Amirkabir University of Technology
(Tehran Polytechnic)

Solution of 2D Laplace's equation
Heat conduction through a square plate

Prepared for Advanced Numerical Analysis Course

By

Hossein Sheikh Shoaie

Supervisor

Dr. Behzad Baghapoor

Mechanical Engineering Department

Winter 2023

1 Problem statement

1.1 Equation and boundary conditions

The first law of thermodynamic for a steady state, conduction heat transfer without heat generation reduces to an elliptic partial differential equation. The PDE is called Laplace's equation, for the two dimensional heat transfer in the Cartesian system of coordinates is subjoined.

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

To solve the above equation numerically some steps must be done. First the geometry and boundary conditions must be clear; which is an important part of the problem statement. As mentioned in the subject, geometry is a thin squared plate with unit length, which can be considered as a two dimensional plate.

Two kind of boundary conditions are considered the Dirichlet and Neumann. The temperature at the left side is zero. Also the lower and upper part experience unit temperature, and the heat flux normal to the right side is zero.

1.2 Discretization

The domain and equations must be discretized as the next step. Domain will be discretized to limited number of points, the prescribed grids for this problem constitute of 100, 400, 2500 and 10000 points.

The equations are relate to the Laplace's, for internal nodes, and the boundary conditions equations. The Laplace's equation can be discretized through several methods, here the central second order finite difference (The five points stencil) is considered.

$$a_{i-1,j}T_{i-1,j} + a_{i,j-1}T_{i,j-1} + a_{i,j}T_{i,j} + a_{i+1,j}T_{i+1,j} + a_{i,j+1}T_{i,j+1} = b_{i,j} \tag{2}$$

Where

$$a_{i-1,j} = a_{i+1,j} = 1 \quad ; \quad a_{i,j-1} = a_{i,j+1} = 1 \quad ; \quad a_{i,j} = -2.0(1 + (\frac{\Delta x}{\Delta x})^2) \quad ; \quad b_{i,j} = 0 \tag{3}$$

The distance between two consecutive points in each direction is equal, so

$$a_{i,j} = -4$$

consequently, the discretized equation for internal nodes reduces to

$$T_{i-1,j} + T_{i,j-1} - 4T_{i,j} + T_{i+1,j} + T_{i,j+1} = 0 \tag{4}$$

The left side boundary condition is Nuemann:

$$\frac{\partial T}{\partial x} = 0 \qquad x = 1 \tag{5}$$

which can be discritized by different methods, however the backward first order finite difference method has been considered.

$$a_{i-1,j} = a_{i,j} = 1 \quad ; \quad b_{i,j} = 0 \tag{6}$$

The internal nodes and the boundary condition are discretized by the second and first order of accuracy methods, respectively. So the problem is solved by truncation error with the order lower than two of Δx .

The Dirichlet boundary conditions are

$$T_{i,j} = 0 \qquad x = 0 \quad ; \quad T_{i,j} = 1 \qquad y = 0 \quad ; \quad T_{i,j} = 1 \qquad y = 1$$

The corner points are not contributed in the five point stencil so the temperature of these points in not necessary.

1.3 Solution

Result of the discretization step is a system of linear algebraic equations. Although the mentioned system can be solved by several linear algebra solvers, the iterative fixed-point methods (Gauss-Seidel) , with and without relaxation factor, and the Krylov subspaces methods (GMRES) , with and without pre-conditioner are considered.

1.4 Problem Objectives

1. Write a function that provides A and b for the linear solver.
2. Sketch the sparsity pattern of matrix for nx = 10. Use matplotlib.pyplot.spy and matplotlib.pyplot.show to show the pattern. A sparse matrix pattern should be passed to spy function.
3. Consider the coordinate format, coo matrix. Write a sparse version of the gaussSeidel module of chapter 2 with a relaxation factor . Test the effect of in convergence of your solver by plotting the norm2 of residual r = b − Ax k along with kth iteration. Name your solver as sparse gs [Hint: To extract coo matrix data of matrix A, use row=A.row, col=A.col, and data=A.data].
4. Investigate the convergence of SciPy gmres solver for this problem. To monitor the con- vergence in scipy.sparse.linalg.gmres, one can return the norm2 of residual by setting the ”callback” option.
   - Study the convergence of gmres for the prescribed grid sizes without preconditioning.
   - Add the incomplete LU preconditioner to gmres with spliu and study the convergence for each grid with fill factior=1.0 and fill factor=10.
5. By plotting convergence history (log-log scale) compare convergence rates of Gauss-Seidel and gmres (with and without preconditioner) for the grid sizes and make conclusions.
6. Plot 2D temperature field for the prescribed grid sizes using contourf in matplotlib. You can set the color-map as set cmap('jet'). To provide the contour field, you need to define mesh by meshgrid and reshape the 1D solution output to 2D array by reshape.

2 Methodology

2.1 COO sparse matrix format

To solve a system of linear equations, the coefficient matrix (A) and right hand side (B) must be introduced, at the beginning. Problems with lower level of connectivity between nodes, such as the most of engineering problem has lots of zero element in their coefficient matrices. Which aren't necessary because their multiplication returns zero. Also these elements occupy memory the same as nonzero elements.

There are several method to compact the matrices, note that matrices mustn't defined as dense firstly. Matrix must introduce in an sparse format.

The COO[1] is a method which gets three vector and then use them to solve the system.

- Value vector which contains the value of all nonzero elements.
- Row indices vector, which keeps the row index of each nonzero element.
- Column indices vector, that stores the column number of nonzero elements.

Instead of the dense matrix, sparse one is introduced to the memory. Then solver must be written related to the sparse format.

2.2 Coefficient matrix and right hand side vector definition

First the number of nodes must be evaluated which is equal to

$$nx \times ny$$

Then position of each node must be defined, so that the suitable connectivity appears between nodes. The south-left corner is considered as the 0th node and so on. After that Boundary conditions play the key rule in the array definition. A counter counts the node number then some if conditions are written which find the position of point in the domain. Then related to the boundary conditions and internal nodes coefficients stores the suitable value as the node coefficient.

Mentioned procedure is repeated for the right hand side vector, but not in a sequential (for) loop. The vectorized feature of numpy is employed instead of for loop.

2.3 Gauss - Seidel solver with relaxation

2.3.1 Gauss - Seidel

Gauss - Seidel is an iterative method, which is known as the successive displacement method, is a linear algebra's solver which is suitable for diagonally dominant matrices. A large body of theory exists for the case where the coefficient matrix arises from the finite difference discretization of Elliptic Partial Differential Equations, which guarantee the convergence of current system of equations.

The Gauss-Seidel method employs the evaluated values in each iteration to calculate other values. Means at the (k+1)th iteration the kth iteration amounts in addition to evaluated ones at (k+1)th iteration contribute in the (k+1)th solution.

$$x_i^{k+1} = \frac{1}{A_{ii}}(b_i - \sum_{j=1}^{i-1} A_{ij}x_j^{k+1} - \sum_{j=i+1}^{n} A_{ij}x_j^k) \tag{7}$$

[1]Coordinate format

2.3.2 Relaxation technique

Relaxation technique helps the convergence, in each iteration the new value is a combination of a fraction of the evaluated value at current iteration plus to a fraction of the last iteration value. This fraction is called the relaxation factor (W). The Gauss-Seidel method in addition to the relaxation technique has the below form.

$$x_i^{k+1} = W(\frac{1}{A_{ii}}(b_i - \sum_{j=1}^{i-1} A_{ij}x_j^{k+1} - \sum_{j=i+1}^{n} A_{ij}x_j^k)) + (1-W)x_i^k \tag{8}$$

There are several methods to evaluate the mentioned factor. Three kinds of relaxation exist.

