Solution of 2-D heat conduction equation using various solution algorithms

Foundations of Computational Fluid Dynamics AM5630

Name - Anand Zambare

Roll no. - AM21S004

Summary

Two dimensional heat conduction equation which is an elliptic equation has been solved on the rectangular domain using Dirichlet boundary conditions. The various techniques like point Gauss-Seidel, Line Gauss-Seidel and Successive over relaxation etc. are used here to solve the given problem. It has been found that the ADI i.e. Alternating Direction Implicit method is the method which takes least no.of iterations for getting converged temperature values. But the computational time required in all of the cases is comparable and we can't find one single method which is efficient in no. of iterations required as well as the computational time required. Finally steady state temperature contours are plotted for all of the solution algorithms.

1 Problem Definition

Given problem is to solve for the steady state temperature profile in a given 2-D domain. The size of domain is user defined (taken as L = 0.3 units and W = 0.4 units) where L is the length along X-axis and W is the height along Y-axis also the no. of grid points are defined by user. We have to write a generalized code to solve a steady 2-D heat conduction equation using Central difference in Space scheme. The Boundary conditions are to be implemented are of Dirichlet type. So the temperature values defined are on the top and bottom edge of the plate and on the sides of the plates as well. Our task is to calculate the temperature distribution inside the domain. We have implemented all the five ways of calculating the temperature distribution in the domain.

L = Length of domain = 0.3 meter (Taken here)
W = Height of domain = 0.4 meter (Taken here)
N_x = No. of grid points in the X direction = 31 (Taken here)
N_y = No. of grid points in the Y direction = 41 (Taken here)
Thermal diffusivity = 11.254×10^{-4} m/s^2 (Taken here)
Thermal conductivity = 380 W/m.K (Taken here)

2 Governing Equation

For the given problem, to find out the steady state temperature distribution we need to solve steady 2-D Heat Conduction equation which is given as follows.

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \tag{1}$$

We will use Central difference Scheme for space to discretize the governing equation i.e. equation (1).

3 Boundary conditions

We have to implement only boundary conditions here. This problem doesn't involve any initial conditions so such types of problem are also known as Boundary Value Problems i.e. BVP. At X_0, for all Y's, the $T_0 = 10$ and at X_N, for all Y's, the $T_N = 40$. for Y_0, for all X's and Y_N, for all X's T_0 and $T_N = 0$. Physically, The boundary condition states that the temperature at the top is 40 and Temp at the bottom is 10. On the sides of the rectangular plate it is 0 through out.

4 Numerical Formulation

Let's look at the numerical formulation of governing differential equation using Finite Difference Method (FDM). The governing equation is

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \tag{2}$$

Here we have to use the central difference scheme for discritization of the equation. Central Difference in Space gives us

$$\frac{\partial^2 T}{\partial x^2} = \frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{(\delta x)^2} \tag{3}$$

$$\frac{\partial^2 T}{\partial y^2} = \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{(\delta y)^2} \tag{4}$$

Using the above equation (3) and (4) we convert the governing equation into following equation.

$$\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{(\delta x)^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{(\delta y)^2} = 0 \tag{5}$$

if we do some more simplification by assuming

$$\beta = \frac{\delta x}{\delta y} \tag{6}$$

On further simplification and writing the unknown on one side of equation we get

$$(T_{i+1,j} - 2T_{i,j} + T_{i-1,j}) + \beta^2 (T_{i,j+1} - 2T_{i,j} + T_{i,j-1}) = 0 \tag{7}$$

5 Various Solution algorithms

There are five different algorithms implemented to solve the equation (7) which is discritized form of the governing equation. We will look at them one by one

5.1 Point Gauss-Seidel Method

In the point point Gauss-Seidel Method, we will solve only for the point using previous values of the other nodes. For this method the equation (7) is rearranged as follows,

$$T_{i,j}^{k+1} = \frac{1}{2(1+\beta^2)}(T_{i+1,j}^{k} + T_{i-1,j}^{k+1} + \beta^2 T_{i,j+1}^{k} + \beta^2 T_{i,j-1}^{k+1}) \tag{8}$$

Python code for implementing this is attached in the appendix.It takes 1023 iterations for the convergence of the temperature values in the interior domain. The convergence criteria used here is maximum error should be less than 0.01. Where the error is calculated as sum of the absolute value of the difference between the temperature at the interior nodes in k th iteration and k+1 th iteration. Figure (1) depicts the temperature distribution in the domain. The computational time is 13.1518 second.

