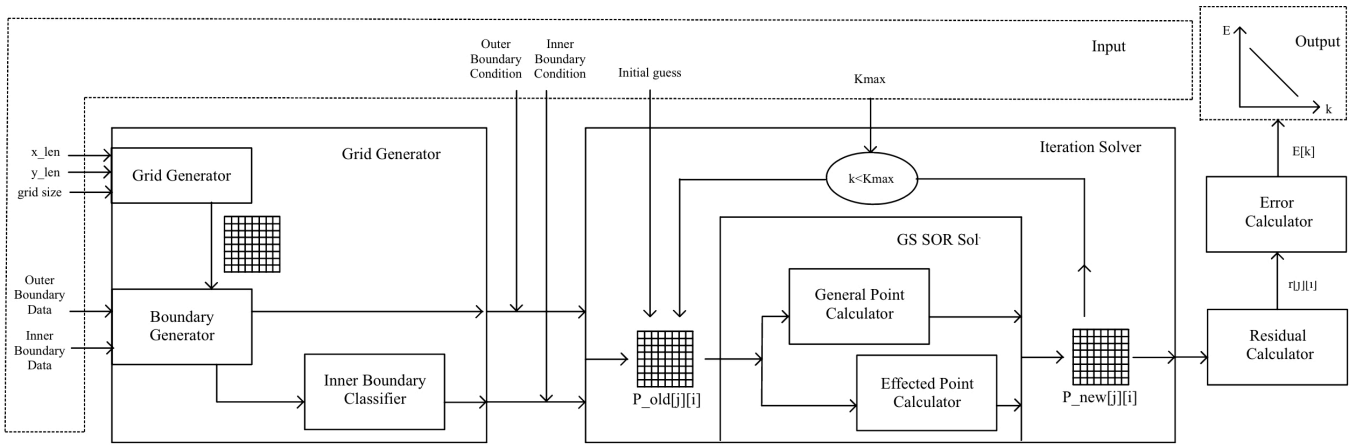


Figure 7: The Solver Diagram

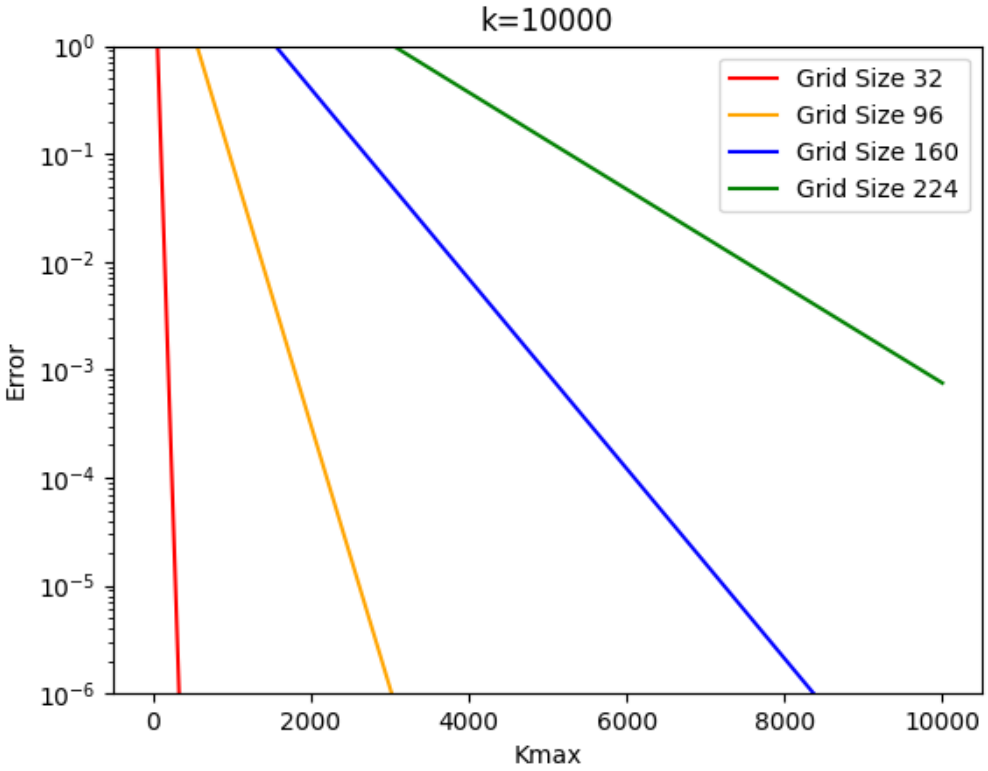
As we can see, the Grid Geenrator generate grids and classify points to General Points (GP) and Effected Points (GP, though it also need to classify EP again), then pass their data to Iteration Solver. The GS SOR Solver use their classification to calculate each points, and pass their new value back and calculate them again, until iteration number k reach Kmax. At each iteration step, Iteration Solver pass the data to Result Analysist, and let it to calculate the residual of our Finite Difference Equation with Partial Difference Equation.

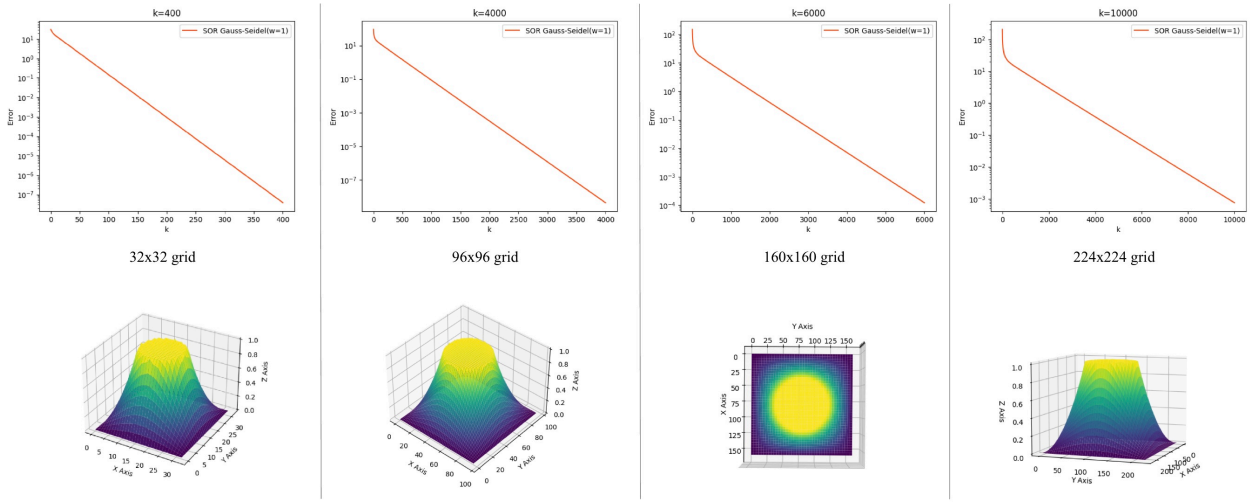
4 (c) Iteration for different grid sizes

Deploy our solver to solve this problem under control 2nd Laplace equation and boundary condition, the result is showing below:

4.1 Result Compare for each grid size

Apply our solver to 32x32, 96x96, 160x160, and 224x224 grid size, the result is showing below:





32x32 grid 96x96 grid 160x160 grid 224x224 grid

It could be seen although in different size grid, the results are similiar as its convergence trends are almost same (except the convergnece rate), which keeping decreasing almost like stright line at semi-log plot. Their final temperature distribution are similiar, which could be seen at secnd row of the diagram, that the highest temperature appear at the inner boundary, and show non-linear decrease characteristic at its path reach the boundary.

4.2 Result analysis

4.2.1 Need more iteration steps with fine grid

For the result, we can see that as we are applying more fine grid, the iteration steps we need is going to below up.

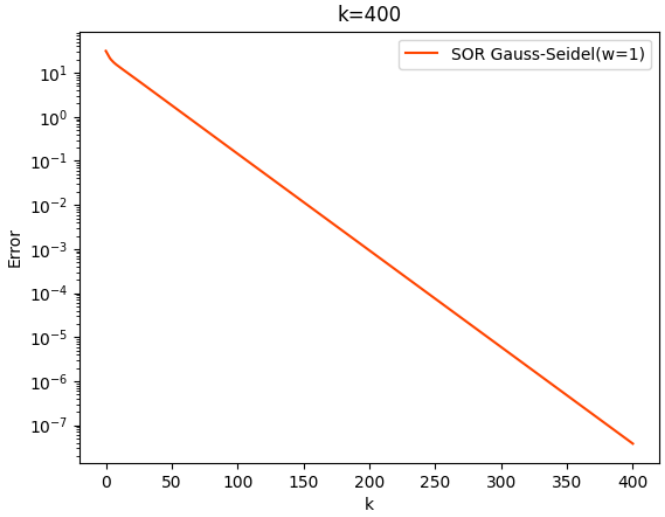
For 32x32 grid, it only need 200 iteration steps to obtain 10^{-3} accuracy, while it needs 2000 iteration steps for 96x96 gird, and for 160x160 grid, is 4000 steps, for 224 grid, is 10000 steps to reach 10^{-3} accuracy.

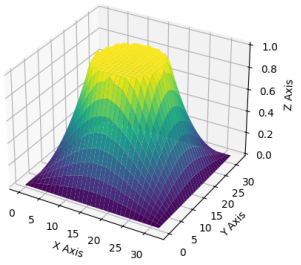
4.2.2 Temperature distribution as expected

We obtained two 3-D result for 32x32 grid at k=400, and also for 96x96 grid at k=4000, the result are similiar, fullfill our expection: the temperature is high around inner boundary, and decrease to zero as it reach outer boundary, the distribution at this boundary condition is not linear at x or y direction.

4.3 Detialed Result Show for each grid size

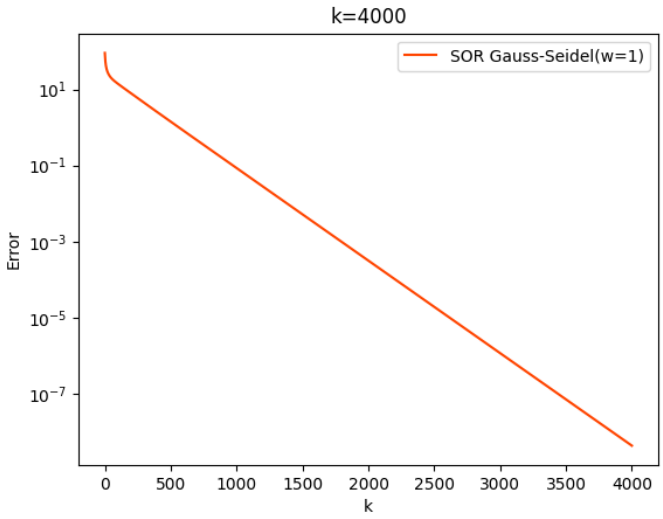
4.3.1 32x32 grid size

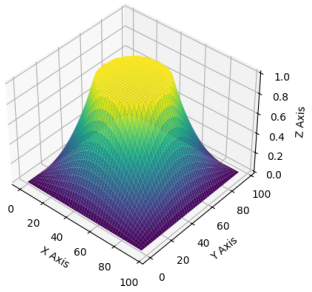




For 32x32 size grid, it could be seen the error is decrasing fast at the semi-log plot, at iteration steps reach to 400, the error is already decrease to 10^{-7}