1. W = 1, no relaxation and original formulation is recovered.
2. W < 1: interpolating between old and new iterations (under-relaxation).
3. W > 1, extrapolation or over-relaxation.

In this article the optimum value of the relaxation factor is has not been asked, however the convergence must be studied for different relaxation factors. So all the three kinds are considered.

2.3.3 sparse-gs function

sparse-gs function is a self-written one, which gets the number of nodes at each direction nx or ny, the first guess vector (x0), the required tolerance ϵ and the amount of relaxation factor.

First, the related sparse matrix in COO format and right hand side vector are formed through the self-written sparse-arrays function. Then a while loop is started, which does the iterations. While condition is the tolerance that is defined by the user. If second norm of residual vector become lower than the ϵ while loop breaks.

Inside the while a nested for loop and a single for loop are written. The nested one evaluates solution by means of the Coo format vectors and relaxation factor. Another loop computes the residual. Finally second norm (Euqlid norm) of the residual is calculated, to be compared with the ϵ as while condition.

2.4 GMRES solver with preconditioner

2.4.1 GEMRES

The Generalized Minimum Residual Method (GMRES) is a projection method based on taking $K = K_n$, in which K_n is the n-th Krylov subspace.

The matrix A is assumed to be invertible of size m-by-m. Furthermore, it is assumed that b is normalized, i.e., that $b = 1$.

The n-th Krylov subspace for this problem is

$$K_n = K_n(A, r_0) = span(r_0, Ar_0, A^2r_0, \ldots, A^{n-1}r_0) \tag{9}$$

where $r_0 = b - Ax_0$ is the initial error given an initial guess $x_0 \neq 0$. Clearly $r_0 = b$ if $x_0 = 0$.

GMRES approximates the exact solution of $Ax = b$ by the vector $x_n \in K_n$ that minimizes the Euclidean norm of the residual $r_n = b - Ax_n$.

The vectors $r_0, Ar_0, \ldots A^{n-1}r_0$ might be close to linearly dependent, so instead of this basis, the Arnoldi iteration is used to find orthonormal vectors $q_1, q_2, \ldots, q_n$ which form a basis for K_n. In particular, $q_1 = \|r_0\|_2^{-1} r_0$.

Therefore, the vector $x_n \in K_n$ can be written as $x_n = x_0 + Q_n y_n$ with $y_n \in \mathbb{R}^n$, where Q_n is the m-by-n matrix formed by $q_1, \ldots, q_n$.
The Arnoldi process also produces an $(n+1)$ - by -n upper Hessenberg matrix $\tilde{H}_n$ with $AQ_n = Q_{n+1}\tilde{H}_n$

For symmetric matrices, a symmetric tri-diagonal matrix is actually achieved, resulting in the minres method. Because columns of Q_n are orthonormal, we have

$$\|r_n\| = \|b - Ax_n\| = \|b - A(x_0 + Q_n y_n)\| = \|r_0 - AQ_n y_n\| = \|\beta q_1 - AQ_n y_n\| = \|\beta q_1 - Q_{n+1}\tilde{H}_n y_n\|$$

$$\|\beta q_1 - Q_{n+1}\tilde{H}_n y_n\| = \|Q_{n+1}(\beta e_1 - \tilde{H}_n y_n)\| = \|\beta e_1 - \tilde{H}_n y_n\|$$

where $e_1 = (1, 0, 0, \dots, 0)^T$ is the first vector in the standard basis of $\mathbb{R}^{n+1}$, and $\beta = \|r_0\|$, r_0 being the first trial vector (usually zero). Hence, x_n can be found by minimizing the Euclidean norm of the residual $r_n = \tilde{H}_n y_n - \beta e_1$.
This is a linear least squares problem of size n. This yields the GMRES method. On the n-th iteration:

1. calculate q_n with the Arnoldi method;
2. find the y_n which minimizes$\|r_n\|$;
3. compute $x_n = x_0 + Q_n y_n$;
4. repeat if the residual is not yet small enough.

2.4.2 Preconditioning

The condition number of a function measures how much the output value of the function can change for a small change in the input argument. This is used to measure how sensitive a function is to changes or errors in the input, and how much error in the output results from an error in the input. preconditioning is the application of a transformation, called the preconditioner, that conditions a given problem into a form that is more suitable for numerical solving methods. Preconditioning is typically related to reducing a condition number of the problem. Preconditioning aims to transform a given problem $Ax = b$ into a form $A'x = b'$.
The original coefficient matrix A is typically changed to $A' = M^{-1}A$. To develop a priconditioner two major steps are considered:

- Choosing a suitable preconditioning matrix M, that is done by selecting the preconditioning method. Some preconditioning methods are Gauss-Seidel, Jacobi, Successive over-relaxation (SOR) and Incomplete LU decomposition (ILU) . The priconditioning matrix M is different in each method.

  The preconditioning matrix must satisfy the below conditions

  1. Solving the linear subsystem Mx = y needs to be inexpensive.
  2. M should be close to A in some sense and clearly be non-singular.

- Selecting an efficient technique to solve the preconditioning subsystem.

2.4.3 Incomplete LU decomposition pre-conditioner

Complete LU decomposition considers the full original sparse matrix not only the nonzero elements. So the LU production results in the original matrix. However the incomplete matrix considers only the nonzero elements of the original matrix. Where the LU production results in a denser matrix than the original sparse one. If decomposed matrix is the original matrix (A), the Incomplete LU decomposition is from zero order. The order growth up by multiplication of the original matrix by itself, for example ILU(1) relates to decomposition of (A^2) and so on. The higher order consequent in more accurate solution, but has larger computational cost. So as a pre-conditioner lower orders are acceptable.

2.4.4 Introduce Incomplete LU preconditioner to scipy.sparse.linalg.gmres

The coefficient matrix in sparse COO format and right hand side vector are formed by the self-written Sparse-arrays function. After that the Incomplete LU decomposition function (spilu) is called, which factorizes the sparse coefficient matrix. The spilu gets the coefficient matrix and fill-factor as arguments. The fill-factor specifies the fill ratio upper bound,which is larger or equal to unit for ILU, also the default value is equal to 10.

Then the resulted subsystem must be solved. A funnction is defined which solves the subsystem by solve attribution from scipy.sparse.linalg. The defined function can't be given to the gmres solver as an argument, instead must be called as a linear operator. This is done by means of LinearOperator function which gets the shape of the decomposed matrix and the defined solver as argument.
finally, The defined linear operator is called in gmres through the M argument.

3 Codes, results and discussions

3.1 Problem.NO:1

The subjoined function creates the coefficient matrix in COO sparse format and right hand side vector. The array-creator function has been written according to the current problem and related boundary conditions.