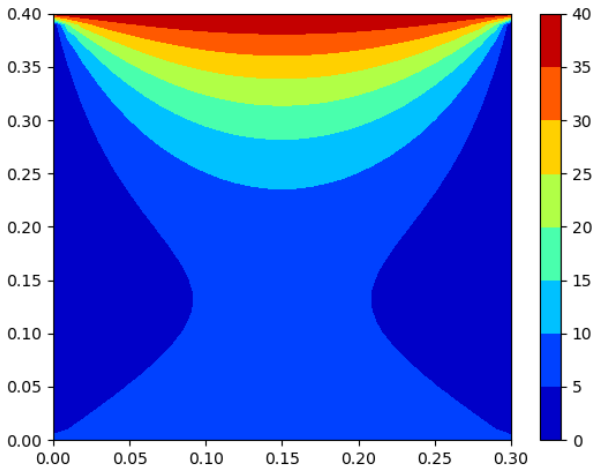

Figure 1: Temperature distribution in case of Point Gauss-Seidel Method.

5.2 Line Gauss-Seidel Method

In the line Gauss-Seidel Method, we will solve for the points or nodes along the x-direction in one step keeping the y or j level same. This is also called as X-Sweep For this method the equation (7) is rearranged as follows,

$$T_{i-1,j}^{k+1} - 2(1+\beta^2)_{i,j}^{k+1} + T_{i+1,j}^{k+1} = -\beta^2(T_{i,j-1}^{k+1} + T_{i,j+1}^{k}) \tag{9}$$

Application of equation (9) to all i nodes at constant j gives us Tri-diagonal system of equation which can be solved using TDMA i.e. Thomas algorithm. Python code for implementing this is attached in the appendix.It takes 522 iterations for the convergence of the temperature values in the interior domain. The convergence criteria used here is maximum error should be less than 0.01. Where the error is calculated as sum of the absolute value of the difference between the temperature at the interior nodes in k th iteration and k+1 th iteration. Figure (2) depicts the temperature distribution in the domain. The computational time is 14.1228 second. We can see from here that though the no.of iterations are less the computational time is almost same.This is because we have to solve the tridiagonal system of equation at each j level while reaching the convergence.

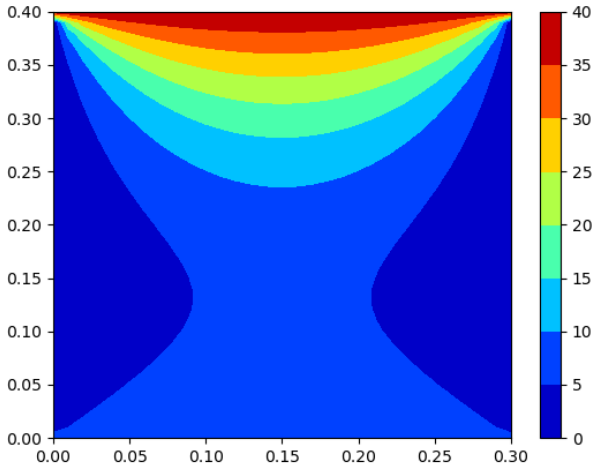

Figure 2: Temperature distribution in case of Line Gauss-Seidel Method.

5.3 Point Successive-Over relaxation Method

In the point Successive-Over relaxation Method, we will solve only for the point using previous values of the other nodes. But the difference is the introduction of omega in this equation. For this method the equation (7) is rearranged as follows, Omega is called as relaxation factor.

$$T_{i,j}^{k+1} = (1-\omega)T_{i,j}^{k} + \frac{\omega}{2(1+\beta^2)}(T_{i+1,j}^{k} + T_{i-1,j}^{k+1} + \beta^2 T_{i,j+1}^{k} + \beta^2 T_{i,j-1}^{k+1}) \tag{10}$$

Python code for implementing this is attached in the appendix.The no.of iterations it takes depend on the omega values. The table 1 depicts the no. of iterations it took to converge the temperature values. The convergence criteria used here is maximum error should be less than 0.01. Where the error is calculated as sum of the absolute value of the difference between the temperature at the interior nodes in k th iteration and k+1 th iteration. Figure (3) depicts the temperature distribution in the domain.

ω	no.of iterations
1.1	857
1.2	714
1.3	591
1.4	482
1.5	384
1.6	295
1.7	211
1.8	127
1.9	142

So we can see that from table the no.of iterations increase when we use omega = 1.9. So the optimum value lies in between 1.8 and 1.9. Now by using omega = 1.85, no. of iterations are 93 so the optimum value is in between 1.8 to 1.85 and in this we calculate for omega 1.825 and use bisection in this interval. Finally we get 1.835 as a optimum omega for which only 85 iterations are needed. Figure (4) represents the graph of omega values vs no.of iterations.