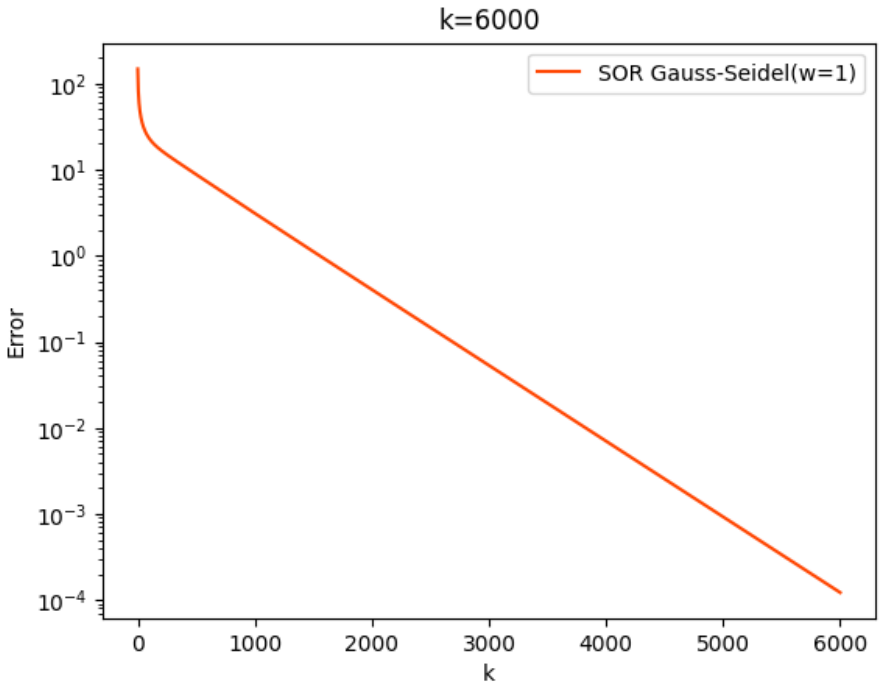
4.3.2 96x96 grid size





For 96x96 grid size reuslt, we can see it is similiar to 32x32 sized grid. However, its convergence much slower.

4.3.3 160x160 grid size



We can see very similiar result with pervious results, that the convergence rate now is also slower than 96x96 grid: the error only reach 10^{-4} as the iteration steps already got to 6000.

The contour plot of 160x160 grid is showing below:

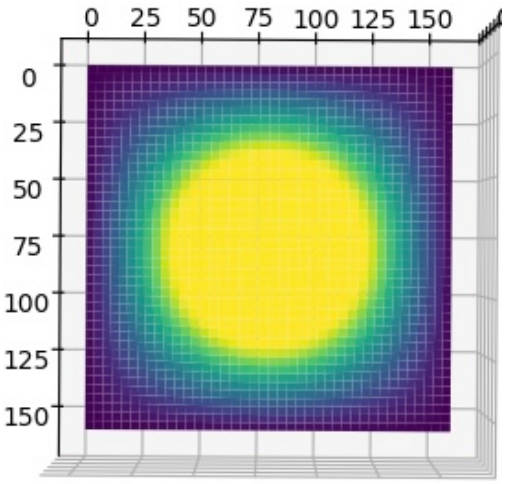
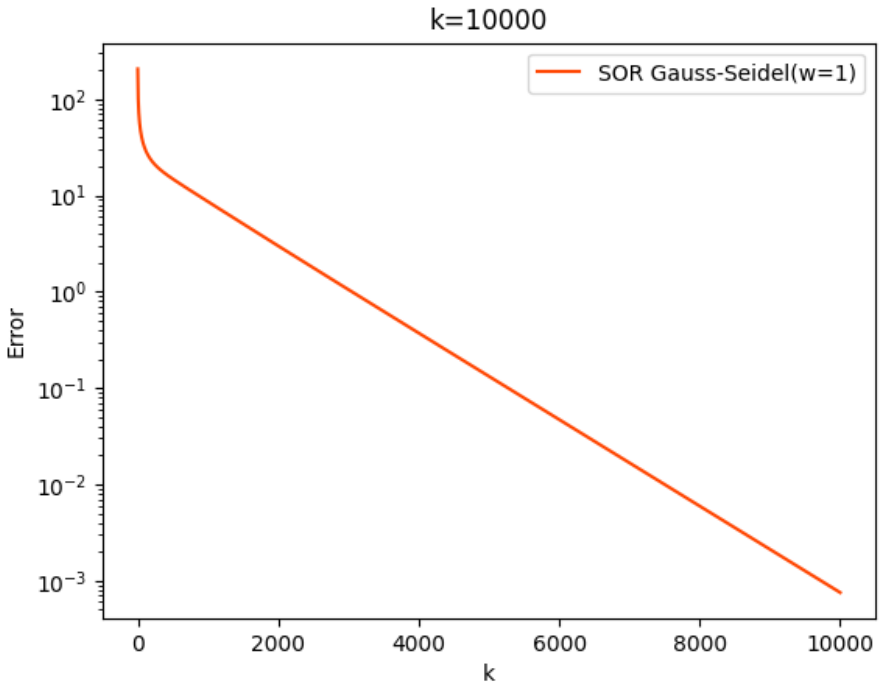


Figure 8: Contour plot of 160x160 grid

4.3.4 224x224 grid size



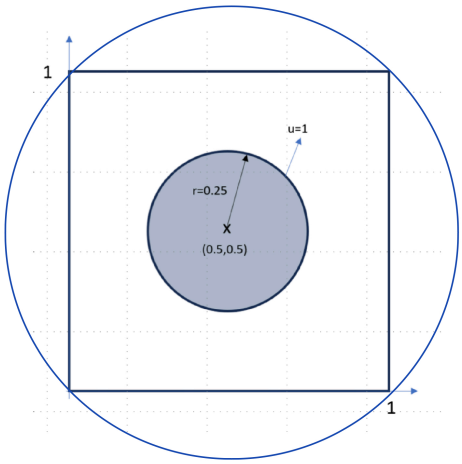
For 224x224 grid, similiarly, it need 10000 iteration steps to let the error reach 10^{-3}.

5 (d) Check correctness of solution

To check the correctness of the solution, it is obvious we cannot find symmetric lines at this boundary condition. Thus, are going to obtain exact solution for another boundary condition where we can implement to our solve domain.

5.1 Exact solution for circular outer boundary.

The most simplest we we can see to obtain an exactsolution is to change the domain from the square to a lrge circle, where the old square domain's four edges at this circle. Let this big circle be the outer domain edge, and let outer boundary codition to u=0, we can obtain exact solution.



5.1.1 Exact solution

Transfer control equation to polar coordinate:

$$\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial \Phi}{\partial r}\right)+\frac{1}{r^2}\frac{\partial^2 \Phi}{\partial \theta^2}=0 \tag{2}$$

By symmetric, we know

$$\frac{\partial^2 \Phi}{\partial \theta^2}=0$$

Thus, the equation becomes

$$\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial \Phi}{\partial r}\right)=0 \tag{3}$$

The solution of the equation is showing below:

$$\Phi = C\ln r + C_2 \tag{4}$$

Apply boundary condition:

$$\Phi\Big|_{r=R_1}=1: \quad 1=C\ln R_1+C_2$$
$$\Phi\Big|_{r=R_2}=0: \quad 0=C\ln R_2+C_2$$

The result is showing below:

$$\Phi=\frac{\ln(r/R_2)}{\ln(R_1/R_2)} \tag{5}$$

Where R_1 is cylinder radius (=0.25), R_2 is outer domain boundary radius (=$\sqrt{2}$).

5.1.2 Apply to numerical boundary

The second step is to use our exact solution at our square boundary to change the old boubdary condition. As we know, the coordinate transfer at the boubdary:

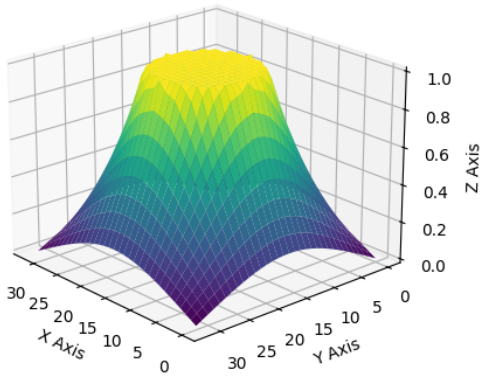
$$r = \sqrt{\left(\frac{a}{2}\right)^2 + (a-x)^2}$$

Then, we could obtain our new boundary comdition:

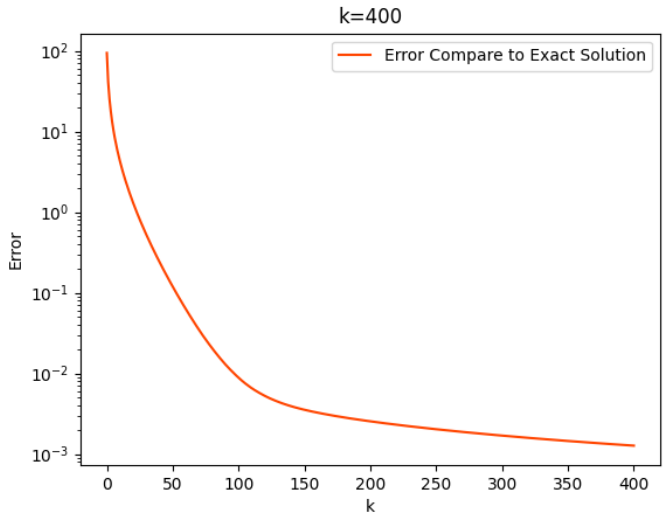
$$\Phi_{BC} = -\frac{\ln\left(\frac{a^2}{4} - ax + x^2\right)}{5\ln 2}$$

$$a = 1: \quad \Phi_{BC} = -\frac{\ln\left(x^2 - x + 1\right)}{5\ln 2}$$

By applying this 'exact' boundary condition to our new outer boundary contition, the result form our solver is showing below:



5.1.3 Result and analysis



It could be seen the error becomes much slower as the iteration going, it could say our simulation is correct, the exactness will be pretty good as iteration going.

6 (e) Solver Performance Check

6.1 (I) Overrelaxation Parameter

For each grid, we test our overrelaxation parameter from in the range of [1,2], step size=0.1, the change with overrelaxation parameter with its convergence speed is showing below:

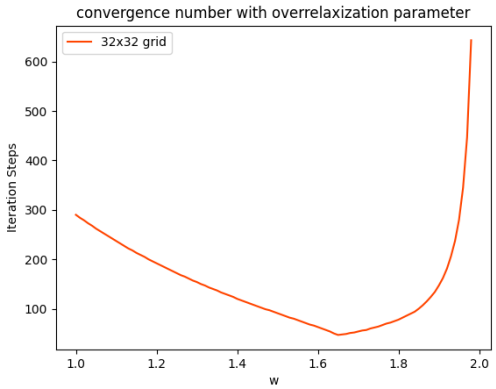


Figure 9: 32x32 grid

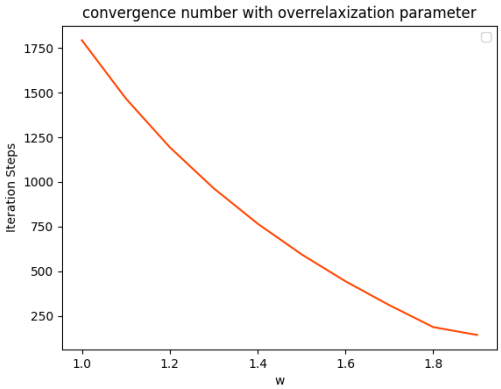


Figure 10: 96x96 grid

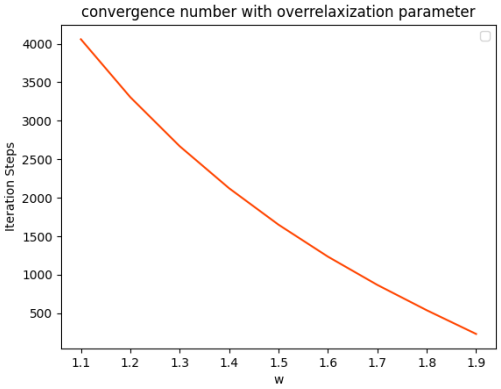


Figure 11: 160x160 grid

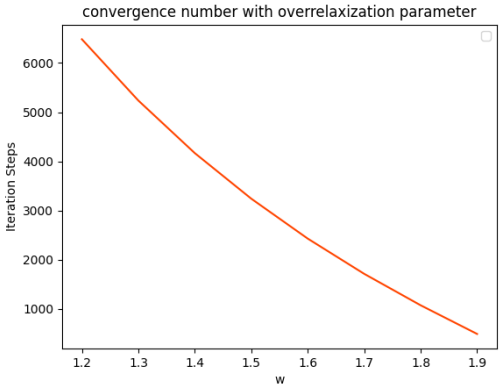


Figure 12: 224x224 grid

Figure 13: Accuracy diagrams for different grid sizes.