The only argument of array-creator is the number of nodes at each direction, which is equal to n. Other explanation are contained in the code body as comments.

Listing 1: Function which provides A , B for linear solver

```
# The necessaty function are imported from the python libraries.

# zeros : creates arrays with all zero elements.

# Remainder : returns the remainder of a division.

# coo_ matrix : creates a coo format sparse matrix by 3 vectors.

from numpy import zeros,remainder
from scipy.sparse import coo_matrix


 ''' Self_defined array_creator functions has been written to
  create the coefficient matrix and right hand side vector.
  The coefficient matrix is form in COO sparse format.'''

  The only argument of this function is the number of nodes in each
  direction (n).

def array_creator(n):

# The following lines create vectors which store the value, the row number
and the column number of nonzero elements.

AA = zeros(5*n**2 - 15*n + 16) #data
IA = zeros(5*n**2 - 15*n + 16) #row number
JA = zeros(5*n**2 - 15*n + 16) #column number

# k is a counter which counts the number of each node.
k = 0

# Two nested for loops has been written to specify the value, row number and
column number of each nonzero element of coefficient matrix.

# First loop counts the rows.
for i in range( 0,(n**2) ):

# Inner loop counts the columns.
 for j in range( 0,(n**2) ):

# If condition sentence checks if i and j are related to the inner nodes.

   if i > n and i < (n-1)*n -1 and remainder(i,n) != 0 and remainder(i,n) != n-1 :
    # Saves the coefficient of node.
    if   j == i    : AA[k] = -4 ; IA[k] = i ; JA[k]= j; k +=1 ;continue

    # Saves the coefficient of left node.
    elif j == i-1 : AA[k] =  1 ; IA[k] = i ; JA[k]= j; k +=1 ;continue

    # Saves the coefficient of lower node.
```

```
        elif j == i-n : AA[k] =  1 ; IA[k] = i ; JA[k]= j; k +=1 ;continue

        # Saves the coefficient of right node.
        elif j == i+1 : AA[k] =  1 ; IA[k] = i ; JA[k]= j; k +=1 ;continue

        # Saves the coefficient of upper node.
        elif j == i+n : AA[k] =  1 ; IA[k] = i ; JA[k]= j; k +=1 ;continue

# Stores the coefficient of lower boundary nodes (Dirichlet).
    elif i < n-1 and i == j:
        AA[k] = 1 ; IA[k] = i ; JA[k]= j; k +=1 ;continue

# Stores the coefficient of upper boundary nodes (Dirichlet).
    elif i > n**2 -n and i < n**2-1  and i ==j:
        AA[k] = 1 ; IA[k] = i ; JA[k]= j  ; k +=1 ;continue

# Stores the coefficient of left boundary nodes (Dirichlet).
    elif remainder(i,n) == 0:
      if i == j :
        AA[k] = 1 ; IA[k] = i ; JA[k]= j  ; k +=1 ;continue

# Stores the coefficient of right boundary nodes (neumann).
    elif remainder(i,n) == n-1 and i == j :

        # related to node
        AA[k] = -1 ; IA[k] = i ; JA[k]= j ;  k +=1

        # related to left node
        AA[k] =  1 ; IA[k] = i ; JA[k]= j-1 ; k +=1 ;continue

# The value, row and column number vectors are already defined.
# Next line creates the coefficient matrix in sparse COO format.
 Asp = coo_matrix((AA, (IA, JA)), shape=(n**2, n**2))


# The following lines create the right hand side vector.

 b = zeros(n**2)

 # lower left corner
 b[0]=0

 # The lower edge nodes.
 b[1:n-1] = 1

 # lower left corner node.
 b[n-1]=0

 # The upper edge nodes.
 b[n**2-n + 1:n**2-1] = 1

 # The uppre left corner node.
 b[n**2 -n] = 0

 # upper right corner node.
 b[n**2 -1] = 0

 # The function outputs are returned by the next line.
 return Asp , b
```

The result of the written function is shown in next subsection.

3.2 Problem.No:2

The sparsity pattern of a 2D array is plotted through the mtplotlib.pyplot.spy which visualizes the nonzero elements of array. The following code plots the sparsity pattern of a 10 x 10 nodes grid.

Listing 2: Plot the sparsity pattern

```
# The pyplot subpackage from matplotlib liberary is imported.
import matplotlib.pyplot as plt

# The linspace attribution is imported from numpy,which divides
# an interval into the same smaller intervals.
from numpy import linspace

# a , b arrays are created by the self-defined array-creator function.
a , b = array-creator(10)

# The spy attribution is called and the arguments are tuned.
plt.spy(a,mec = 'black',mfc = 'red',precision=0, markersize = 5)

# The following line set the plot appearances.

plt.title('Sparsity␣pattern␣of␣a␣10␣x␣10␣grid')
plt.xticks(linspace(0,100,11))
plt.yticks(linspace(0,100,11))
plt.ylabel('Row␣number')
plt.xlabel('Column␣number')
```

Resulted pattern is observable in figure 1.

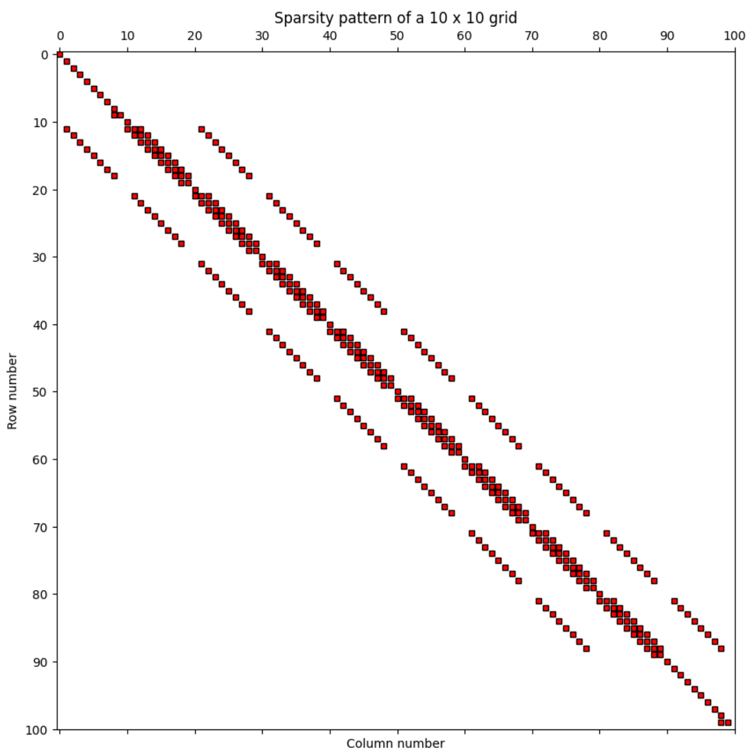


Figure 1: The sparsity pattern of coefficient matrix related to 10 x 10 grid.

3.3 Problem.No:3

sparse-gs function consists of two phase, first computes the solution and second evaluates the residual and Euqlid norm of that. The iterations are done in a while loop which stops when the second norm of residual is less than the ϵ value ($\epsilon = 10^{-5}$).