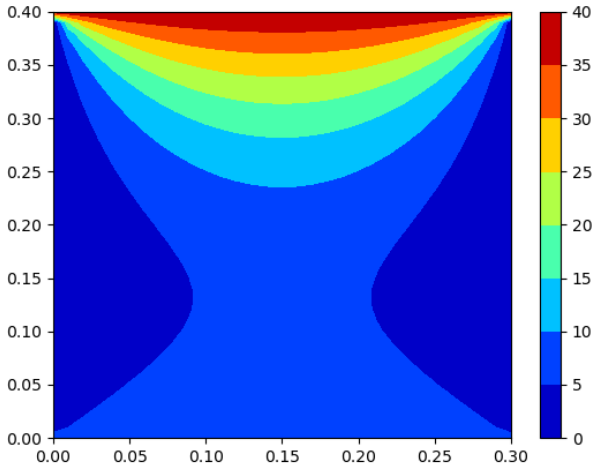

Figure 3: Temperature distribution in case of Point Successive relaxation Method.

5.4 Line Successive-Over relaxation Method

In the line Successive-Over relaxation Method, we will solve for the points or nodes along the x-direction in one step keeping the y or j level same. But the difference is the introduction of omega in this equation. For this method the equation (7) is rearranged as follows, Omega is called as relaxation factor.In the case of line over relaxation we don't get the convergence of temperature for all omega values. We get convergence for very specific values of omega. So here we need to find the omega value which gives less no. of iterations using trial and error method only.

$$T^{k+1}_{i-1,j} - 2(1+\beta^2)T^{k+1}_{i,j} + T^{k+1}_{i+1,j} = -2(1-\omega)(1+\beta^2)T^{k}_{i,j} - \omega\beta^2(T^{k+1}_{i,j-1} + T^{k}_{i,j+1}) \tag{11}$$

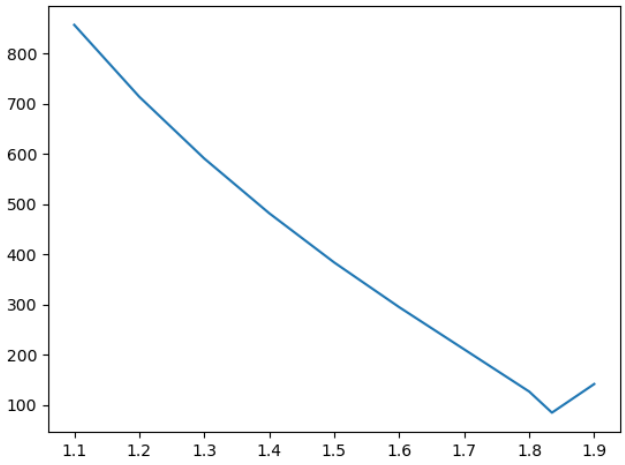

Figure 4: Temperature distribution in case of line successive over relaxation Method.

Here the optimum value of omega for which we get least no.of iterations is 1.3 and the no.of iterations required were 65 only. The computational time required was still comparable to all the other methods because the tridiagonal system is to be solved for each and every iteration. The required code is attached in the appendix. Figure (5) shows the converged temperature distribution for w = 1.3 in case of line over-relaxation method.

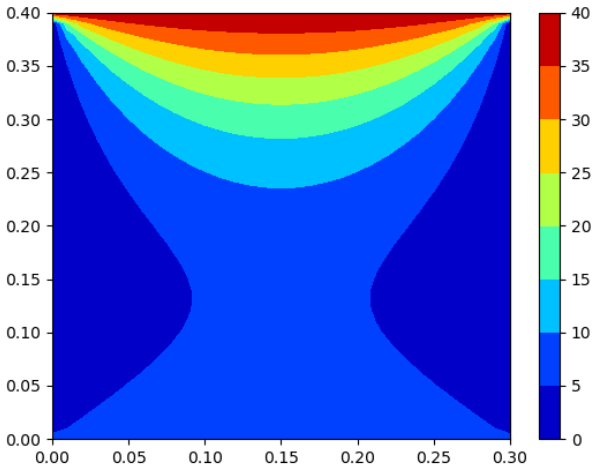

Figure 5: Temperature distribution in case of line Successive relaxation Method.

5.5 Alternate Direction Implicit Method i.e. ADI Method

In this method, we use the two steps within one iteration only. In first step we do X-sweep like what we did in case of the Line gauss-seidel and then we do Y-sweep with the same logic. This is used to reduce the required no.of iterations drastically when the relaxation factor is 1. The relaxation factor can also be included here to accelerate the solution further more. But in this assignment only focus is on ADI method, so we haven't done the relaxation factor calculation here. The discritized equation (7) takes the below form in this case.

$$T_{i-1,j}^{k+\frac{1}{2}} - 2(1+\beta^2)_{i,j}^{k+\frac{1}{2}} + T_{i+1,j}^{k+\frac{1}{2}} = -\beta^2(T_{i,j-1}^{k+\frac{1}{2}} + T_{i,j+1}^{k}) \tag{12}$$

$$T_{i,j-1}^{k+1} - 2(1+\beta^2)_{i,j}^{k+1} + T_{i,j+1}^{k+1} = -\beta^2(T_{i-1,j}^{k+1} + T_{i+1,j}^{k+\frac{1}{2}}) \tag{13}$$

Equation (12) is solved to get the first inter-step values and then all of these interstep values are used to solve for full step values. This procedure is called alternate direction implicit as we solve for X and Y alternately within the iterations. Figure (6) depicts the solution for the temperature distribution in given domain. It takes 279 iterations only for it's convergence without any relaxation factor. Which is the fastest converging method when we compare all methods without relaxation.The code for ADI method is attached in Appendix.