Size \ w	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	2.0
32 grid	199	163	132	106	83	63	45	**39**	58	107	$K_{\max}$
96 grid	1794	1468	1195	965	767	595	444	310	187	**143**	$K_{\max}$
160 grid	6001	4059	3307	2670	2124	1651	1236	869	540	**232**	$K_{\max}$
224 grid	$K_{\max}$	$K_{\max}$	6843	5236	4166	3239	2427	1710	1070	**487**	$K_{\max}$

Table 1: The iteration number to reach 10^{-3} Error with different ws

It could be found for 32 size grid, the optimal w is around 1.7, where for 96, 160, and 224 grid, the optimal w is around 1.9. It shows the finier the grid as, the more optimal overrelaxation parameter w move towards 2.

Consider the formula for Amplifaction factor:

$$G = \frac{1-\omega+\frac{\omega}{2}e^{ik_x\Delta x}}{1-\frac{\omega}{2}e^{-ik_x\Delta x}} = \frac{2-2\omega+\omega[\cos\beta_x+i\sin\beta_x]}{2-\omega[\cos\beta_x-i\sin\beta_x]} = \frac{(2-2\omega+\omega\cos\beta_x)+i\omega\sin\beta_x}{2-\omega\cos\beta_x+i\omega\sin\beta_x} \tag{6}$$

6.2 (II) CPU time

For Error get 10^{-5}, w =1, the CPU time for each method is showing below:

Grid Size	CPU Time (seconds)	Iteration Steps (K)	Iteration Speed (it/s)
32x32	0.9722	290	305.96
96x96	70.5959	2616	35.32
160x160	561.3094	7234	12.89
320x320	2136.2018	14181	6.64

Table 2: CPU Time Till Convergnece for Different Grid Sizes

It could be seen number iteration speed is have some relationship with grid size, As the grid become finier, the iteration speed become much more slower, nearly to Flop =~ N^3

6.3 (III) Order of accuracy

The result shown 1st Order Accuracy for K=100:

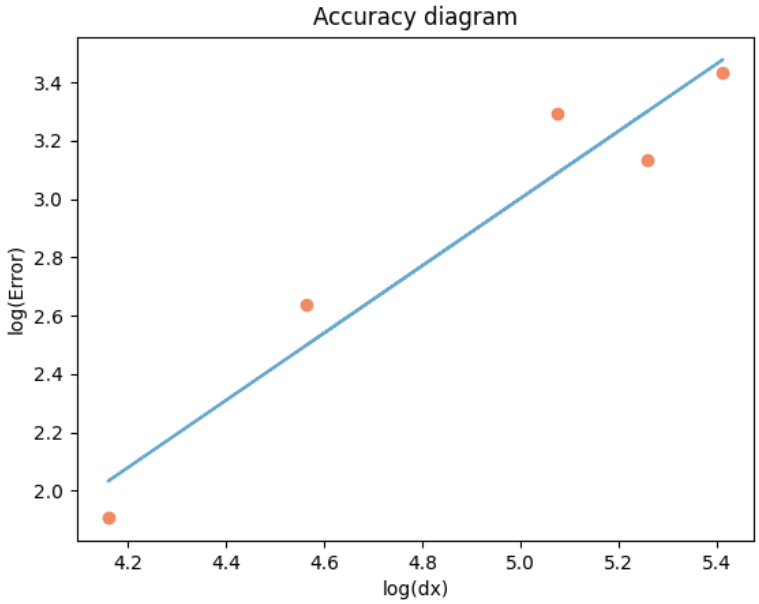


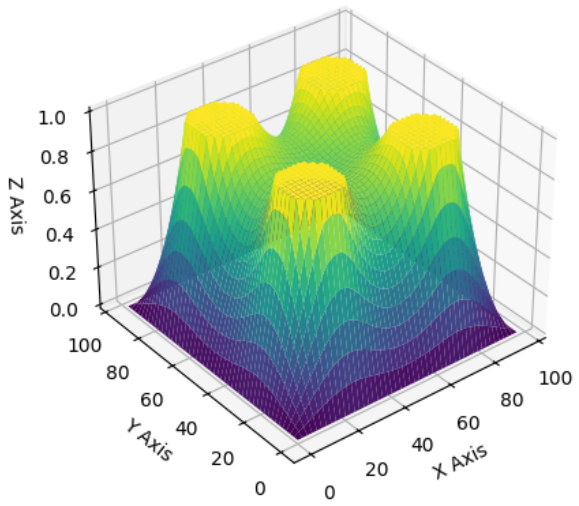
Figure 14: Order of Accuracy log-log

It could be see the slop is about 2, which means the error is 2nd order, this proves the method we are using (point GS-SOR) is second order accuracy.

7 (f) 4-Cylinder Iteration Approach

For 4-Cylinder Domain, adjust the inner boundary for the new condition, the grid generator for this step to generate 1-cylinder inner boundary first, and then duplicate to correspond coordinate. The rest for the solver is same.

The simulation for 4-cylinder condition is showing below:



7.1 (I) Optimal overrelaxation parameter

As we observed in the section (e), ther do exist optimal parameter w even can speed up 224 size grid from 10000 to 487, thus, this time we set up $K_{\max}$ for only 1001, to find out optimal parameter for each grid, and speed up our program.

The result is showing below:

Size \ w	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	2.0
32 grid	309	252	205	164	130	99	71	**54**	75	143	$K_{\max}$
96 grid	$K_{\max}$	$K_{\max}$	$K_{\max}$	$K_{\max}$	$K_{\max}$	820	613	429	263	**199**	$K_{\max}$
160 grid	$K_{\max}$	$K_{\max}$	$K_{\max}$	$K_{\max}$	$K_{\max}$	$K_{\max}$	$K_{\max}$	$K_{\max}$	728	**322**	$K_{\max}$
224 grid	$K_{\max}$	$K_{\max}$	$K_{\max}$	$K_{\max}$	$K_{\max}$	$K_{\max}$	$K_{\max}$	$K_{\max}$	$K_{\max}$	**656**	$K_{\max}$

Table 3: The iteration number to reach 10^{-3} Error with different ws

Compare to our pervious 1-cylinder iteration, it could be seen as the boundary condition become more complicated, the convergence rate decrease, while the optimal parameter is almost in same condition: for 32 grid, is about 1.7, for others, is about 1.9.

The plot for each grid size is showing below:

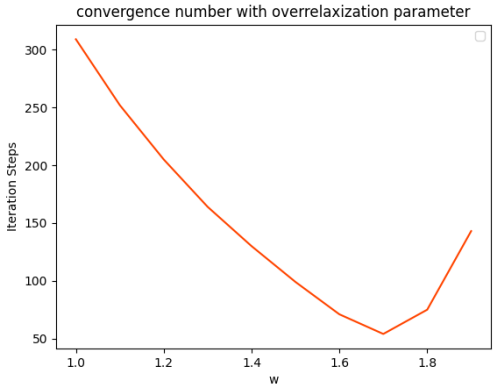


Figure 15: 32x32 grid

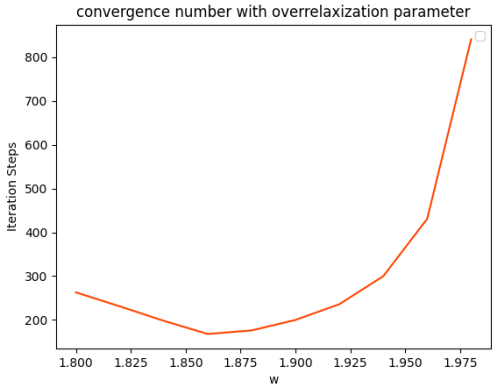


Figure 16: 96x96 grid

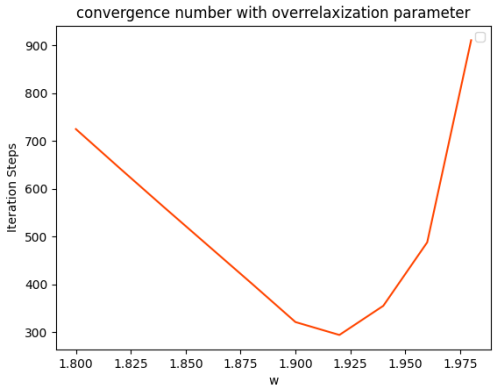


Figure 17: 160x160 grid

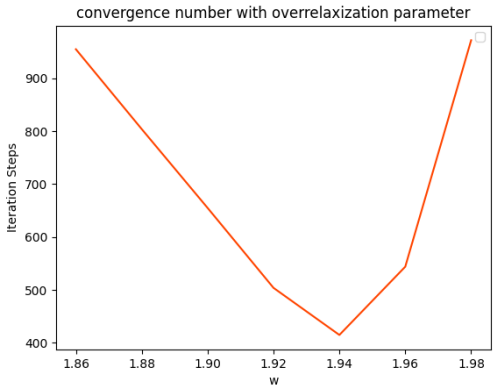


Figure 18: 224x224 grid

Figure 19: Accuracy diagrams for different grid sizes.

For more detailized analysis, we try for 0.02 grid size for 95x96 grid, 160x160 grid, and 224x224 grid, with w in range of [1.7, 2.0]

It could be seen, for 4 cylinder, w stepsize=0.02, the optimal overrelaxation parameter for 96 grid is 1.86, which need 168 iteration to converge to 10^{-3}.

The optimal value for 160 grid is 1.92, need 294 iterations to converge to 10^{-3}.

For 224 grid, is 1.94, which repersent 415 iteration operations to decrease error to 10^{-3}.