Listing 3: The Gauss-Seidel solver with relaxation technique

```
# List of necessary attributions:

# where : finds the indices of a value in an array.

# asarray : converts an input to array.

# dot : does the inner product of vectors.

# sqrt : evaluates the second root of a value.

from numpy import array,zeros,remainder,where,asarray,dot
from math import sqrt

def sparse_gs(n,x0,epsilon=10**(-5), w = 1,):

''' sparse-gs function does the solution of a linear system of algebraic
equations through the Gauss-Seidel method with relaxation technique.

The function gets number of node in each direction (n), the first geuss
vector(x0), the required second norm of residual and the relaxation factor.

Also returns three vectors, iteration number vector, corresponding second
norm vector and the solution as output.
'''
# Create the a and b by self-defined array-creator.
 a , b = array_creator(n)

 # Exteract three vector from the coo sparse format a

 aa = a.data #values
 ai = a.row  #row numbers
 aj = a.col  #column numbers

 # Equal the solution vector to the first geuss.
 x  = x0

 # Create a vector to store the residual.
 r  = zeros(n**2)

 # Pre-definition of iteration number vector and corresponding second norm vector.
 tolvec = zeros(2000)
 itervec = zeros(2000)

 # Let the start second norm equal to 1, which starts the while loop.
 tol = 1

 # set counter zero.
 counter = 0

 # start the while loop.
 while(tol>10**(-5)):


  counter += 1
  itervec[counter-1] = counter

# The solution is done throough this for loop, where condition sentences
# find the position of the node.

  for i in range(0,n**2):
      if i > n and i < (n-1)*n -1 and remainder(i,n) != 0 and remainder(i,n) != n-1 :

     # t stores indices of the row number vector which are equal to the node number.
        t = asarray(where(ai == i))

     # the solution for internal nodes is done through below line.
     # note aa[t(0,0) till aa[(0,5)]] are the coeffitient of the i-th equation, which
     # are exteracted from the value vector by their indices that are extracted by the
     # where() attribution.

        x[i] = w * ( b[i] - x[i-n] * aa[t[0,0]] - x[i-1] * aa[t[0,1]] - x[i+1] * aa[t[0,3]] -\
        x[i+n] * aa[t[0,4]] )/aa[t[0,2]] + (1-w)*x[i]

     #Mentioned steps are repeated for boundary nodes in the following:

     #lower edge
      elif i < n-1 and i > 0 :
        t = asarray(where(ai == i))
        x[i] = (b[i] /aa[ai[t[0,0]]])

     # upper edge
      elif i > n**2 -n and i < n**2-1 :
        t = asarray(where(ai == i))
        x[i] = b[i]/aa[t[0,0]]

     # left edge
      elif remainder(i,n) == 0:
        t = asarray(where(ai == i))
        x[i] = b[i]/aa[t[0,0]]

     # right neumann
      elif remainder(i,n) == n-1 :
        t = asarray(where(ai == i))
        x[i] = w * ( (b[i]- aa[t[0,1]] * x[i-1])/ aa[t[0,0]]) + (1-w) * x[i]

  # The solution at this iteration is done, the residual is available.
  # For this purpose another for loop starts.
  for i in range(0,n**2):

    # Again conditional sentences specify the position of point, then by where() attribution
    # the Indices are exteracted and the residual is evaluated for each node.

    # internal nodes
     if i > n and i < (n-1)*n -1 and remainder(i,n) != 0 and remainder(i,n) != n-1 :
        t = asarray(where(ai == i))
        r[i] = b[i] - ( x[i-n] * aa[t[0,0]] + x[i-1] * aa[t[0,1]] + x[i] * aa[t[0,2]] + x[i+1] * \
        aa[t[0,3]] + x[i+n] * aa[t[0,4]])

    # lower edge
     elif i < n-1 and i > 0 :
        t = asarray(where(ai == i))
        r[i] =  b[i] - x[i] * aa[t[0,0]]

    # upper edge
     elif i > n**2 -n and i < n**2-1 :
        t = asarray(where(ai == i))
        r[i] = b[i] - x[i] * aa[t[0,0]]
```

```
            # left edge
            elif remainder(i,n) == 0:
                x[i] = 0

            # right neumann
            elif remainder(i,n) == n-1 :
                t = asarray(where(ai == i))
                r[i] = b[i] - x[i] * aa[t[0,0]]  - x[i-1] * aa[t[0,1]]

        # The residual has been already evaluated, then the second norm is evaluated by next line.
        tol = sqrt(dot(r,r))

        # The evaluated second norm is stored in corresponding vector, which is going to
        be plotted later.
        tolvec[counter-1] = tol

    # Return outputs.
    return itervec, tolvec, x
```

To Test the effect of relaxation factor in convergence of the solver, three relaxation factors are considered 0.5, 1, 1.5 which reperesent under relaxation, no relaxation and over relaxation, respectively. A grid with nx = ny = 5 points is considered to have a suitable plot. Then the second norm of each iteration is plotted against the iteration number. Plot is illustrated in figure 2. Clearly, the under relaxation technique is not suitable for this problem, because the rate of convergence decreases. How ever the over relaxation is approximatly better than no relaxation. One can geuss the optimum relaxation factor is more than unit, means the over relaxation technique works here. From another point of view the Gauss-Seidel method shows a lower rate of convergence at primary iterations, then suddenly the rate increases.

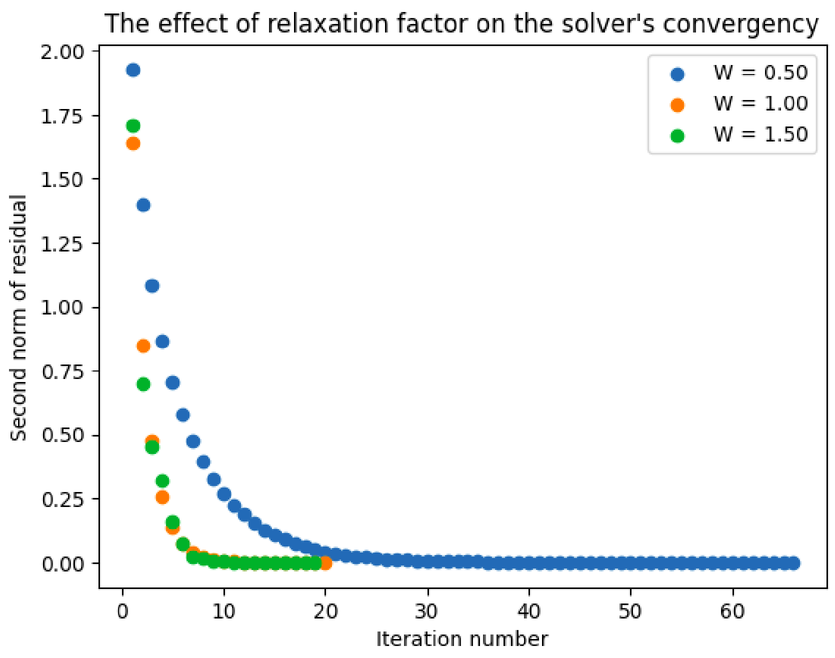


Figure 2: The sparsity pattern of coefficient matrix related to 10 x 10 grid.