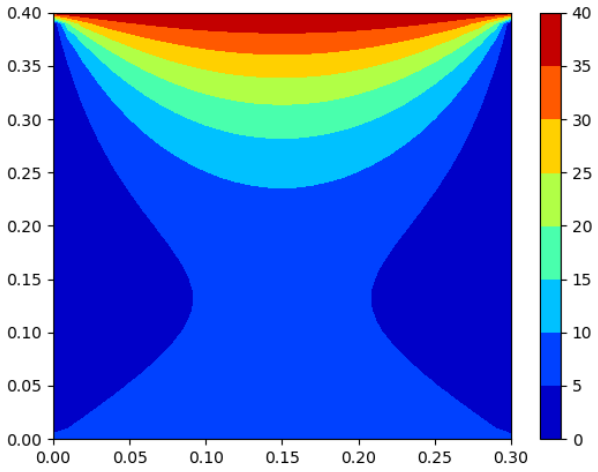

Figure 6: Temperature distribution in case of line Successive relaxation Method.

6 Results

Following important results were found through this exercise.
1. The no.of iterations required to get converged solution is more in case of Point Gauss-Seidel method and least in ADI method when all are compared without any relaxation factor.
2. The no.of iterations required to get converged solution is more in case of Point successive over relaxation method and least in ADI method when all are compared with relaxation factor. Though

the value of relaxation factor is different for different algorithms.
3. The Computational time required in ADI method is more than other methods since we do x as well as y sweeping for that method. Solving tri-diagonal matrix system always consumes more computational time than required for point based algorithms. 4. omega = 1.835 is the optimum omega value for solving this problem using PSOR i.e. point successive over relaxation ($\omega = 1.835$) 4. omega = 1.3 is the optimum omega value for solving this problem using LSOR i.e. line successive over relaxation ($\omega = 1.3$)

7 Appendix

7.1 Python code for PGSM

import matplotlib.pyplot as plt
from numpy import *
import time
$\text{stat}_time = time.time()$
$W = 0.4 width of the plate along Y$
$L = 0.3 length of the plate along X$
$Nx = 31 no. of nodes in the x direction$
$Ny = 41 no. of nodes in the y direction$
$dx = L/(Nx-1) grid size in x direction$
$dy = W/(Ny-1) grid size in y direction$
$beta = dx/dy numerical constant used in formulation$
$X = arange(0, L+dx, dx) X vector$
$Y = arange(0, W+dy, dy) Y vector$
$XX, YY = meshgrid(X, Y)$
$Errormax = 0.01$
$initial condition in the domain$
$T = zeros((Ny, Nx)) Initialize Temperature in 2-D domain$
$def BC(T1):$
$T1[:, 0] = 0$
$T1[:, Nx-1] = 0$
$T1[0, :] = 10$
$T1[Ny-1, :] = 40$
$return T1$

def error(T1, T2):
err = 0
T2 is the latest value and T1 is the one iteration before value
for i in range(1, Ny-1):
for j in range(1, Nx-1):
err = err + abs(T2[i, j] - T1[i, j])
return err

T = BC(T)
$\mathrm{T}_prev = T.copy()$
$plt.figure(1)$
$plt.contourf(XX, YY, T, cmap =' jet')$
$plt.colorbar()$
$foriinrange(1, Ny - 1) :$
$forjinrange(1, Nx - 1) :$
$T[i, j] = (1/(2 * (1 + (beta * *2)))) * (T[i, j + 1] + T[i, j - 1] + ((beta * *2) * (T[i + 1, j] + T[i - 1, j])))$
$T = BC(T)$
$Error = error(T_prev, T)$
$iter = 1$
$whileError > Errormax :$
$T_prev = T.copy()$
$foriinrange(1, Ny - 1) :$
$forjinrange(1, Nx - 1) :$
$T[i, j] = (1/(2 * (1 + (beta * *2)))) * (T[i, j + 1] + T[i, j - 1] + ((beta * *2) * (T[i + 1, j] + T[i - 1, j])))$
$T = BC(T)$
$iter = 1 + iter$
$Error = error(T_prev, T)$
$plt.figure(2)$
$plt.contourf(XX, YY, T, cmap =' jet')$
$plt.colorbar()$
$plt.show()$
$print(iter)$
$end_time = time.time()$
$print(end_time - stat_time)$

