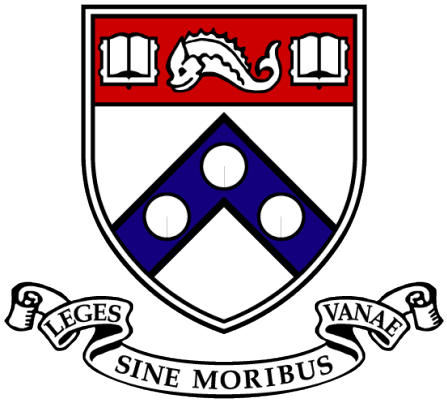
Report on

Assignment 2: Solving System of Linear Equations using Different Solving Approaches

Submitted in partial fulfillment of the requirements of
ENM 502
Numerical Methods and Modeling
by

Jalaj Maheshwari
M.S.E., Mechanical Engineering and Applied Mechanics



University of Pennsylvania
School of Engineering and Applied Sciences

4th March, 2016

Table of Contents

Table of Figures

1. Introduction

The problem statement of this particular project deals with solving the diffusion problem represented by Equation 1 using different iterative schemes and juxtaposing the schemes by running test cases.

$$-\nabla^2 u = \sin(2\pi x) * \sin(2\pi y) \tag{1a}$$

$$u(x, y) = 0 \text{ on all boundaries } (\partial D) \tag{1b}$$

$$D = (0 \le x \le 1) \cup (0 \le y \le 1) \tag{1c}$$

The equation and boundary conditions are first discretized using a finite difference approximation. The MATLAB code for solving the system of equations resulting from the boundary value problem using four different solvers, namely the LU Decomposition solver, Jacobi and Gauss-Seidel iterative solvers and the Newton's Method solver is written.

Test cases are run for each solver for different grid sizes and contour maps are generated to depict the variation of the solution of the boundary value problem over the domain. Further, the solution error defined as a function of the grid resolution is plotted to give an idea of the scaling of the error with each solver.

Finally, the computational scaling of the Jacobi and the Gauss-Seidel iterative solvers is compared to the standard LU decomposition solver and the stability of the two iterative schemes is commented on.

These solving techniques which give us a solution for the boundary value problem are looked upon in this project and are compared to each other and the variation with the increase in grid resolution is seen.

2. Problem Setup and Formulation

The boundary value problem represented by Equation 1 needs to be discretized before it can be used further in any solver. This is done using the finite difference approximation. The domain is divided into smaller finite elements with the numbering shown as in Figure 1.

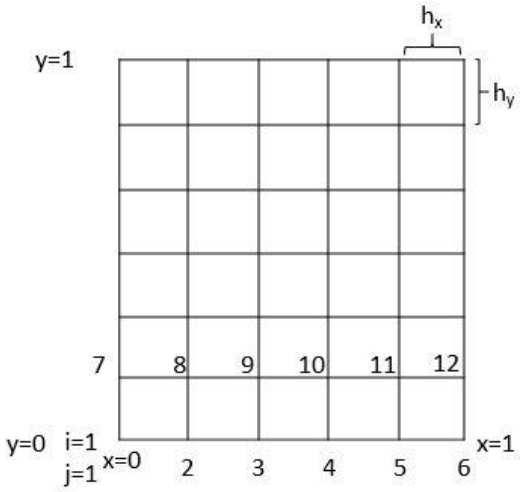


Fig 1: Computational grid generated over domain

Starting from the bottom left corner, the grid points are numbered in the order shown. Here, h_x and h_y represent the grid spacing in the *x* and *y* directions. Any boundary value problem described as Equation 2 can be solved using forward and the backward difference approximation for the first derivative in the Taylor series expansion.

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = f(x, y) \tag{2}$$

Thus, on applying the Taylor series expansion, we get:

$$f'(X_i) = \frac{f(X_{i+1}) - f(X_i)}{h} + 0(h) \tag{3a}$$

$$\text{and } f''(X_i) = \frac{f(X_{i+1}) - 2f(X_i) + f(X_{i-1})}{h^2} + 0(h^2) \tag{3b}$$

where, *I* represents the point at which we calculate out solution. Returning to the PDE, in terms of the grid point being considered, the equation turns to:

$$\frac{u_{i+1,j} - 2u_{ij} + u_{i-1,j}}{h_x^{\,2}} + \frac{u_{i,j+1} - 2u_{ij} + u_{i,j-1}}{h_y^{\,2}} = f_{ij} \tag{4}$$

Thus, we write the equation for each point, giving *n* equations in *n* unknowns which can be represented in the form:

$$Au = f \tag{5}$$

The matrices in Equation 5 are then used in individual solvers to find the solution to the boundary value problem. For each solver, an initial guess for the value of *u* is taken. In our case, we take this value to be a zero vector.