3.4 Problem.No:4

3.4.1 Convergence of gmres without preconditioning

The scipy.sparse.linalg.gmres solves the linear system through the Generalized minimum residual method. The function arguments are

scipy.sparse.linalg.gmres$(A, b, x0 = None, tol = 1e{-}05, M = None, callback = None, callback_type)$

A = The real or complex N-by-N matrix of the linear system. Alternatively, A can be a linear operator which can produce Ax using, e.g., scipy.sparse.linalg.LinearOperator.
b = Right hand side of the linear system.
x0 = Starting guess for the solution (a vector of zeros by default).
tol = Tolerance for convergence.
M = Linear operator, which solves the preconditioning subsystem (is none by default).
callback = User-supplied function to call after each iteration. where args are selected by callback-type.
callback-type : ('x', 'pr-norm', 'legacy'), optional
Callback function argument requested: x: current iterate (ndarray), called on every restart.
pr-norm: relative (preconditioned) residual norm (float), called on every inner iteration.
Also outputs are
x = The converged solution.
info = int, Provides convergence information:
0 : successful exit
>0 : convergence to tolerance not achieved, number of iterations
<0 : illegal input or breakdown
The gmres solver and necessary arguments have been introduced already. To monitor the gmres convergence, a function is added to code which is subjoined below.

Listing 4: Convergence monitoring of gmres, without preconditioning

```
from numpy import array
from scipy.sparse.linalg import gmres

''' counter function prints the iteration number and corresponding
second norm at the end of each iteration.

All the printed outputs are stored in corresponding .CSV format file.
in the end the dataset is converted to numpy arrays and the convergence history
is plotted.'''

def counter(rk=None):
counter.niter += 1
print("{:3d},{}".format(counter.niter, str(rk)))

# nodes stores the number of points, related to each grid size.
nodes = array([10,20,50,100])

# The solution of all prescribed grids is done through a sequential for loop,
and the convergence history is printed by the callback argument of gmres solver.

for i in range(4):

# For each grid the coefficient sparse array and right hand side are created by
#array-creator.
a , b = sparse-creator(nodes[i])
counter.niter = 0
gmres(a,b,callback = counter )
```

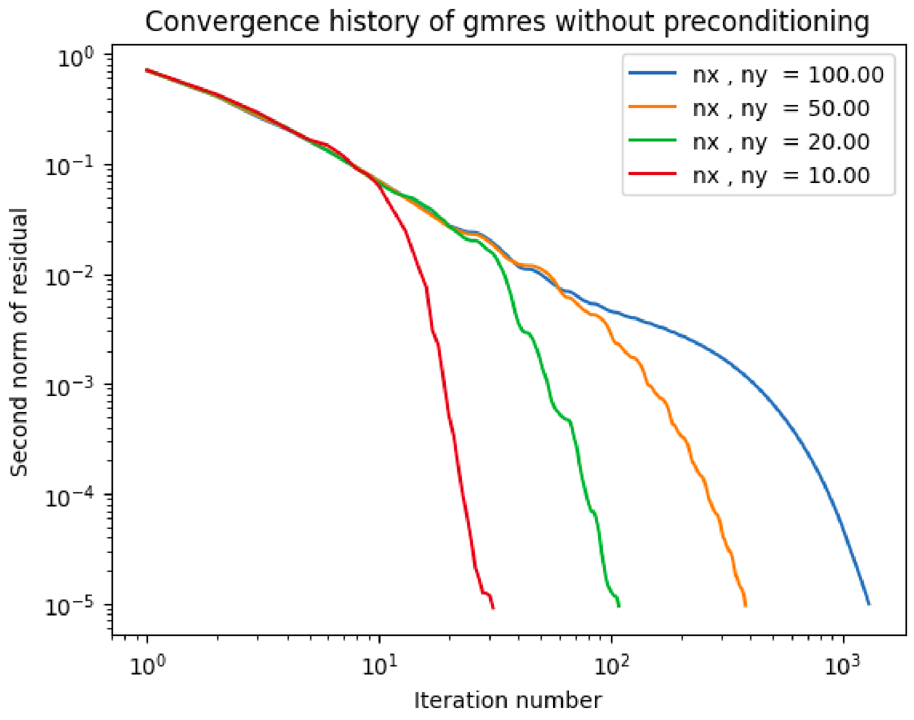


Figure 3: Convergence history of gmres, without preconditioning.

Convergence history is presented in figure 3. Evidently the gmres second norm behaves exponentially related to the iteration number. At the lower iterations the convergence rate is low then suddenly the rate increases sharply. If a curve is fitted, in the beginning the exponent is a small negetive value but in the end is a large one.

3.4.2 Convergence of gmres with preconditioning

As were discussed in the methodology section, to introduce the preconditioner to gmres, first the subsystem solver which is defined as a function is converted to a linear operator. After that the created linear operator will be called as an argument of gmres solver. The process is implemented in the following code.

Listing 5: Convergence monitoring of gmres, with incomplete LU preconditioning

```
# Necessary attributions are

#spilu : spilu is an attribution from the scipy.sparse.linalg subpackage, which decomposes a
#matrix to L and U part by incomplete decomposition method. Note that the input matrix must
#be in Compressed sparse column (CSC) format, otherwise the spilu converts the matrix into
#the required format.
#Another argument of splu is the fill-factor which has been defined before.

# LinearOperator : gets the shape of the coefficient matrix and the subsystem's solver as
#arguments, and returns an operator which is callable in gmres arguments.
from scipy.sparse.linalg import spilu,LinearOperator

# Below loop does the solution for each grid.
```

```
for i in range(4):
a , b = Sparce_arrays(nodes[i])

# The incomplete decomposition of the coeffecient matrix.
M2 = spilu(a,fill_factor=1) # fill ratio upper bound

# define a function to solve the subsystem.
M_x = lambda x: M2.solve(x)

# Convert the solver to a linear operator.
M = LinearOperator(((nodes[i])**2,(nodes[i])**2), M_x)

counter.niter = 0

# Solve the problem by calling the defined preconditioner operator.
x1 = gmres(a,b,M=M,callback= counter,tol=10**(-5))
```

Figure 4 displays convergence history of gmres solver plus incomplete LU decomposition with unit fill factor as preconditioner. Generally, trend of second norm is similar to gmres solver, how ever the required iterations for the same tolerance are an order of magnitude lower than gmres.