7.2 (II) CPU time

For Error get 10^{-5}, w =1, the CPU time for each method is showing below:

Grid Size	CPU Time (seconds)	Iteration Steps (K)	Iteration Speed (it/s)
32x32	1.1654	459	410.28
96x96	70.5959	2616	35.32
160x160	561.3094	7234	12.89
320x320	2136.2018	14181	6.64

Table 4: CPU Time Till Convergnece for Different Grid Sizes

7.3 (III) Order of accuray

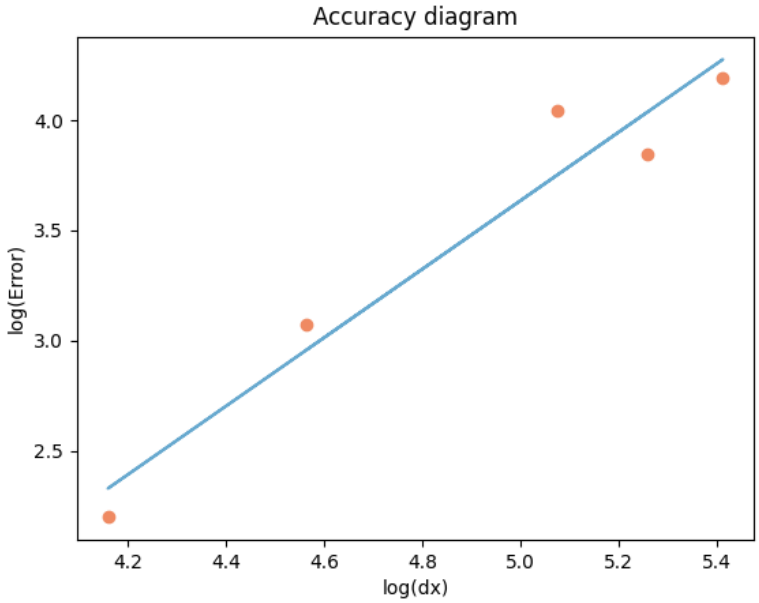


Figure 20: Order of Accuracy log-log

8 (g) Line-SOR Approach

Apply Line SOR to our control equation, showing below:

(i) Line matrix operation
The control equation could be writen as discrete form:

$$\left(\frac{\delta^2}{\Delta x^2}+\frac{\delta^2}{\Delta y^2}\right)u = S$$

Where, we apply line operation:

$$\frac{\delta^2 u^{k+1}}{\Delta x^2}+\frac{\delta^2 u^{k}}{\Delta y^2} = S$$

k+1 is for the new value, where k means the old value, which we already known:

$$u_{i,j}^{k+1} - 2u_{i,j}^{k+1} + u_{i+1,j}^{k+1} + u_{i,j-1}^{k} - 2u_{i,j}^{k+1} + u_{i,j+1}^{k} = S$$

(ii) SOR operation

$$u^{k+1} = (1-\omega)u^{k} + \omega u^{*}$$

$$u_{i,j}^{\text{new}} = (1-\omega)u_{i,j}^{\text{old}} + \omega u_{i,j}^{*}$$

In general, the matrix is showing below:

$$\begin{bmatrix} -1 & 1 & 0 & \cdots & 0 \\ 1 & -4 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & 1 & -4 & 1 \\ 0 & \cdots & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} \vdots \\ u_{i-1} \\ u_i \\ u_{i+1} \\ \vdots \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 1 & 0 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & 1 & 0 & 1 \\ 0 & \cdots & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \vdots \\ u_{j-1} \\ u_j \\ u_{j+1} \\ \vdots \end{bmatrix} + B \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

However, consider the Inner Boundary condition, the matrix need to be adjusted:

$$\begin{bmatrix} A & 0 & 0 \\ 0 & B & 0 \\ 0 & 0 & A \end{bmatrix} \left[u_i\right]_j = \begin{bmatrix} 0 & I & 0 \\ I & 0 & I \\ 0 & I & 0 \end{bmatrix} \left[u_j\right]_i (1 - B_{oundary}) + \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} (B_{oundary})$$

where, $B_{oundary}$ means 'boudnary judgement', which means if its bondary, it will be 1, otherwise, it will be 0.

$$\mathbf{A} = \begin{bmatrix} -4 & 1 & \cdots & 0 \\ 1 & -4 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 1 \\ 0 & \cdots & 1 & -4 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{bmatrix}$$

After we get the matrix equation for Line SOR with booundary, the algorithm is showing below:

Algorithm 3 Line Successive Over-Relaxation (Line SOR) Solver

function LineSOR_Solver($P, i\_max, j\_max, w,$ ... other parameters ...)
  Initialize D, a, b, c matrices
  for $i \leftarrow 1$ **to** $i\_max - 1$ **do**
    for $j \leftarrow 1$ **to** $j\_max - 1$ **do**
      $D[j][i] \leftarrow -(P[j-1][i]+P[j+1][i])\cdot(1-C_{\text{inner}}[j][i])+C_{\text{inner}}[j][i]-C_{\text{left}}[j][i]-C_{\text{right}}[j][i]$
      $c[j][i] \leftarrow (1 - C_{\text{left}}[j][i]) \cdot (1 - C_{\text{inner}}[j][i])$
      $a[j][i] \leftarrow (1 - C_{\text{right}}[j][i]) \cdot (1 - C_{\text{inner}}[j][i])$
      $b[j][i] \leftarrow -4 \cdot (1 - C_{\text{inner}}[j][i]) + C_{\text{inner}}[j][i]$
    end for
  end for
  for $j \leftarrow 1$ **to** $j\_max - 1$ **do**
    $P_n \leftarrow$ TDMA(a,b,c,D)
    for $i \leftarrow 1$ **to** $i\_max - 1$ **do**
      $P_{\text{new}}[j][i] \leftarrow (1 - w) \cdot P[j][i] + w \cdot P_n[i]$
      $P[j][i] \leftarrow P_{\text{new}}[j][i]$
    end for
  end for
  return P
end function

8.1 Line SOR VS. Gauss-Seidel SOR

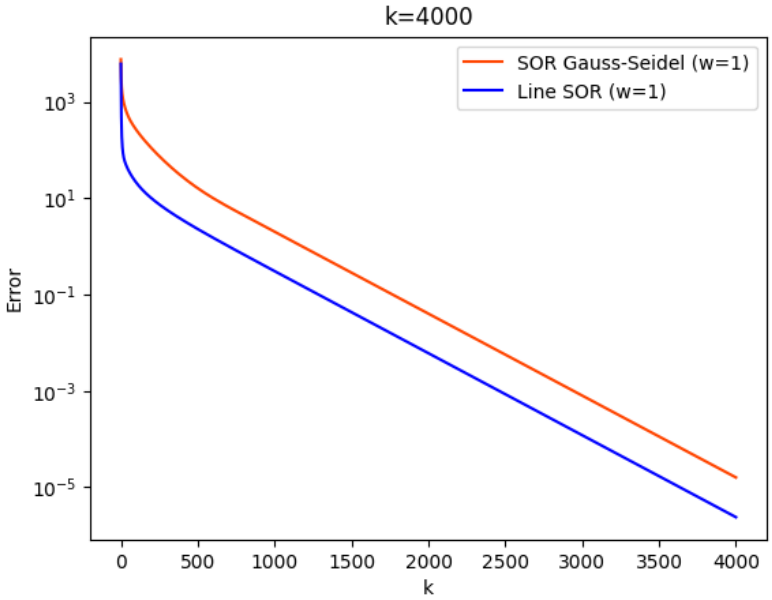


Figure 21: 96x96 grid Comparsion

Compere to Point Gauss-Seidel method, Line SOR is much faster, and converge much faster:

For 4-cylinder platform, Line SOR could speed up to 48.35 it/s, where Point GS-SOR iteraton speed is 21.73 it/s, or 563.64 VS. 141.83 CPU time, that is almost twice faster as GS-SOR method. It also converge faster than point GS-SOR, for iteration calculation and need less iteration steps to converge to same level as GS-SOR.

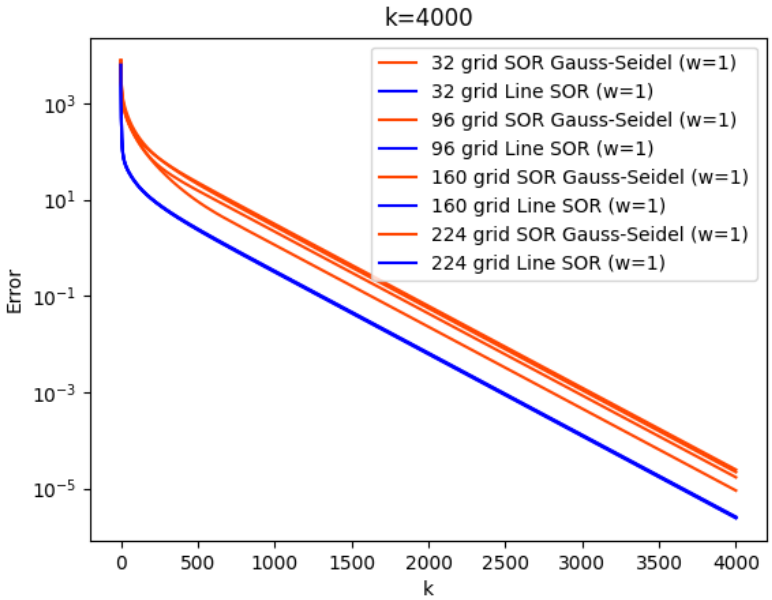


Figure 22: Different grid Comparsion

This plot compared Line SOR with point GS-SOR in different grid size, it could be shown that Line SOR is converge much faster than point GS-SOR in each grid.

9 (h) Potential Flow Calculate

As we known, for potential flow, flow cannot pass the boundary into the cylinder, or anyother boundary not allowed flow to pass, we get:

$$\frac{u_{\text{boundary}} - u_{\text{direction next point}}}{\Delta_{\text{direction}}} = 0$$

$$u_{\text{boundary}} = u_{\text{direction next point}}$$

However, we need to create potential flow, which means it also need another boundary condition to allow flow pass by:

$$\frac{u_{\text{boundary}} - u_{\text{direction next point}}}{\Delta_{\text{direction}}} = Flow$$

$$u_{\text{boundary}} = u_{\text{direction next point}} + \Delta_{\text{direction}} Flow$$

The boundary condition setting is shwoing below:

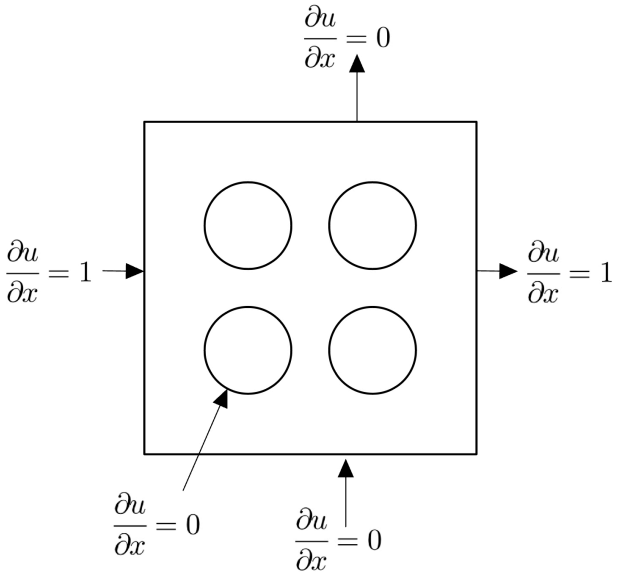


Figure 23: Von-Neumann Boundary Condition

We also need to expand our domain, the method is showing below:

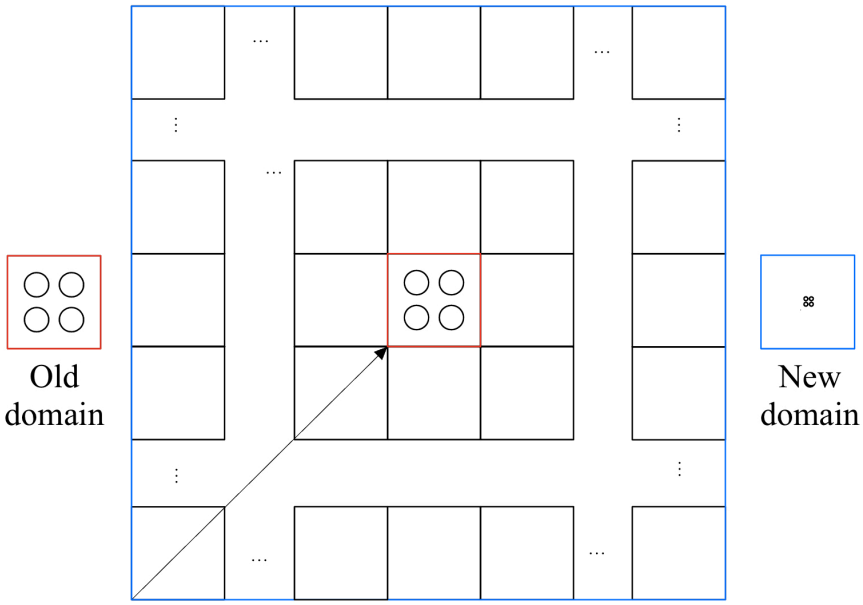


Figure 24: Domain Change

For this configuration, we only need to create a large domain, and transfer our inner boundary (4 cylinders) to that large domain, and calculate the result.