For the Jacobian solver, the next value of *u* over each new iteration *k* is given by:

$$x_i^{k+1} = \frac{-\sum_{j \neq i} a_{ij} x_j^{\,k} + b_i}{a_{ii}} \quad \text{for } i=1,2,\ldots,n \tag{6}$$

Similarly, for the Gauss-Seidel iterative solver, the previous guess and the new value of *u* are used to evaluate the value at the next iteration. It is represented by Equation 7:

$$x_i^{k+1} = \frac{-\sum_{j=1}^{i-1} a_{ij} x_j^{\,k+1} - \sum_{j+1}^{n} a_{ij} x_j^{\,k} + b_i}{a_{ii}} \quad \text{for } i=1,2,\ldots,n \tag{7}$$

For the Newton's method, there are two steps that are involved in computing the solution:

$$\underline{\underline{J}}\,|_{x_k}\ \delta \underline{x^k} = -\underline{R}(\underline{x_k}) \tag{8a}$$

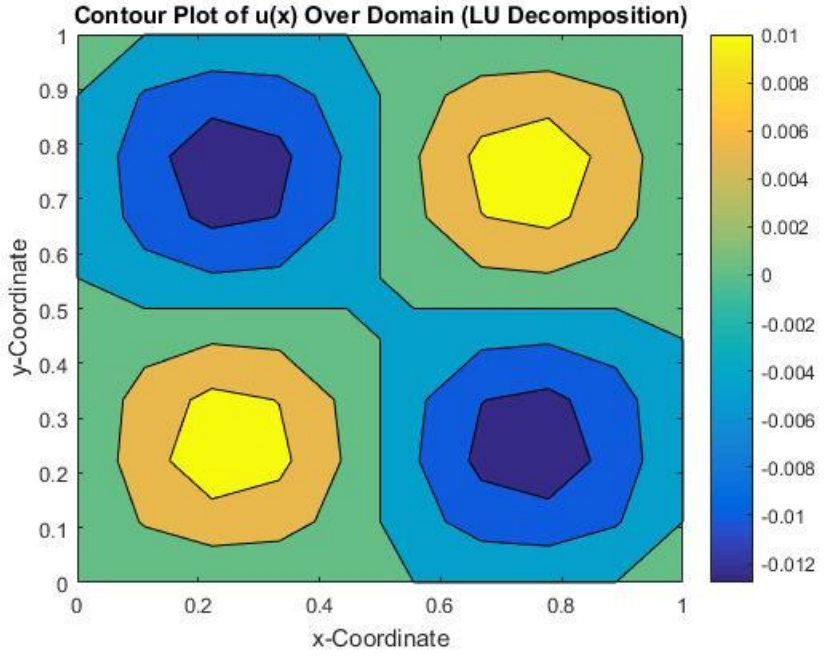
$$\underline{x^{k+1}} = \underline{x^k} + \delta \underline{x^k} \tag{8b}$$

These solvers are coded in MATLAB and the subsequent solution for the boundary value problem is obtained. The tolerance for the iterative solvers is set to 1e-06. Moreover, in the MATLAB code, an upper cap for the number of iterations is given just in case to provide a failsafe for divergence in the solution, if any.