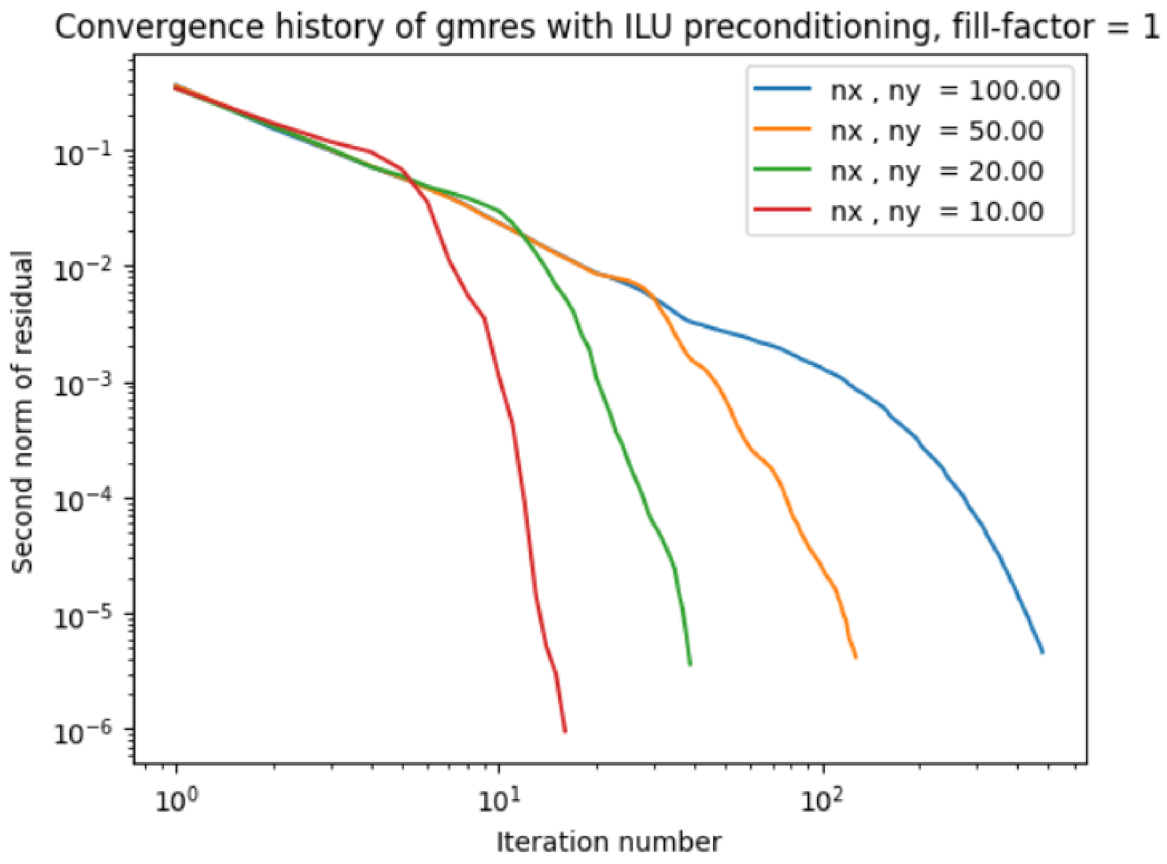


Figure 4: Convergence history of gmres plus ILU with f=1 preconditioning.

The convergence history of gmres solver plus incomplete LU decomposition , with fill factor equal to ten, as preconditioner is presented in figure 5.

For the grids with significant number of nodes the trend is like the examined gmres solvers, and

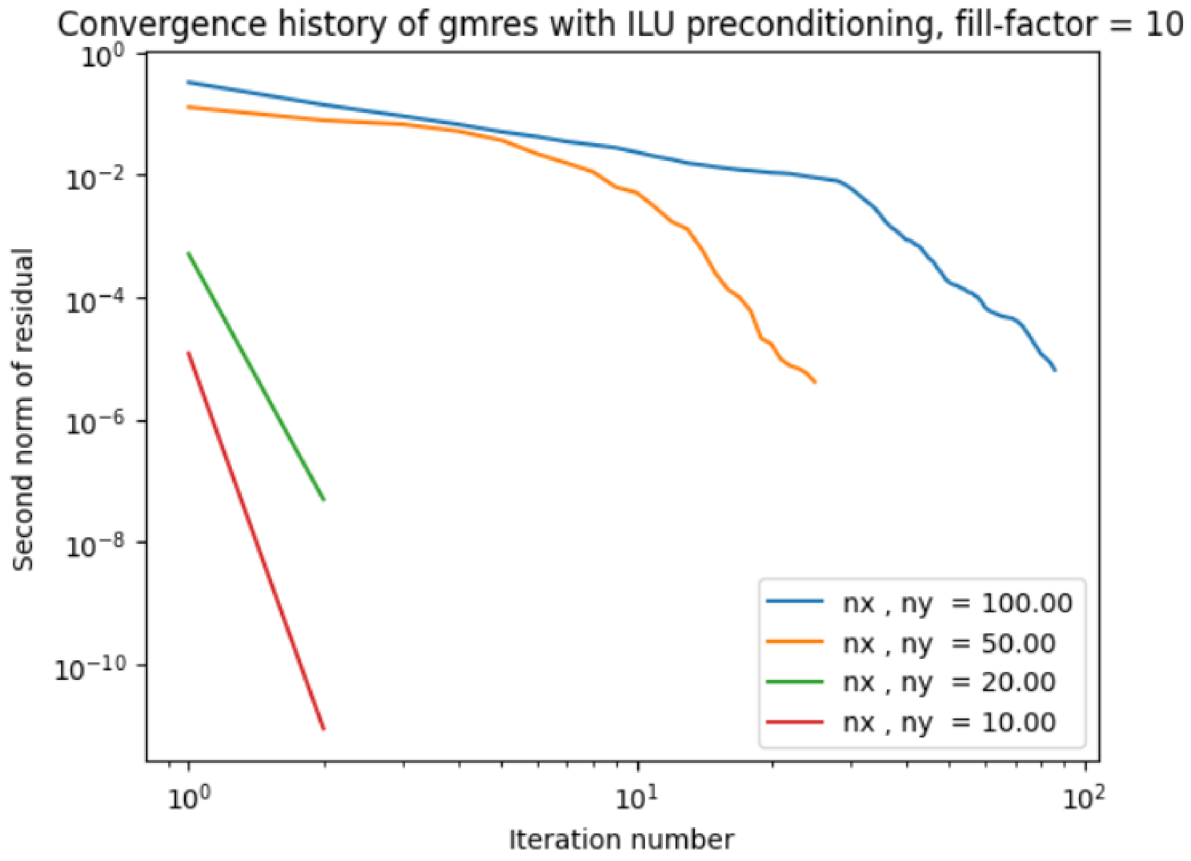


Figure 5: Convergence history of gmres plus ILU with f=10 preconditioning.

an the required tolerance is achieved by an orther of magnitude lower iteration numbers related to priconditioning with fill factor equal to 1. However for grids with lower node number the trend is different, means the exponent is a large number and there is no slow convergence rate in the beginning.

The second norm criteria is reached in two iterations for grids with ten and twenty nodes. This high convergence rate is because of the large fill factor; when the fill-factor is large the preconditioner subsystem is close to the main system. Means in each iteration the system is solved twice.

3.5 Problem.No:5

To plot the concvergence history, the below code is run, which plots the last problem's dataset for gmres solver. Also for Gauss-Seidel convergence history the self-defined sparse-gs function is called, then the related history is plotted for different grid sizes.