Apply this boundary condition, the result is showing below:

In the normal domain (not been expanded), the reuslt is showing below.

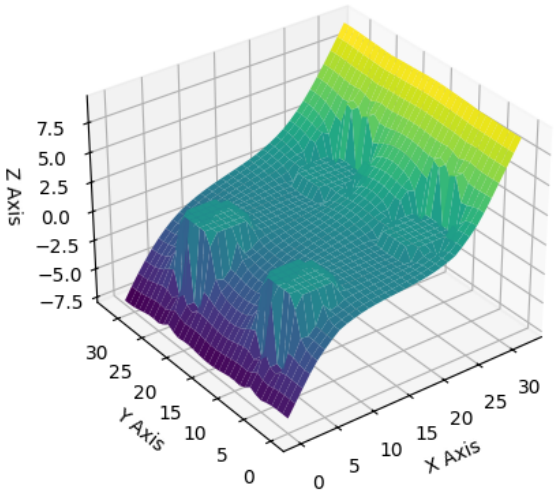


Figure 25: Potential flow pass cylinder in domain

Expand the boundary to 9 times large, the result is showing below:

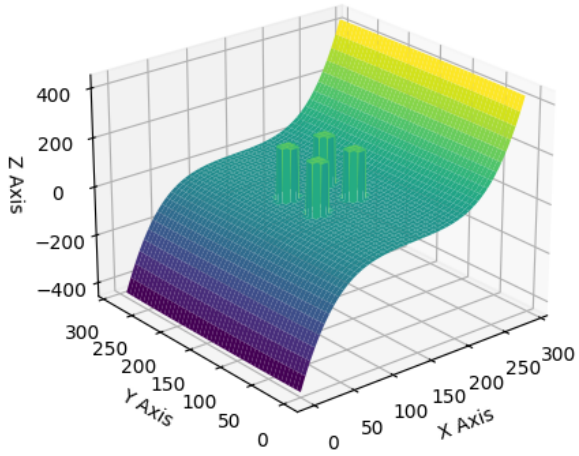


Figure 26: Potential flow pass cylinder in domain on expanded Boundary

Calculate the pressure distribution and streamlines, the result is showing below:

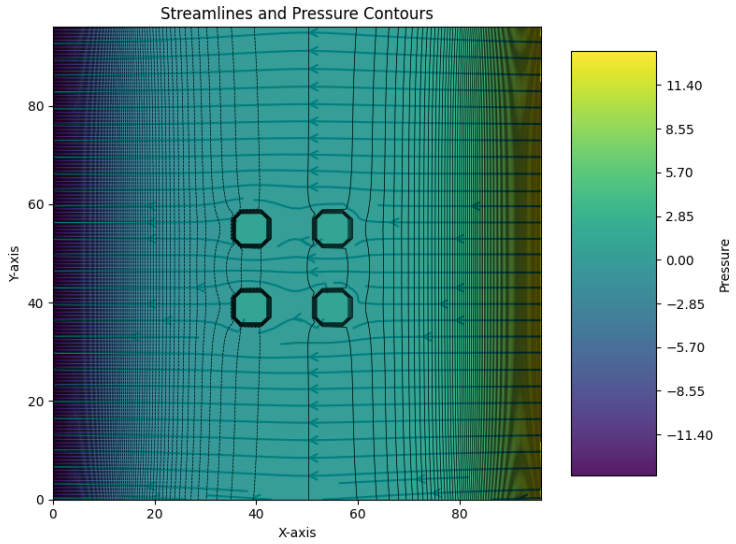


Figure 27: Potential flow pass cylinder at large domain

For much large domain, the result is showing below:

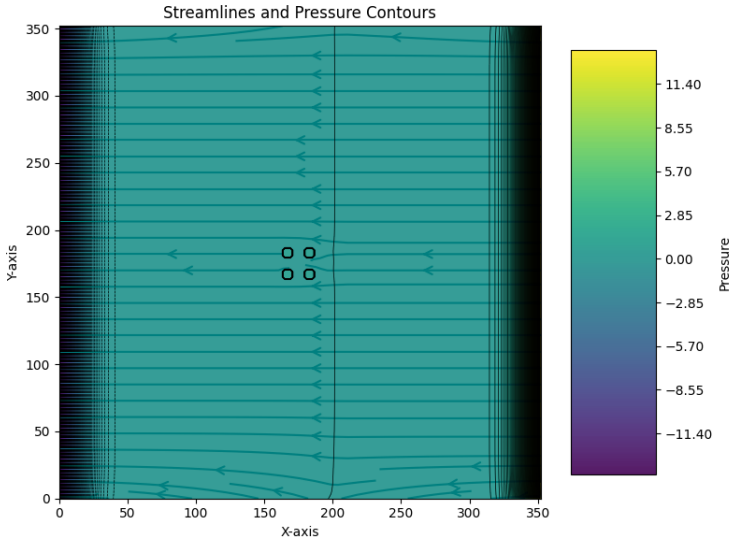


Figure 28: Potential flow pass cylinders in large domain cylinder at large domain

Appendix

Listing 1: Python code–Functions for Grid Generator

```
import copy
import matplotlib.pyplot as plt
#import numpy as np
import matplotlib as mpl


import tqdm
from tqdm import tqdm




##############################################################################################

def real_True(number): # turn complex number to negative
    if isinstance(number, int):
        return number
    elif isinstance(number, float):
        return number
    elif isinstance(number, complex):
        return -100


def int_True(number): # return int
    if isinstance(number, complex):
        return 0
    elif (number % 1==0):
        return -1
    else:
        return 0



def IBcboundary(i, j ,  x_len, y_len, d, r): # calculate boundary points for \
each i,j

    x = i * d
    y = j * d
    i_B1a_0= (x_len/2 - ( (r**2) - (y-y_len/2)**2 )**(1/2))/d
    i_B2b_0=(x_len/2 + ( (r**2) - (y-y_len/2)**2 )**(1/2))/d
    j_B1a_0=(y_len/2 - ( (r**2) - (x-x_len/2)**2 )**(1/2))/d
    j_B2b_0=(y_len/2 + ( (r**2) - (x-x_len/2)**2 )**(1/2))/d


    i_B1a= int(real_True(i_B1a_0))
    i_B2b=int(real_True(i_B2b_0)) +1 + int_True(i_B2b_0)
    j_B1a=int(real_True(j_B1a_0))
    j_B2b=int(real_True(j_B2b_0))+1 + int_True(j_B2b_0)



    return i_B1a, i_B2b, j_B1a, j_B2b
##############################################################################################




##############################################################################################

def IBcondiitionor(  i_B1a, i_B2b, j_B1a, j_B2b, j_max, i_max): # use each \
EP i,j to classify each EP
    GeneralP = [[0 for _ in range(0,j_max+1)] for _ in range(0,i_max+1)]
    P_xbl0 = copy.deepcopy(GeneralP)
    P_xbh0 = copy.deepcopy(GeneralP)
    P_ybl0 = copy.deepcopy(GeneralP)
    P_ybh0 = copy.deepcopy(GeneralP)
```

```
C_xbl = copy.deepcopy(GeneralP)
C_xbh = copy.deepcopy(GeneralP)
C_ybl = copy.deepcopy(GeneralP)
C_ybh = copy.deepcopy(GeneralP)

C_edge = copy.deepcopy(GeneralP)

C_xlyl = copy.deepcopy(GeneralP)
C_xlyh = copy.deepcopy(GeneralP)
C_xhyl = copy.deepcopy(GeneralP)
C_xhyh = copy.deepcopy(GeneralP)



a = 1
b = 10
c = 41
d = 51


for j in range(1,j_max):
    if i_B1a[j]>0:
        P_xbl0[j][i_B1a[j]] = a
    if i_B2b[j]>0:
        P_xbh0[j][i_B2b[j]] = b
for i in range(1,i_max):
    if j_B1a[i]>0:
        P_ybl0[j_B1a[i]][i] = c
    if j_B2b[i]>0:
        P_ybh0[j_B2b[i]][i] = d

for j in range(1,j_max):
    for i in range(1,i_max):
        if ((P_xbl0[j][i] + P_xbh0[j][i]==a+b) or (P_ybl0[j][i] + P_ybh0[j][i]\
        ==c+d)): # find out grid point just on the edge -->C_edge
            C_edge[j][i] = 1

        if (P_xbl0[j][i] + P_ybl0[j][i] == a+c):
         # find out where the edge cut x and y line, where x low y low -->C_xlyl
            C_xlyl[j][i] = 1 - C_edge[j][i]
        if (P_xbl0[j][i] + P_ybh0[j][i] == a+d):
        # find out where the edge cut x and y line, where x low y high -->C_xlyh
            C_xlyh[j][i] = 1 - C_edge[j][i]
        if (P_xbh0[j][i] + P_ybl0[j][i] == b+c):
        # find out where the edge cut x and y line, where x high y low -->C_xhyl
            C_xhyl[j][i] = 1 - C_edge[j][i]
        if (P_xbh0[j][i] + P_ybh0[j][i] == b+d):
        # find out where the edge cut x and y line, where x high y high -->C_xhyh
            C_xhyh[j][i] = 1 - C_edge[j][i]

        special = C_edge[j][i] +  C_xlyl[j][i] + C_xlyh[j][i] + C_xhyl[j][i]\
          + C_xhyh[j][i]

for j in range(1,j_max):
    for i in range(1,int(i_max/2)):

        C_xbl[j][i] = P_xbl0[j][i]//a - C_edge[j][i] -  C_xlyl[j][i] - \
        C_xlyh[j][i] # find out where edge ONLY cut x low lines

for j in range(1,j_max):
    for i in range(int(i_max/2),i_max):

        C_xbh[j][i] = P_xbh0[j][i]//b - C_edge[j][i]  - C_xhyl[j][i] - \
        C_xhyh[j][i] # find out where edge ONLY cut x high lines

for j in range(1,int(j_max/2)):
    for i in range(1,i_max):
        C_ybl[j][i] = P_ybl0[j][i]//c - C_edge[j][i] - C_xlyl[j][i]  - \
        C_xhyl[j][i] # find out where edge ONLY cut y low lines

for j in range(int(j_max/2),j_max):
    for i in range(1,i_max):
        C_ybh[j][i] = P_ybh0[j][i]//d - C_edge[j][i]  - C_xlyh[j][i]  - \
        C_xhyh[j][i] # find out where edge ONLY cut y high lines
```

```
    return C_xbl, C_xbh, C_ybl, C_ybh, C_edge, C_xlyl, C_xlyh, C_xhyl, C_xhyh
##########################################################################################
```

```
##########################################################################################
def Cinner_and_IBi(i_max, j_max, i_Bmin, i_Bmax, j_Bmin, j_Bmax, C_xbl, C_xbh, \
C_ybl, C_ybh, C_edge, C_xlyl, C_xlyh, C_xhyl, C_xhyh):

    C_total = copy.deepcopy(C_xbl)
    C_inner = copy.deepcopy(C_xbl)

    for j in range(0,j_Bmax+1):
        for i in range(0,i_Bmax+1):
            C_total[j][i] = C_xbl[j][i] + C_xbh[j][i] + C_ybl[j][i] + \
            C_ybh[j][i] + C_edge[j][i] + C_xlyl[j][i] + C_xlyh[j][i] + \
            C_xhyl[j][i] + C_xhyh[j][i]
            C_inner[j][i] = 1 - C_total[j][i]

    i_l = [0 for _ in range(0, j_max+1)]
    i_h = [0 for _ in range(0, j_max+1)]



    for j in range(0,j_Bmin):
        for i in range(0,i_max):
            C_inner[j][i] = 0

    for j in range(j_Bmax+1,j_max):
        for i in range(0,i_max):
            C_inner[j][i] = 0


    for j in range(j_Bmin,j_Bmax+1):
        for i in range(0,i_max+1):
            if C_total[j][i] == 1:
                i_l[j] = i
                break
            else: C_inner[j][i] -= 1
        for i in range(i_max,i_Bmin,-1):
            if C_total[j][i] == 1:
                i_h[j] = i
                break
            else: C_inner[j][i] -= 1

    for j in range(0,j_max+1):
        for i in range(0,i_max+1):
            if C_inner[j][i] < 0: C_inner[j][i] = 0

    return C_inner, C_total, i_l, i_h # C_total means the total inner \
    boundary, while C_inner is the points in the inner boundary
##########################################################################################
```