7.2 Python code for LGSM

import matplotlib.pyplot as plt
from numpy import *
import time
$\mathrm{stat}_time = time.time()$
$W = 0.4 width of the plate along Y$
$L = 0.3 length of the plate along X$
$Nx = 31 no. of nodes in the x direction$
$Ny = 41 no. of nodes in the y direction$
$dx = L/(Nx - 1) grid size in x direction$
$dy = W/(Ny - 1) grid size in y direction$
$beta = dx/dy numerical constant used in formulation$
$X = arange(0, L + dx, dx) X vector$
$Y = arange(0, W + dy, dy) Y vector$
$XX, YY = meshgrid(X, Y)$
$Errormax = 0.01$
$initial condition in the domain$
$T = zeros((Ny, Nx)) Initialize Temperature in 2 - D domain$
$defBC(T1) :$

$T1[:,0] = 0$
$T1[:,Nx-1] = 0$
$T1[0,:] = 10$
$T1[Ny-1,:] = 40$
$return T1$
$def error(T1,T2):$
$err = 0$
$T2 is the latest value and T1 is the one iteration before value$
$for i in range(1,Ny-1):$
$for j in range(1,Nx-1):$
$err = err + abs(T2[i,j] - T1[i,j])$
$return err$

def Solver(Q, R): Q is unknown vector and R is RHS which is known.
m1 = len(Q)
$x_s = zeros(m1) solution vector.$
$x_s[0] = Q[0]$
$x_s[m1-1] = Q[m1-1]$
$s_s = zeros(m1-3)$
$r_s = zeros(m1-3)$
$s_s[0] = (1/(2*(1+(beta**2)))) - (2*(1+(beta**2)))$
$for i in range(1,m1-3):$
$s_s[i] = (-2*(1+(beta**2))) - (1/s_s[0])$
$r_s[0] = (R[0]/(2*(1+(beta**2)))) + R[1]$
$for i in range(1,m1-3):$
$r_s[i] = R[i+1] - (r_s[i-1]/s_s[i-1])$
$x_s[m1-2] = r_s[m1-4]/s_s[m1-4]$
$for i in range(m1-3,1,-1):$
$x_s[i] = (r_s[i-2] - x_s[i+1])/(s_s[i-2])$
$x_s[1] = (R[0] - x_s[2])/(-2*(1+(beta**2)))$
$return x_s$

def RHS(T00, T11, T22):
m1 = len(T00) or len(T22)
B11 = zeros(m1-2)
for i in range(1, m1-1):
B11[i-1] = (-1*(beta**2))*(T00[i]+T22[i])
B11[0] = B11[0]-T11[0]
B11[m1-3] = B11[m1-3]-T11[m1-1]
return B11

T = BC(T)
$T_prev = T.copy()$
$plt.figure(1)$
$plt.contourf(XX,YY,T,cmap='jet')$
$plt.colorbar()$
$k = 1$
$while k < Ny - 1:$

$R = RHS(T[k-1,:], T[k,:], T[k+1,:])$
$T[k,:] = Solver(T[1,:], R)$
$k = k + 1$
$T = BC(T)$
$Error = error(T_prev, T)$
$iter = 1$

while Error ¿ Errormax:
$T_prev = T.copy()$
$k = 1$
$whilek < Ny - 1 :$
$R = RHS(T[k-1,:], T[k,:], T[k+1,:])$
$T[k,:] = Solver(T[1,:], R)$
$k = k + 1$
$T = BC(T)$
$iter = 1 + iter$
$Error = error(T_prev, T)$
$plt.figure(2)$
$plt.contourf(XX, YY, T, cmap =' jet')$
$plt.colorbar()$
$plt.show()$
$print(Error, iter)$
$end_time = time.time()$
$print(end_time - stat_time)$

7.3 Python code for PSOR

import matplotlib.pyplot as plt
from numpy import *
import time
$stat_time = time.time()$
$W = 0.4 width of the plate along Y$
$L = 0.3 length of the plate along X$
$Nx = 31 no. of nodes in the x direction$
$Ny = 41 no. of nodes in the y direction$
$dx = L/(Nx-1) grid size in x direction$
$dy = W/(Ny-1) grid size in y direction$
$beta = dx/dy numerical constant used in formulation$
$X = arange(0, L + dx, dx) X vector$
$Y = arange(0, W + dy, dy) Y vector$
$XX, YY = meshgrid(X, Y)$
$Errormax = 0.01$
$omega = 1.835 relaxation factor$
$initial condition in the domain$
$T = zeros((Ny, Nx)) Initialize Temperature in 2 - D domain$
$def BC(T1) :$
$T1[:, 0] = 0$
$T1[:, Nx-1] = 0$

$T1[0,:] = 10$
$T1[Ny-1,:] = 40$
$returnT1$

def error(T1, T2):
err = 0
T2 is the latest value and T1 is the one iteration before value
for i in range(1, Ny-1):
for j in range(1, Nx-1):
err = err + abs(T2[i, j] - T1[i, j])
return err