Listing 6: Plot convergence history of different solvers for pre-scribed grids

```
# Necessary subpackages are

# pyplot from matplotlib library, that plots the results.
from matplotlib import pyplot as plt

# genfromtxt attribution of numpy library which creates an array from excel .csv file.

# array creates an array from given elements.

from numpy import genfromtxt, array

# To store the iteration number and corresponded second norm, a large zeos vector is
#considered, so in the end lots of zero elements will be remained. Below function is written
#to plot just the nonzero elements of vectors.

def zero_to_nan(values):
"""Replace every 0 with 'nan' and return a copy."""
return [float('nan') if x==0 else x for x in values]

# Node number of each grid is collected from below vector, through the for loop.
nodes = array([10,20,50,100])

# The figure size is set by
plt.figure(figsize=(10,9))

# The for loop computes then plots the second norm of each eteration by sparse-gs function.
# on the prescribed grids, without relaxation factor.
for i in range(4):
itervec ,tolvec, x = sparse_gs(nodes[3-i],w=1)

# loglog attribution of the pyplot subpackage, plots data in logarithmic scale in both directions.
plt.loglog(zero_to_nan(itervec),zero_to_nan(tolvec),\
label='nx␣,␣ny␣␣=␣{0:.2f}.␣Gauss-Seidel␣W=1'.format(nodes[3-i]))

# The following code sections plot the gmres without relaxation and gmres plus incomplete LU
#decomposition with fill factors equal to 1 and 10, respectively. The dataset has been
#copied from the printed reports of the gmres solver's callback function. Which were
#provided by the gmres iteration monitoring function (counter).

gmres = genfromtxt('qu4100.csv', delimiter=',')
for i in range(4):
plt.loglog(gmres[:,2*i],gmres[:,2*i+1],\
label='nx␣,␣ny␣␣=␣{0:.2f}.␣GMRES'.format(nodes[3-i]))


gmres_plus_ilu1 = genfromtxt('qu4ilu1.csv', delimiter=',')
for i in range(4):
plt.loglog(gmres_plus_ilu1[:,2*i],gmres_plus_ilu1[:,2*i+1],\
label='nx␣,␣ny␣␣=␣{0:.2f}.␣GMRES␣+␣ILU␣f=1'.format(nodes[3-i]))


gmres_plus_ilu10 = genfromtxt('qu4ilu10.csv', delimiter=',')
for i in range(4):
plt.loglog(gmres_plus_ilu10[:,2*i],gmres_plus_ilu10[:,2*i+1],\
label='nx␣,␣ny␣␣=␣{0:.2f}.␣GMRES␣+␣ILU␣f␣=10'.format(nodes[3-i]))
```

```
# Next lines set the figure appearances.

plt.legend(loc='lower␣right')
plt.title('Convergence␣history␣of␣gmres,␣wich␣and␣without␣preconditioner,␣and␣Gauss−Seidel')
plt.xlabel('Iteration␣number')
plt.ylabel('Second␣norm␣of␣residual')
```

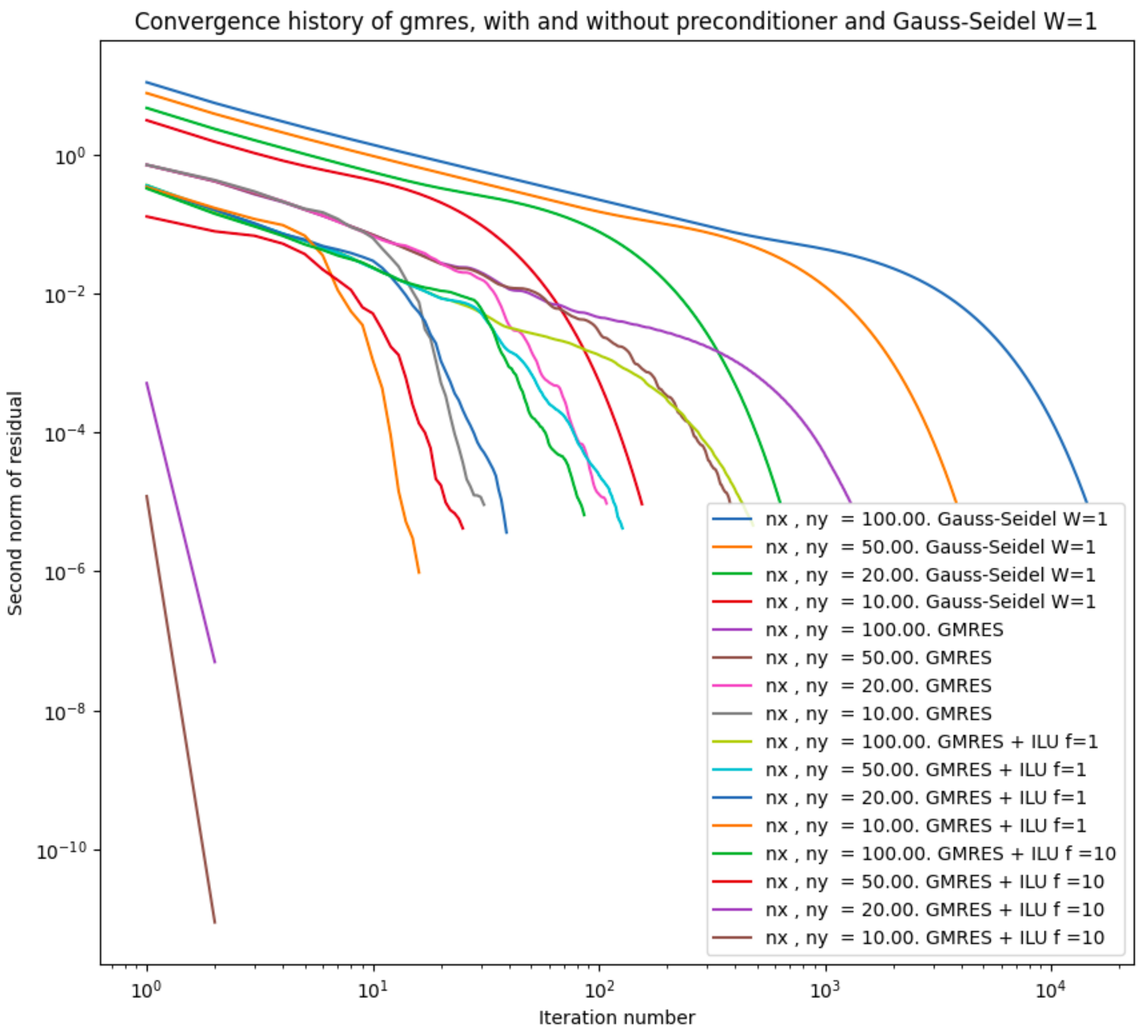


Figure 6: Convergence history of gmres, without preconditioning and Gauss-Seidel without relaxation.

Figure 6 provides a good comparison between the mentioned solvers. Evidently the lowest iteration numbers to accomplish the considered second norm for each grid belongs to the gmres plus ILU preconditioning, when fill factor is equal to 10.

By considering the largest grid, the required iterations in Gauss-Seidel method is an order of magnitude larger than the gmres without preconditioning. Also for the gmres, number of iterations is an order of magnitude larger than the gmres plus ILU preconditioner with fill factor equal to one and so on. So between the considered solvers gmres plus ILU preconditioner with fill factor equal to ten, needs lower iterations number.