Listing 2: Python code-Functions for iteration Calculator

```
##########################################################################################
def calculator_GSSOR(P_inputSOR, j_limlow, j_limhigh, i_limlow, i_limhigh , w):
#Gauss Sadiel SOR method Pin—>Pout
    PgSOR = copy.deepcopy(P_inputSOR)
    for j in range(j_limlow, j_limhigh+1):
        for i in range(i_limlow, i_limhigh+1):
            PgSOR[j][i] = \
                1/4 * (  PgSOR[j][i-1] + P_inputSOR[j][i+1] + PgSOR[j-1][i] \
                + P_inputSOR[j+1][i] )*w + (1-w)*P_inputSOR[j][i]
    return PgSOR
```

```
def Point_Calculator_IBGSSOR(P, PgSOR, j, i, w, C_inner, C_total, C_xbl, C_xbh, \
C_ybl, C_ybh, C_edge, C_xlyl, C_xlyh, C_xhyl, C_xhyh):


    PgSOR[j][i] = \
        (1/4 * (  PgSOR[j][i-1] + P[j][i+1] + PgSOR[j-1][i] + P[j+1][i] )*w + \
        (1-w)*P[j][i])*(1-C_total[j][i])*(1-C_inner[j][i]) +(\
            C_edge[j][i] \
        + C_xbl[j][i] * IB1_cal(PgSOR[j][i-1],PgSOR[j-1][i],P[j+1][i])\
        + C_xbh[j][i] * IB1_cal(P[j][i+1],PgSOR[j-1][i],P[j+1][i])\
        + C_ybl[j][i] * IB1_cal(PgSOR[j-1][i],PgSOR[j][i-1],P[j][i+1])\
        + C_ybh[j][i] * IB1_cal(P[j+1][i],PgSOR[j][i-1],P[j][i+1])\
        + C_xlyl[j][i] * IB2_cal(PgSOR[j][i-1],PgSOR[j-1][i])\
        + C_xlyh[j][i] * IB2_cal(PgSOR[j][i-1],P[j+1][i])\
        + C_xhyl[j][i] * IB2_cal(P[j][i+1],PgSOR[j-1][i])\
        + C_xhyh[j][i] * IB2_cal(P[j][i+1],P[j+1][i]) ) *(1-C_inner[j][i])\
        + C_inner[j][i] * 1

    return PgSOR[j][i]
#################################################################################################

#################################################################################################
def IB1_cal(a,b,c):
    P = (a*4/3+8/3+b+c)/6
    return P
#################################################################################################

#################################################################################################
def IB2_cal(a,b):
    P = (a+b)/6 + 2/3
    return P
#################################################################################################




#################################################################################################
def GSSOR_Solver(P_input, i_max, j_max, i_Bmin, i_Bmax, j_Bmin, j_Bmax, i_l, i_h, w \
                 ,C_inner, C_total, C_xbl, C_xbh, C_ybl, C_ybh, C_edge, C_xlyl, \
                 C_xlyh, C_xhyl, C_xhyh):

    # Solve for low GP
    P = copy.deepcopy(P_input)

    P = calculator_GSSOR(P, 1 , j_Bmin-1, 1 , i_max-1 , w)

    # Solve for EP + GP in middle
    PgSOR = copy.deepcopy(P)
    for j in range(j_Bmin, j_Bmax+1):


        # solve for left GP
        for i in range(1, i_l[j]):
            PgSOR[j][i] = \
                1/4 * (  PgSOR[j][i-1] + P[j][i+1] + PgSOR[j-1][i] + P[j+1][i] )\
                *w + (1-w)*P[j][i]
        P = PgSOR


        # solve for middle (including point judgement (mathematically))
        for i in range(i_l[j], i_h[j]+1):
            PgSOR[j][i] = Point_Calculator_IBGSSOR(P, PgSOR, j, i, w,\
                                                   C_inner, C_total, C_xbl, C_xbh, \
                                                   C_ybl, C_ybh, C_edge, C_xlyl, \
                                                   C_xlyh, C_xhyl, C_xhyh)
        P = PgSOR

        # solve for right GP
        for i in range(i_h[j]+1, i_max):
            PgSOR[j][i] = \
```

```
                1/4 * (  PgSOR[j][i-1] + P[j][i+1] + PgSOR[j-1][i] + P[j+1][i] )\
                *w + (1-w)*P[j][i]
        P = PgSOR

    # solve for high GP
    P = calculator_GSSOR(P, j_Bmax+1 , j_max-1 ,1, i_max-1 , w)

    return P

##############################################################################################
```

Listing 3: Python code-Functions for Iteration Solver and Result Analysist

```
##############################################################################################


def Res(r,Pin,d,i_l,i_h,i_max,j_max): #calcuate residual (Pin, source)-->Rout
    P = copy.deepcopy(Pin)
    rc = copy.deepcopy(r)
    for j in range(1,j_max):
        for i in range(1,i_l[j]):
            rc[j][i] = \
                ( P[j][i-1] + P[j][i+1] + P[j-1][i] + P[j+1][i] - 4*P[j][i] )
        for i in range(i_h[j]+1,i_max):
            rc[j][i] = \
                ( P[j][i-1] + P[j][i+1] + P[j-1][i] + P[j+1][i] - 4*P[j][i] )
    return rc

def Error(r_in, i_max, j_max): #calculate error Rin-->e out
    r = copy.deepcopy(r_in)
    e = 0
    for j in range(1,j_max):
        for i in range(1,i_max):
            e += abs(r[j][i])   #**2
    return e




def Generaleach(x_len,y_len,d,r,x_c,y_c, Kmax, w):
    ######### create grid information
    i_max = int(x_len/d)  # i in [0,i_max]
    j_max = int(y_len/d)


    P = [[0 for _ in range(0,j_max+1)] for _ in range(0,i_max+1)]

    import random
    for j in range(1,y_len-1):
        for i in range(1,x_len-1):
            P[j][i] = random.randint(-1,1)




    ######### define inner boundary limit
    i_Bmin = int((x_c-r)/d)
    i_Bmax = int((x_c+r)/d) +1 +  int_True((x_c+r)/d)
    j_Bmin = int((y_c-r)/d)
    j_Bmax = int((y_c+r)/d) +1 +  int_True((y_c+r)/d)


    ######### define inner boundary
    iline = [0 for _ in range(0,i_max+1)]
    jline = [0 for _ in range(0,j_max+1)]

    i_B1a = copy.deepcopy(iline)
    i_B2b = copy.deepcopy(iline)
    j_B1a = copy.deepcopy(jline)
    j_B2b = copy.deepcopy(jline)
```

```
    for j in range(j_Bmin,j_Bmax+1):
        for i in range(i_Bmin, i_Bmax+1):
            i_B1a[j], i_B2b[j], j_B1a[i], j_B2b[i] = IBcboundary(i, j , \
            x_len, y_len, d, r)


    C_xbl, C_xbh, C_ybl, C_ybh, C_edge, C_xlyl, C_xlyh, C_xhyl, C_xhyh \
    = IBcondiitionor( i_B1a, i_B2b, j_B1a, j_B2b, j_max, i_max)

    C_inner, C_total, i_l, i_h = Cinner_and_IBi(i_max, j_max, \
    i_Bmin, i_Bmax, j_Bmin, j_Bmax, \
     C_xbl, C_xbh, C_ybl, C_ybh, C_edge, C_xlyl, C_xlyh, C_xhyl, C_xhyh)



    ################################################## Iteration Part ##############################



    r1 = copy.deepcopy(P)
    e1 = [0 for _ in range(0,Kmax+1)]



    P_input = copy.deepcopy(P)


    for k in tqdm(range(0,Kmax+1)):

        P = GSSOR_Solver(P_input, i_max, j_max, i_Bmin, i_Bmax, \
        j_Bmin, j_Bmax, i_l, i_h, w , \
                 C_inner, C_total, C_xbl, C_xbh, C_ybl, C_ybh, C_edge, \
                 C_xlyl, C_xlyh, C_xhyl, C_xhyh)
        P_input = copy.deepcopy(P)

        r1 = Res(r1,P,d,i_l,i_h,i_max,j_max)
        e1[k] = Error(r1, i_max, j_max)

    return e1



############


def eploting(   e32, e96, e160, e224,    d,    Kmax      ):
    x = [0 for _ in range(0, Kmax+1)]
    for c in range(0,Kmax+1):
        x[c] = c
    plt.plot(x, e32, color = 'red', label = 'Grid␣Size␣32')
    plt.plot(x, e96, color = 'orange', label = 'Grid␣Size␣96')
    plt.plot(x, e160, color = 'blue', label = 'Grid␣Size␣160')
    plt.plot(x, e224, color = 'green', label = 'Grid␣Size␣224')
    plt.legend()

    plt.yscale('log')
    plt.xlabel('k')
    plt.ylabel('Error')
    #plt.ylim(0,1)
    plt.ylim(10**(-6),1)

    plt.title("Kmax=%d"%Kmax)




    plt.show()
```

```
    def Ploting3D(P):
        import matplotlib.pyplot as plt
        from mpl_toolkits.mplot3d import Axes3D
        import numpy as np

        data = P


        data_np = np.array(data)

        rows, cols = len(data), len(data[0])

        x = np.linspace(0, cols - 1, cols)
        y = np.linspace(0, rows - 1, rows)
        x, y = np.meshgrid(x, y)

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

        ax.plot_surface(x, y, data_np, cmap='viridis')

        ax.set_xlabel('X Axis')
        ax.set_ylabel('Y Axis')
        ax.set_zlabel('Z Axis')

        plt.show()
```