T = BC(T)
$T_prev = T.copy()$
$plt.figure(1)$
$plt.contourf(XX, YY, T, cmap =' jet')$
$plt.colorbar()$
$foriinrange(1, Ny-1):$
$forjinrange(1, Nx-1):$
$T[i,j] = ((1-omega) * (T_prev[i,j])) + ((omega/(2*(1+(beta**2)))) * (T[i,j+1] + T[i,j-1] +$
$((beta**2) * (T[i+1,j] + T[i-1,j]))))$
$T = BC(T)$
$Error = error(T_prev, T)$
$iter = 1$
$whileError > Errormax:$
$T_prev = T.copy()$
$foriinrange(1, Ny-1):$
$forjinrange(1, Nx-1):$
$T[i,j] = ((1-omega) * (T_prev[i,j])) + ((omega/(2*(1+(beta**2)))) * (T[i,j+1] + T[i,j-1] +$
$((beta**2) * (T[i+1,j] + T[i-1,j]))))$
$T = BC(T)$
$iter = 1 + iter$
$Error = error(T_prev, T)$
$plt.figure(2)$
$plt.contourf(XX, YY, T, cmap =' jet')$
$plt.colorbar()$
$plt.show()$
$print(iter)$
$end_time = time.time()$
$print(end_time - stat_time)$

7.4 Python code for LSOR

import matplotlib.pyplot as plt
from numpy import *
import time
$stat_time = time.time()$

$W = 0.4 width of the plate along Y$
$L = 0.3 length of the plate along X$
$Nx = 31 no. of nodes in the x direction$
$Ny = 41 no. of nodes in the y direction$
$dx = L/(Nx-1) grid size in x direction$
$dy = W/(Ny-1) grid size in y direction$
$beta = dx/dy numerical constant used in formulation$
$X = arange(0, L+dx, dx) X vector$
$Y = arange(0, W+dy, dy) Y vector$
$XX, YY = meshgrid(X, Y)$
$Errormax = 0.01$
$initial condition in the domain$
$T = zeros((Ny, Nx)) Initialize Temperature in 2-D domain$
$def BC(T1):$
$T1[:, 0] = 0$
$T1[:, Nx-1] = 0$
$T1[0, :] = 10$
$T1[Ny-1, :] = 40$
$return T1$
$def error(T1, T2):$
$err = 0$
$T2 is the latest value and T1 is the one iteration before value$
$for i in range(1, Ny-1):$
$for j in range(1, Nx-1):$
$err = err + abs(T2[i, j] - T1[i, j])$
$return err$

def Solver(Q, R): Q is unknown vector and R is RHS which is known.
m1 = len(Q)
$x_s = zeros(m1) solution vector.$
$x_s[0] = Q[0]$
$x_s[m1-1] = Q[m1-1]$
$s_s = zeros(m1-3)$
$r_s = zeros(m1-3)$
$s_s[0] = ((omega**2)/(2*(1+(beta**2)))) - (2*(1+(beta**2)))$
$for i in range(1, m1-3):$
$s_s[i] = (-2*(1+(beta**2))) - ((omega**2)/s_s[0])$
$r_s[0] = (omega * R[0]/(2*(1+(beta**2)))) + R[1]$
$for i in range(1, m1-3):$
$r_s[i] = R[i+1] - ((omega * r_s[i-1])/s_s[i-1])$
$x_s[m1-2] = r_s[m1-4]/s_s[m1-4]$
$for i in range(m1-3, 1, -1):$
$x_s[i] = (r_s[i-2] - (omega * x_s[i+1]))/(s_s[i-2])$
$x_s[1] = (R[0] - (omega * x_s[2]))/(-2*(1+(beta**2)))$
$return x_s$

def RHS(T00, T11, T22):
m1 = len(T00) or len(T22)
B11 = zeros(m1-2)

for i in range(1, m1-1):
B11[i-1] = (-(1-omega)*T10[i])+((omega*-1*(beta**2))*(T00[i]+T22[i]))
B11[0] = B11[0]-T11[0]
B11[m1-3] = B11[m1-3]-T11[m1-1]
return B11

T = BC(T)
T$_p rev = T.copy()$
$plt.figure(1)$
$plt.contourf(XX, YY, T, cmap =' jet')$
$plt.colorbar()$
$k = 1$
$while k < Ny - 1 :$
$R = RHS(T[k-1,:], T[k,:], T[k+1,:])$
$T[k,:] = Solver(T[1,:], R)$
$k = k + 1$
$T = BC(T)$
$Error = error(T_p rev, T)$
$iter = 1$

while Error ¿ Errormax:
T$_p rev = T.copy()$
$k = 1$
$while k < Ny - 1 :$
$R = RHS(T[k-1,:], T[k,:], T[k+1,:])$
$T[k,:] = Solver(T[1,:], R)$
$k = k + 1$
$T = BC(T)$
$iter = 1 + iter$
$Error = error(T_p rev, T)$
$plt.figure(2)$
$plt.contourf(XX, YY, T, cmap =' jet')$
$plt.colorbar()$
$plt.show()$
$print(Error, iter)$
$end_t ime = time.time()$
$print(end_t ime - stat_t ime)$