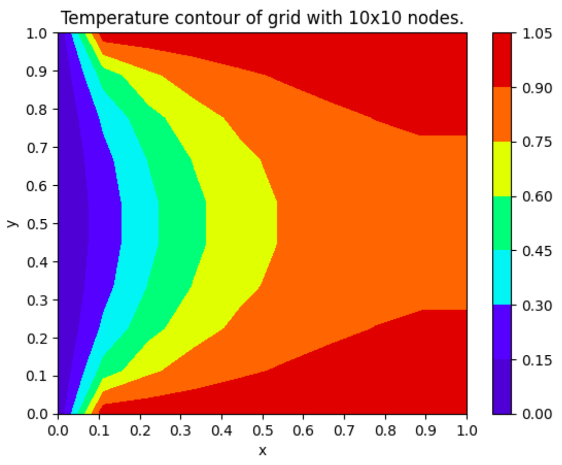
The convergence trend is the same for both gmres and Gauss-Seidel, convergence rate is low at primary iterations, then increases sharply.

3.6 Problem.NO:6

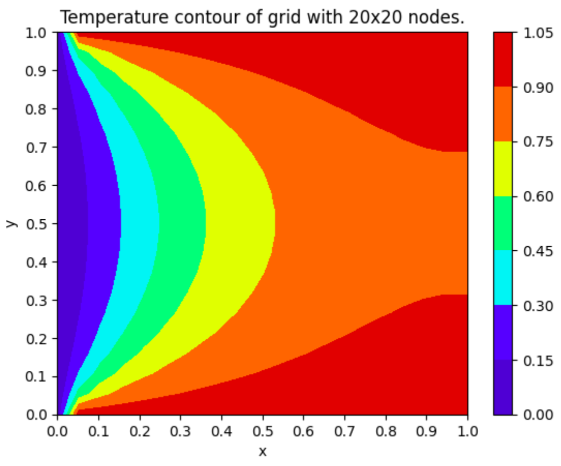
To plot the temperature contour, the problem is solved again through the fastest examined method (gmres plus incomplete LU decomposition with fill factor equal to 10), then the corresponded result are plotted. Code and related explanations are subjoined.

Listing 7: Plot the evaluated temperature contour of pre-scribed grids

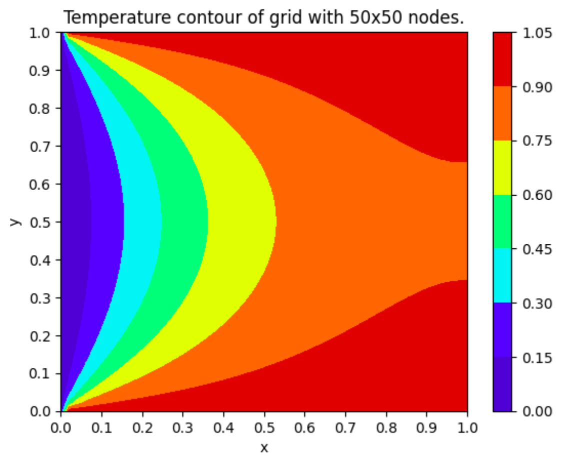
```
# All the necessary attributions unless meshgrid has been explained before.

# meshgrid(): Return coordinate matrices from coordinate vectors. The coordinate
# matrix is applicable in contour plots.


from scipy.sparse.linalg import spilu,LinearOperator,gmres
from matplotlib import pyplot as plt
from numpy import linspace,copy,meshgrid

# nodes array keeps the node numbers in each directions of prescribed grids.
nodes = array([10,20,50,100])

# The solution of each grid is evaluated, then plotted by the next for loop.
for i in range(4):

#Coefficient coo format matrix and right hand side vector are created by
# the array-creator function.
a , b = array-creator(nodes[i])

# Incomplete LU decomposition with fill factor equal to 10, is done by
#next line.
M2 = spilu(a,fill_factor=10)

# M_x is a function which solves the preconditioning subsystem.
M_x = lambda x: M2.solve(x)

# The function is converted to a linear operator through next line.
M = LinearOperator(((nodes[i])**2,(nodes[i])**2), M_x)

# Eventually, the solution is done by the following line.
x2 , r = gmres(a,b,M=M)

# Two next lines introduce the x and y direction vectors; to generate
# a 2D domain for contourf attribution.
x = linspace(0,1,nodes[i])
# A deep copy!
y = copy(x)

# A 2D coordinate matrix is generated by the meshgrid() attribution of numpy.
X,Y = meshgrid(x,y)

# Temperature matrix stores the evaluated value; to be plotted by contourf.
# The evaluate temperatures are located in mentioned matrix through a for loop.
T = zeros((nodes[i],nodes[i]))
for j in range(nodes[i]):
T[j,:] = x2[j * nodes[i]: (j+1)*nodes[i] ]

# Finally the contourf attribution of pyplot subpackage plots the temperature
# distribution on the generated domain.
plt.contourf(X,Y,T,cmap='jet')

# Countor appearances are set by the last lines.
plt.title('Temperature␣contour␣of␣grid␣with␣{}x{}␣nodes.'.format(nodes[i],nodes[i]))
plt.xlabel('x')
plt.ylabel('y')
plt.xticks(linspace(0,1,11))
plt.yticks(linspace(0,1,11))
plt.colorbar()
plt.show()
```



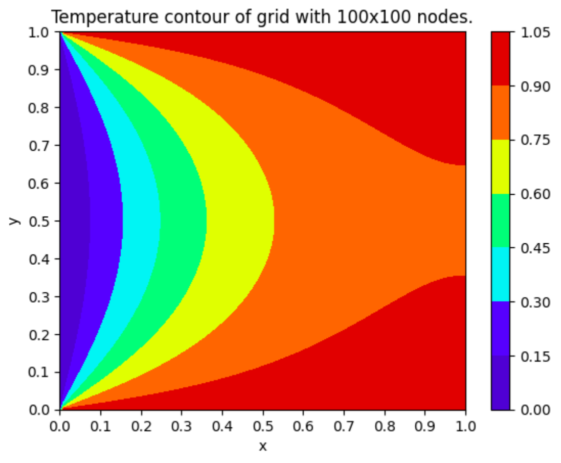
(a) 10 × 10



(b) 20 × 20



(c) 50 × 50



(d) 100 × 100

Figure 7: Temperature contour of prescribed grids.

Temperature contours of prescribed grids are illustrated in figure 7. Clearly, the best contour belongs to grid with higher node number. Difference between results is observable by considering the right corner sides of contours. The low nodes grids can't capture the distribution well, however the solution time is lower for them.

4 Conclusions

- For the solved problem, between the considered solvers the gmres plus ILU preconditioning converges faster than others, specially when the fill factor is equal to ten.
- The required iteration number to acheive the given socond norm for gmres solver is an order of magnitude lower than Gauss-Seidel. Also this number for gmres with considered ILU preconditioner (f = 1) is an order of magnitude lower than gmres. Finally the gmres with ILU priconditioning (f=10) converges with an order of magnitude lower iteration number related to the same solver with unit fill factor (f = 1).
- Second norm of all the solvers shows an exponetial trend with iteration number. Primary iterations experience slower convergence rate, then the rate increases sharply.
- For this problem the over-relaxation Gauss-Seidel method is better than under-relaxation one.
- The grid independency study has not be done, but according to the plotted temperature contours, the low nodes grids can't visualize the temperature distribution accurately.