Listing 4: The python Source code of Main Function

```
def main():

    ########## input mathematical domain inforamtion
    x_len = y_len = len = 1
    r = 0.25
    x_c = x_len/2
    y_c = y_len/2


    ########### Iteration Setup

    w =1
    Kmax = 10000




    #print(P)
    #Ploting3D(P)

    ######### input numerical grid information
    size = 32

    d = len/size


    e32 = Generaleach(x_len,y_len,d,r,x_c,y_c, Kmax, w)

    ######### input numerical grid information
    size = 96

    d = len/size


    e96 = Generaleach(x_len,y_len,d,r,x_c,y_c, Kmax, w)

    ######### input numerical grid information
    size = 160

    d = len/size
```

```
    e160 = Generaleach(x_len,y_len,d,r,x_c,y_c, Kmax, w)

    ######### input numerical grid information
    size = 224

    d = len/size


    e224 = Generaleach(x_len,y_len,d,r,x_c,y_c, Kmax, w)


    eploting(   e32, e96, e160, e224,    d,    Kmax     )



if __name__ == "__main__":
    main()
```

Listing 5: Python code-Potential Flow

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


    import tqdm
    from tqdm import tqdm



    ######################### TEST FUNCTION
```

```
    ##############################################################################################

    def real_True(number): # turn complex number to negative
        if isinstance(number, int):
            return number
        elif isinstance(number, float):
            return number
        elif isinstance(number, complex):
            return -100


    def int_True(number): # return int
        if isinstance(number, complex):
            return 0
        elif (number % 1==0):
            return -1
        else:
            return 0



    def IBcboundary(i, j ,  x_len, y_len, d, r):
    # calculate boundary points for each i,j

        x = i * d
        y = j * d
        i_B1a_0= (x_len/2 - ( (r**2) - (y-y_len/2)**2 )**(1/2))/d
        i_B2b_0=(x_len/2 + ( (r**2) - (y-y_len/2)**2 )**(1/2))/d
        j_B1a_0=(y_len/2 - ( (r**2) - (x-x_len/2)**2 )**(1/2))/d
```

```
    j_B2b_0=(y_len/2 + ( (r**2) - (x-x_len/2)**2 )**(1/2))/d


    i_B1a= int(real_True(i_B1a_0))
    i_B2b=int(real_True(i_B2b_0)) +1 + int_True(i_B2b_0)
    j_B1a=int(real_True(j_B1a_0))
    j_B2b=int(real_True(j_B2b_0))+1 + int_True(j_B2b_0)



    return i_B1a, i_B2b, j_B1a, j_B2b
##########################################################################################################



##########################################################################################################

def IBcondiitionor(  i_B1a, i_B2b, j_B1a, j_B2b, j_max, i_max):
# use each EP i,j to classify each EP
    GeneralP = [[0 for _ in range(0,j_max+1)] for _ in range(0,i_max+1)]
    P_xbl0 = copy.deepcopy(GeneralP)
    P_xbh0 = copy.deepcopy(GeneralP)
    P_ybl0 = copy.deepcopy(GeneralP)
    P_ybh0 = copy.deepcopy(GeneralP)


    C_xbl = copy.deepcopy(GeneralP)
    C_xbh = copy.deepcopy(GeneralP)
    C_ybl = copy.deepcopy(GeneralP)
    C_ybh = copy.deepcopy(GeneralP)

    C_edge = copy.deepcopy(GeneralP)

    C_xlyl = copy.deepcopy(GeneralP)
    C_xlyh = copy.deepcopy(GeneralP)
    C_xhyl = copy.deepcopy(GeneralP)
    C_xhyh = copy.deepcopy(GeneralP)



    a = 1
    b = 10
    c = 41
    d = 51


    for j in range(1,j_max):
        if i_B1a[j]>0:
            P_xbl0[j][i_B1a[j]] = a
        if i_B2b[j]>0:
            P_xbh0[j][i_B2b[j]] = b
    for i in range(1,i_max):
        if j_B1a[i]>0:
            P_ybl0[j_B1a[i]][i] = c
        if j_B2b[i]>0:
            P_ybh0[j_B2b[i]][i] = d

    for j in range(1,j_max):
        for i in range(1,i_max):
            if ((P_xbl0[j][i] + P_xbh0[j][i]==a+b) or (P_ybl0[j][i] + P_ybh0[j][i]==c+d)):
            # find out grid point just on the edge —>C_edge
                C_edge[j][i] = 1

            if (P_xbl0[j][i] + P_ybl0[j][i] == a+c):
            # find out where the edge cut x and y line, where x low y low —>C_xlyl
                C_xlyl[j][i] = 1 - C_edge[j][i]
            if (P_xbl0[j][i] + P_ybh0[j][i] == a+d):
            # find out where the edge cut x and y line, where x low y high —>C_xlyh
                C_xlyh[j][i] = 1 - C_edge[j][i]
            if (P_xbh0[j][i] + P_ybl0[j][i] == b+c):
            # find out where the edge cut x and y line, where x high y low —>C_xhyl
                C_xhyl[j][i] = 1 - C_edge[j][i]
            if (P_xbh0[j][i] + P_ybh0[j][i] == b+d):
            # find out where the edge cut x and y line, where x high y high —>C_xhyh
```

```
                C_xhyh[j][i] = 1 - C_edge[j][i]

            special = C_edge[j][i] +  C_xlyl[j][i] + C_xlyh[j][i] + C_xhyl[j][i]  + C_xhyh[j][i]

    for j in range(1,j_max):
        for i in range(1,int(i_max/2)):

            C_xbl[j][i] = P_xbl0[j][i]//a - C_edge[j][i] -  C_xlyl[j][i] - C_xlyh[j][i]
            # find out where edge ONLY cut x low lines

    for j in range(1,j_max):
        for i in range(int(i_max/2),i_max):

            C_xbh[j][i] = P_xbh0[j][i]//b - C_edge[j][i]  - C_xhyl[j][i] - C_xhyh[j][i]
            # find out where edge ONLY cut x high lines

    for j in range(1,int(j_max/2)):
        for i in range(1,i_max):
            C_ybl[j][i] = P_ybl0[j][i]//c - C_edge[j][i] - C_xlyl[j][i]  - C_xhyl[j][i]
            # find out where edge ONLY cut y low lines

    for j in range(int(j_max/2),j_max):
        for i in range(1,i_max):
            C_ybh[j][i] = P_ybh0[j][i]//d - C_edge[j][i]  - C_xlyh[j][i]  - C_xhyh[j][i]
            # find out where edge ONLY cut y high lines



    return C_xbl, C_xbh, C_ybl, C_ybh, C_edge, C_xlyl, C_xlyh, C_xhyl, C_xhyh
##############################################################################################




##############################################################################################
def Cinner_and_IBi(i_max, j_max, i_Bmin, i_Bmax, j_Bmin, j_Bmax, C_xbl, C_xbh, C_ybl, C_ybh,\
 C_edge, C_xlyl, C_xlyh, C_xhyl, C_xhyh):

    C_total = copy.deepcopy(C_xbl)
    C_inner = copy.deepcopy(C_xbl)

    for j in range(0,j_Bmax+1):
        for i in range(0,i_Bmax+1):
            C_total[j][i] = C_xbl[j][i] + C_xbh[j][i] + C_ybl[j][i] + C_ybh[j][i]\
             + C_edge[j][i] + C_xlyl[j][i] + C_xlyh[j][i] + C_xhyl[j][i] + C_xhyh[j][i]
            C_inner[j][i] = 1 - C_total[j][i]

    i_l = [0 for _ in range(0, j_max+1)]
    i_h = [0 for _ in range(0, j_max+1)]



    for j in range(0,j_Bmin):
        for i in range(0,i_max):
            C_inner[j][i] = 0

    for j in range(j_Bmax+1,j_max):
        for i in range(0,i_max):
            C_inner[j][i] = 0


    for j in range(j_Bmin,j_Bmax+1):
        for i in range(0,i_max+1):
            if C_total[j][i] == 1:
                i_l[j] = i
                break
            else: C_inner[j][i] -= 1
        for i in range(i_max,i_Bmin,-1):
            if C_total[j][i] == 1:
                i_h[j] = i
                break
            else: C_inner[j][i] -= 1

    for j in range(0,j_max+1):
        for i in range(0,i_max+1):
```

```
            if C_inner[j][i] < 0: C_inner[j][i] = 0

    return C_inner, C_total, i_l, i_h
    # C_total means the total inner boundary, while C_inner is the points in the inner boundary
##########################################################################################



def inner_boundary_generator(i_max, j_max, d, x_len, y_len, len, x_c, y_c, r, size):



    ######### define inner boundary limit
    i_Bmin = int((x_c-r)/d)
    i_Bmax = int((x_c+r)/d) +1 +  int_True((x_c+r)/d)
    j_Bmin = int((y_c-r)/d)
    j_Bmax = int((y_c+r)/d) +1 +  int_True((y_c+r)/d)


    ######### define inner boundary
    iline = [0 for _ in range(0,i_max+1)]
    jline = [0 for _ in range(0,j_max+1)]

    i_B1a = copy.deepcopy(iline)
    i_B2b = copy.deepcopy(iline)
    j_B1a = copy.deepcopy(jline)
    j_B2b = copy.deepcopy(jline)



    for j in range(j_Bmin,j_Bmax+1):
        for i in range(i_Bmin, i_Bmax+1):
            i_B1a[j], i_B2b[j], j_B1a[i], j_B2b[i] = \
            IBcboundary(i, j ,  x_len, y_len, d, r)


    C_xbl, C_xbh, C_ybl, C_ybh, C_edge, C_xlyl, C_xlyh, C_xhyl, C_xhyh = \
    IBcondiitionor(  i_B1a, i_B2b, j_B1a, j_B2b, j_max, i_max)

    C_inner, C_total, i_l, i_h = Cinner_and_IBi(i_max, j_max, i_Bmin, i_Bmax, j_Bmin, j_Bmax, \
                                                C_xbl, C_xbh, C_ybl, C_ybh, C_edge, C_xlyl,\
                                                C_xlyh, C_xhyl, C_xhyh)

    return  C_inner, C_total, i_l, i_h,\
         i_Bmin, i_Bmax, j_Bmin, j_Bmax,\
            C_xbl, C_xbh, C_ybl, C_ybh, C_edge, C_xlyl, C_xlyh, C_xhyl, C_xhyh \


def multiplyA(P, C_in):

    C_out = copy.deepcopy(P)

    for j in range(1, len(C_in)-1):
        for i in range(1,len(C_in)-1):
            C_out[j][i] += C_in[j][i]
            C_out[j][i+len(C_in)-1] += C_in[j][i]
            C_out[j+len(C_in)-1][i] += C_out[j][i]
            C_out[j+len(C_in)-1][i+len(C_in)-1] = C_in[j][i]
    return C_out


def OBdefine(P):
    i_max =len(P)
    j_max = len(P)
    C_Oxl = copy.deepcopy(P)
    C_Oxh = copy.deepcopy(P)
    C_Oyl = copy.deepcopy(P)
    C_Oyh = copy.deepcopy(P)
    for i in range(1,i_max):
        C_Oxl[0][i] = 1
        C_Oxh [j_max][i] = 1
    for j in range(0,j_max+1):
        C_Oyl [j][0] = 1
        C_Oyh [j][i_max] = 1
```

```
    return C_Oxl, C_Oxh, C_Oyl, C_Oyh


def Large_transfer(F, n, j_max, i_max):
    large_j_max = n * j_max
    large_i_max = n * i_max

    center_j_large = large_j_max // 2
    center_i_large = large_i_max // 2

    center_j_small = j_max // 2
    center_i_small = i_max // 2

    start_j = center_j_large - center_j_small - (0 if j_max % 2 else 1)
    start_i = center_i_large - center_i_small - (0 if i_max % 2 else 1)