7.5 Python code for ADI

import matplotlib.pyplot as plt
from numpy import *
import time
stat$_t ime = time.time()$
$W = 0.4 width of the plate along Y$
$L = 0.3 length of the plate along X$
$Nx = 31 no. of nodes in the x direction$

$Ny = 41 no. of nodes in the y direction$
$dx = L/(Nx-1) grid size in x direction$
$dy = W/(Ny-1) grid size in y direction$
$beta = dx/dy numerical constant used in formulation$
$X = arange(0, L+dx, dx) X vector$
$Y = arange(0, W+dy, dy) Y vector$
$XX, YY = meshgrid(X, Y)$
$Errormax = 0.01$
$initial condition in the domain$
$T = zeros((Ny, Nx)) Initialize Temperature in 2-D domain$
$def BC(T1):$
$T1[:, 0] = 0$
$T1[:, Nx-1] = 0$
$T1[0, :] = 10$
$T1[Ny-1, :] = 40$
$return T1$
$def error(T1, T2):$
$err = 0$
$T2 is the latest value and T1 is the one iteration before value$
$for i in range(1, Ny-1):$
$for j in range(1, Nx-1):$
$err = err + abs(T2[i, j] - T1[i, j])$
$return err$

def Solver$_1(Q, R)$: $P is coefficient matrix. Q is unknown vector and R is RHS which is known.$
$m1 = len(Q)$
$x_s = zeros(m1) solution vector.$
$x_s[0] = Q[0]$
$x_s[m1-1] = Q[m1-1]$
$s_s = zeros(m1-3)$
$r_s = zeros(m1-3)$
$s_s[0] = (1/(2*(1+(beta**2)))) - (2*(1+(beta**2)))$
$for i in range(1, m1-3):$
$s_s[i] = (-2*(1+(beta**2))) - (1/s_s[0])$
$r_s[0] = (R[0]/(2*(1+(beta**2)))) + R[1]$
$for i in range(1, m1-3):$
$r_s[i] = R[i+1] - (r_s[i-1]/s_s[i-1])$
$x_s[m1-2] = r_s[m1-4]/s_s[m1-4]$
$for i in range(m1-3, 1, -1):$
$x_s[i] = (r_s[i-2] - x_s[i+1])/(s_s[i-2])$
$x_s[1] = (R[0] - x_s[2])/(-2*(1+(beta**2)))$
$return x_s$
$def RHS_1(T00, T11, T22):$
$m1 = len(T00) or len(T22)$
$B11 = zeros(m1-2)$
$for i in range(1, m1-1):$
$B11[i-1] = (-1*(beta**2))*(T00[i] + T22[i])$
$B11[0] = B11[0] - T11[0]$
$B11[m1-3] = B11[m1-3] - T11[m1-1]$
$return B11$

def $\mathrm{RHS}_2(T00, T11, T22)$:
$m1 = len(T00)$
$B22 = zeros(m1 - 2)$
$foriinrange(1, m1 - 1)$:
$B22[i - 1] = -1 * (T00[i] + T11[i])$
$B22[0] = -1 * (T00[0] + T11[0]) - ((beta * *2) * T22[0])$
$B22[m1 - 3] = -1 * (T00[m1 - 2] + T11[m1 - 2]) - ((beta * *2) * T22[m1 - 1])$
$returnB22$

def $\mathrm{Solver}_2(Q, R)$:
$m1 = len(Q)$
$y_s = zeros(m1)solutionvector.$
$y_s[0] = Q[0]$
$y_s[m1 - 1] = Q[m1 - 1]$
$s_s = zeros(m1 - 3)$
$r_s = zeros(m1 - 3)$
$s_s[0] = ((beta * *4)/(2 * (1 + (beta * *2)))) - (2 * (1 + (beta * *2)))$
$foriinrange(1, m1 - 3)$:
$s_s[i] = (-2 * (1 + (beta * *2))) - ((beta * *4)/s_s[0])$
$r_s[0] = ((R[0] * (beta * *2))/(2 * (1 + (beta * *2)))) + R[1]$
$foriinrange(1, m1 - 3)$:
$r_s[i] = R[i + 1] - ((beta * *2) * r_s[i - 1]/s_s[i - 1])$
$y_s[m1 - 2] = r_s[m1 - 4]/s_s[m1 - 4]$
$foriinrange(m1 - 3, 1, -1)$:
$y_s[i] = (r_s[i - 2] - ((beta * *2) * y_s[i + 1]))/(s_s[i - 2])$
$y_s[1] = (R[0] - ((beta * *2) * y_s[2]))/(-2 * (1 + (beta * *2)))$
$returny_s$