    P = [[0 for _ in range(large_i_max)] for _ in range(large_j_max)]

    for j in range(j_max):
        for i in range(i_max):
            P[start_j + j][start_i + i] = F[j][i]

    return P




##########################################################################################
def calculator_GSSOR(P_inputSOR, j_limlow, j_limhigh, i_limlow, i_limhigh , w):
#Gauss Sadiel SOR method Pin-->Pout
    PgSOR = copy.deepcopy(P_inputSOR)
    for j in range(j_limlow, j_limhigh+1):
        for i in range(i_limlow, i_limhigh+1):
            PgSOR[j][i] = \
                1/4 * (  PgSOR[j][i-1] + P_inputSOR[j][i+1] + PgSOR[j-1][i] + \
                P_inputSOR[j+1][i] )*w + (1-w)*P_inputSOR[j][i]
    return PgSOR



def Point_Calculator_IBGSSOR(P, PgSOR, j, i, w, C_inner, C_total, C_xbl, C_xbh, \
C_ybl, C_ybh, C_edge, C_xlyl, C_xlyh, C_xhyl, C_xhyh):


    PgSOR[j][i] = \
        (1/4 * (  PgSOR[j][i-1] + P[j][i+1] + PgSOR[j-1][i] + P[j+1][i] )*w \
        + (1-w)*P[j][i])*(1-C_total[j][i]-C_inner[j][i]) \
        + C_xbl[j][i] * PgSOR[j][i-1]\
        + C_xbh[j][i] * P[j][i+1]\
        + C_ybl[j][i] * PgSOR[j-1][i]\
        + C_ybh[j][i] * P[j+1][i]\
        + C_inner[j][i] *200\
        + C_xlyl[j][i] * (P[j-1][i]+P[j][i-1])/2\
        + C_xlyh[j][i] * (P[j][i-1]+P[j+1][i])/2\
        + C_xhyl[j][i] * (P[j][i+1]+P[j-1][i])/2\
        + C_xhyh[j][i] * (P[j+1][i]+P[j][i+1])/2\


    return PgSOR[j][i]
##########################################################################################

##########################################################################################
def IB1_cal(a,b,c):
    P = (a*4/3+8/3+b+c)/6
    return P
##########################################################################################

##########################################################################################
def IB2_cal(a,b):
    P = (a+b)/6 + 2/3
    return P
##########################################################################################
```

```
#############################################################################################
def GSSOR_Solver(P_input, i_max, j_max,  w , \
                 C_inner, C_total, C_xbl, C_xbh, C_ybl, C_ybh, C_edge, C_xlyl,\
                  C_xlyh, C_xhyl, C_xhyh):

    # Solve for low GP
    P = copy.deepcopy(P_input)


    # Solve for EP + GP in middle
    PgSOR = copy.deepcopy(P)
    U =10



    for j in range(1,j_max):
        for i in range(1, i_max):
            if C_xbl[j][i]==0 and C_xbl[j-1][i]==1 and C_xbl[j+1][i]==1:
                C_xbl[j][i] = 1
            if C_xbh[j][i]==0 and C_xbh[j-1][i]==1 and C_xbh[j+1][i]==1:
                C_xbh[j][i] = 1
            if C_ybl[j][i]==0 and C_ybl[j][i-1]==1 and C_ybl[j][i+1]==1:
                C_ybl[j][i] = 1
            if C_ybh[j][i]==0 and C_ybh[j][i-1]==1 and C_ybh[j][i+1]==1:
                C_ybh[j][i] = 1


    for j in range(0,j_max+1):
        PgSOR[j][0] = P[j][1] - U
        PgSOR[j][i_max] = P[j][i_max-1] +U
    for i in range(1,i_max):
        PgSOR[0][i] = P[1][i]
        PgSOR[j_max][i] = P[j_max-1][i]




    for j in range(1,j_max):
        for i in range(1, i_max):
            PgSOR[j][i] = Point_Calculator_IBGSSOR(P, PgSOR, j, i, w,\
                                                   C_inner, C_total, C_xbl, \
                                                   C_xbh, C_ybl, C_ybh, C_edge,\
                                                    C_xlyl, C_xlyh, C_xhyl, C_xhyh)


    P = PgSOR



    return P

#############################################################################################


def Res(r,Pin,d,C_inner,i_max,j_max): #calcuate residual (Pin, source)-->Rout
    P = copy.deepcopy(Pin)
    rc = copy.deepcopy(r)
    for j in range(1,j_max):
        for i in range(1,i_max):
            rc[j][i] = \
                ( P[j][i-1] + P[j][i+1] + P[j-1][i] + P[j+1][i] - 4*P[j][i] )\
                 *(1-C_inner[j][i])#*(d**2)

    return rc

def Error(r_in, i_max, j_max): #calculate error Rin-->e out
    r = copy.deepcopy(r_in)
    e = 0
    for j in range(1,j_max):
        for i in range(1,i_max):
            e += abs(r[j][i])    #**2
```

```
    return e


############


def eploting( e1, d, Kmax ):
    x = [0 for _ in range(0, Kmax+1)]
    for c in range(0,Kmax+1):
        x[c] = c
    plt.plot(x, e1, color = 'orangered', label = 'SOR Gauss-Seidel(w=1)')
    plt.legend()

    plt.yscale('log')
    plt.xlabel('k')
    plt.ylabel('Error')
    #plt.ylim(0,1)
    #plt.ylim(0,10**(-6))

    plt.title("k=%d"%Kmax)




    plt.show()



def Ploting3D(P):
    import matplotlib.pyplot as plt
    from mpl_toolkits.mplot3d import Axes3D
    import numpy as np

    data = P







    data_np = np.array(data)

    rows, cols = len(data), len(data[0])

    x = np.linspace(0, cols - 1, cols)
    y = np.linspace(0, rows - 1, rows)
    x, y = np.meshgrid(x, y)

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

    ax.plot_surface(x, y, data_np, cmap='viridis')

    ax.set_xlabel('X Axis')
    ax.set_ylabel('Y Axis')
    ax.set_zlabel('Z Axis')

    plt.savefig('3D_diagram.png')

    plt.show()







def Streamlines(P, i_max, j_max):
    import numpy as np
```

```
    import matplotlib.pyplot as plt
    from matplotlib import cm

    P_array = np.array(P)

    V, U = np.gradient(-P_array)

    Y, X = np.mgrid[0:j_max+1, 0:i_max+1]


    fig, ax = plt.subplots(figsize=(8, 6))


    strm = ax.streamplot(X, Y, U, V , color='teal', density=1.0, arrowstyle='->', arrowsize=1.5)

    U, V = np.gradient(-P_array)

    Y, X = np.mgrid[0:j_max+1, 0:i_max+1]

    pressure_contour_filled = ax.contourf(X, Y, P_array, levels=200, cmap=cm.viridis, alpha=0.9)
    # Create contour lines
    pressure_contour_lines = ax.contour(X, Y, P_array, levels=200, colors='black', linewidths=0.5)
    # Add streamlines

    fig.colorbar(pressure_contour_filled, ax=ax, shrink=0.1, aspect=5, label='Pressure')
    # Add labels to contour lines
    #ax.clabel(pressure_contour_lines, inline=True, fontsize=8)

    ax.set_title('Streamlines␣and␣Pressure␣Contours')
    ax.set_xlabel('X-axis')
    ax.set_ylabel('Y-axis')
    plt.tight_layout()
    plt.show()
```

```
##########################################################################################
##########################################################################################
##########################################################################################
##########################################################################################


def main():


    ########## input mathematical domain inforamtion
    x_len = y_len = len = 1
    r = 0.25




    ######### input numerical grid information
    size = 96


    d = len/size


    ######### create grid information
    i_max = int(x_len/d)  # i in [0,i_max]
    j_max = int(y_len/d)
```

```
P = [[0 for _ in range(0,j_max+1)] for _ in range(0,i_max+1)]


# for 0.5*0.5 general mesh


size_A = size/2
x_len_A = y_len_A = len_A= len/2

d_A = d
i_max_A = int(x_len_A/d_A)  # i in [0,i_max]
j_max_A = int(y_len_A/d_A)


r_A = r/2


x_c_A = x_len_A/2
y_c_A = y_len_A/2


C_inner_A, C_total_A, i_l_A, i_h_A,\
     i_Bmin_A, i_Bmax_A, j_Bmin_A, j_Bmax_A,\
        C_xbl_A, C_xbh_A, C_ybl_A, C_ybh_A, C_edge_A, C_xlyl_A, C_xlyh_A, C_xhyl_A, C_xhyh_A \
        = inner_boundary_generator(i_max_A, j_max_A, d_A, x_len_A, y_len_A, len_A, x_c_A, y_c_A, r_A,


#### B



C_inner = multiplyA(P, C_inner_A)
C_total = multiplyA(P, C_total_A)
C_xbl = multiplyA(P, C_xbl_A)
C_xbh = multiplyA(P, C_xbh_A)
C_ybl = multiplyA(P, C_ybl_A)
C_ybh = multiplyA(P, C_ybh_A)
C_edge = multiplyA(P, C_edge_A)
C_xlyl = multiplyA(P, C_xlyl_A)
C_xlyh = multiplyA(P, C_xlyh_A)
C_xhyl = multiplyA(P, C_xhyl_A)
C_xhyh = multiplyA(P, C_xhyh_A)



################################## Large platform

n = 3 # Large times



P = [[0 for _ in range(0,n *j_max+1)] for _ in range(0,n* i_max+1)]

Pin = copy.deepcopy(P)



C_inner = Large_transfer(C_inner,  n, j_max, i_max)
C_total = Large_transfer(C_total, n, j_max, i_max)
C_xbl = Large_transfer(C_xbl, n, j_max, i_max)
C_xbh = Large_transfer(C_xbh, n, j_max, i_max)
C_ybl = Large_transfer(C_ybl, n, j_max, i_max)
C_ybh = Large_transfer(C_ybh, n, j_max, i_max)
C_edge = Large_transfer(C_edge, n, j_max, i_max)
C_xlyl = Large_transfer(C_xlyl, n, j_max, i_max)
C_xlyh = Large_transfer(C_xlyh, n, j_max, i_max)
C_xhyl = Large_transfer(C_xhyl, n, j_max, i_max)
C_xhyh = Large_transfer(C_xhyh, n, j_max, i_max)
```

```
    x_len= n*x_len
    y_len= n*y_len

    i_max = int(x_len/d)  # i in [0,i_max]
    j_max = int(y_len/d)



    #print(C_inner)







    P = [[0 for _ in range(0,j_max+1)] for _ in range(0,i_max+1)]



    #C_Oxl, C_Oxh, C_Oyl, C_Oyh = OBdefine(P)

    import random
    for j in range(1,y_len-1):
        for i in range(1,x_len-1):
            P[j][i] = random.randint(-1,1)




    ######################################################## Iteration Part ################################

    w =1
    Kmax = 3000



    r1 = copy.deepcopy(P)
    e1 = [0 for _ in range(0,Kmax+1)]



    P_input = copy.deepcopy(P)


    for k in tqdm(range(0,Kmax+1)):

        P = GSSOR_Solver(P_input, i_max, j_max,  w , \
                 C_inner, C_total, C_xbl, C_xbh, C_ybl, C_ybh, C_edge, C_xlyl, C_xlyh, C_xhyl, C_xhyh)
        P_input = copy.deepcopy(P)

        r1 = Res(r1,P,d, C_inner, i_max,j_max)
        e1[k] = Error(r1, i_max, j_max)

    Ploting3D(P)
    #print(C_xbh)

    #Streamlines(P, i_max, j_max)



if __name__ == "__main__":
    main()
```