T = BC(T)
$\mathrm{T}_prev = T.copy()$
$k = 1$
$whilek < Ny - 1$:
$R = RHS_1(T[k - 1, :], T[k + 1, :], T[k, :])$
$T[k, :] = Solver_1(T[k, :], R)$
$k = k + 1$
$T = BC(T)$
$k = 1$
$whilek < Nx - 1$:
$R = RHS_2(T[:, k - 1], T[:, k + 1], T[:, k])$
$T[:, k] = Solver_2(T[:, k], R)$
$k = k + 1$
$T = BC(T)$
$Error = error(T_prev, T)$
$iter = 1$
$whileError > Errormax$:
$T_prev = T.copy()$
$k = 1$

$while k < Ny - 1:$
$R = RHS_1(T[k-1,:], T[k,:], T[k+1,:])$
$T[k,:] = Solver_1(T[k,:], R)$
$k = k + 1$
$T = BC(T)$
$k = 1$
$while k < Nx - 1:$
$R = RHS_2(T[:,k-1], T[:,k+1], T[:,k])$
$T[:,k] = Solver_2(T[:,k], R)$
$k = k + 1$
$T = BC(T)$
$Error = error(T_prev, T)$
$iter = 1 + iter$
$print(iter, Error)$
$plt.figure(1)$
$plt.contourf(XX, YY, T, cmap =' jet')$
$plt.colorbar()$
$plt.show()$

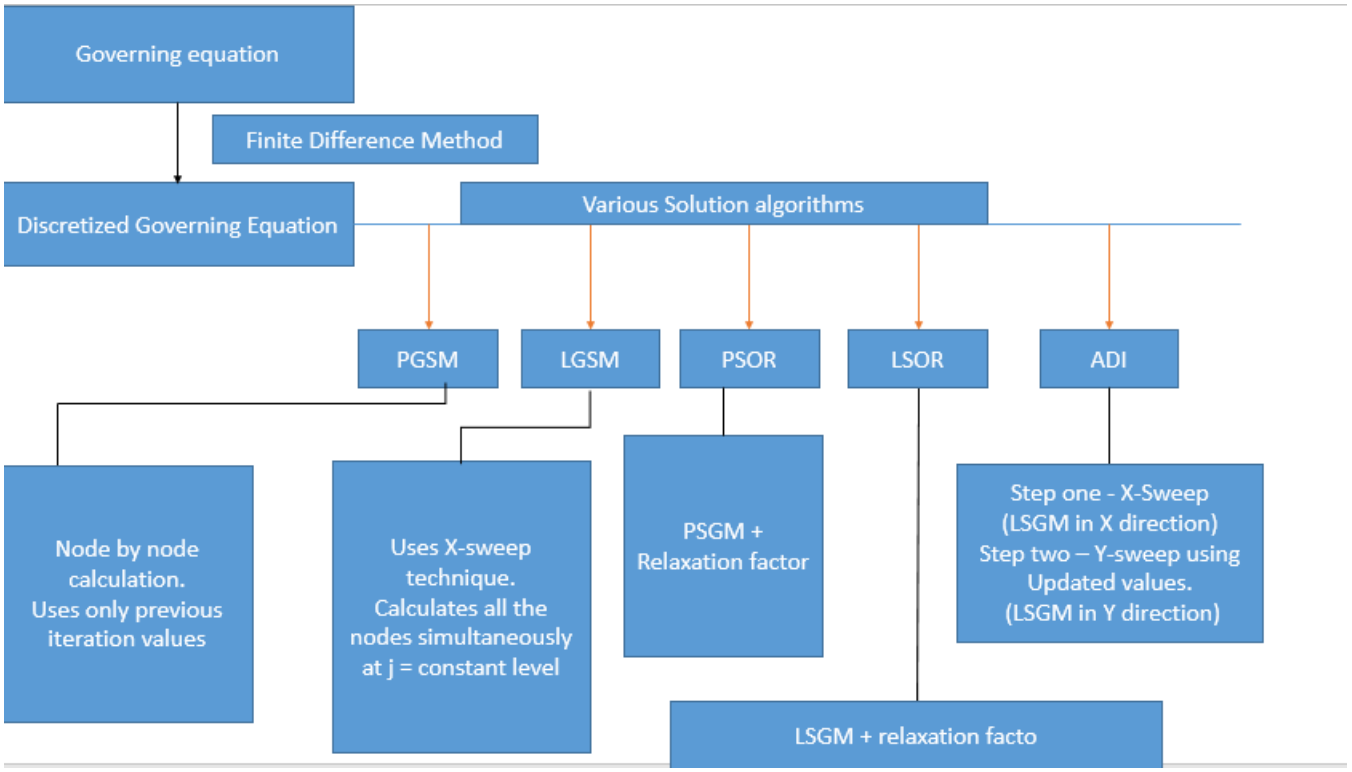

Figure 7: Temperature distribution in case of line successive over relaxation Method.