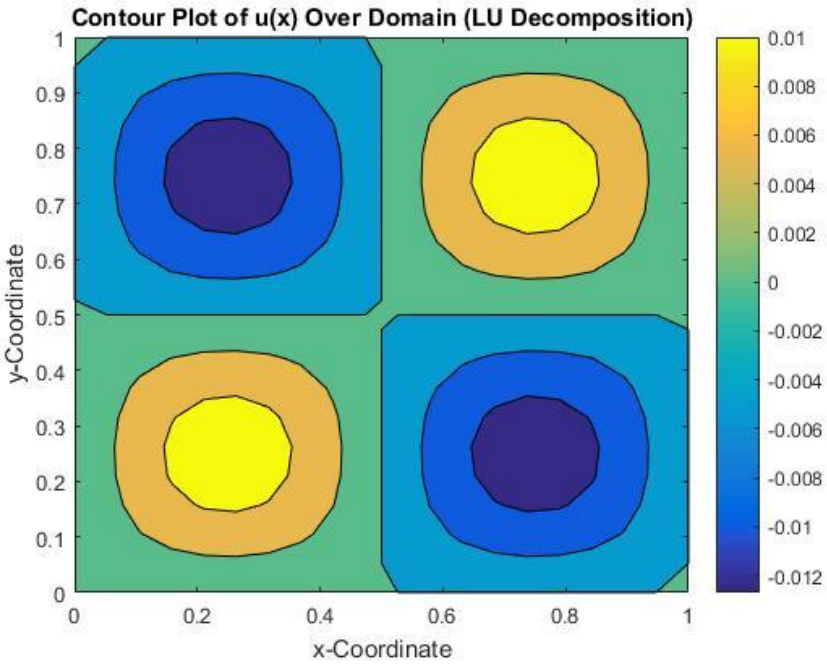
3. Results and Discussion

3.1 Numerical Solution using LU Decomposition

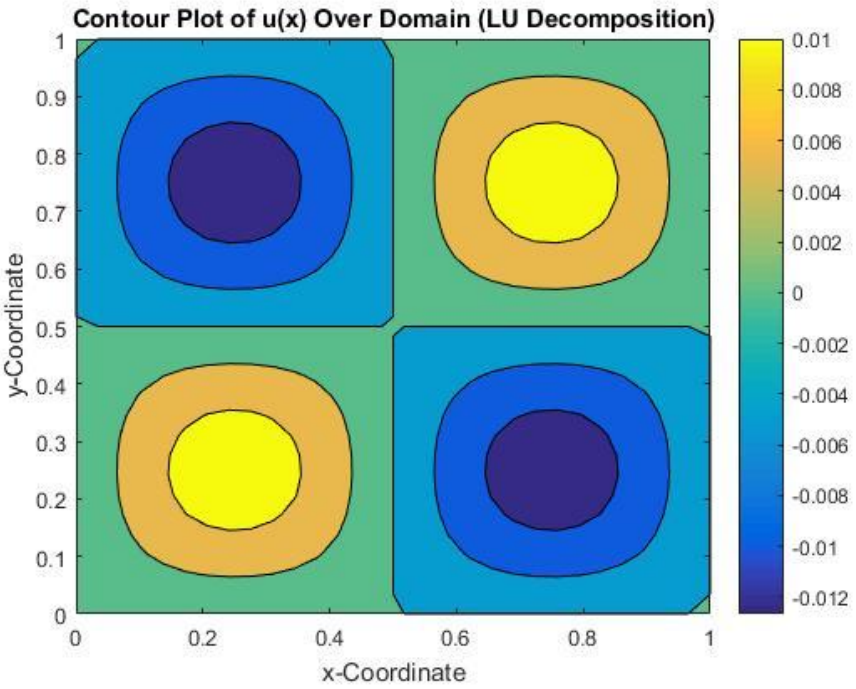
For the first part of the project, the solution to the boundary value problem is calculated using the LU decomposition code. Matrix $A\,(n\times n)$ and $b\,(n\times 1)$ are generated based on the problem and the boundary conditions defined in the problem. The contours for the variation of *u* over the domain are shown using 10, 20, 30 and 40 grid points per axis. Figure 2 shows the results for the same.



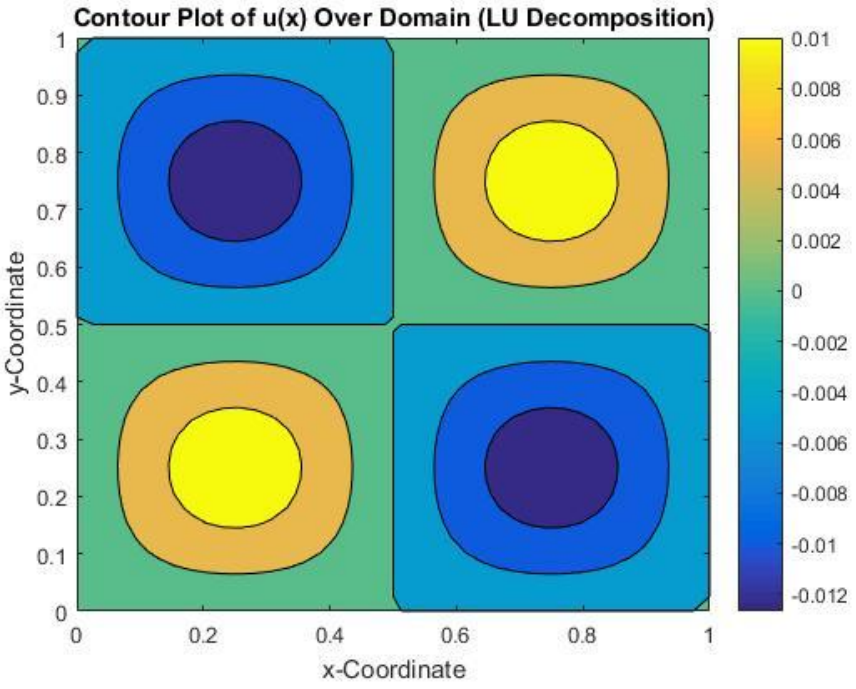
(a)



(b)



(c)



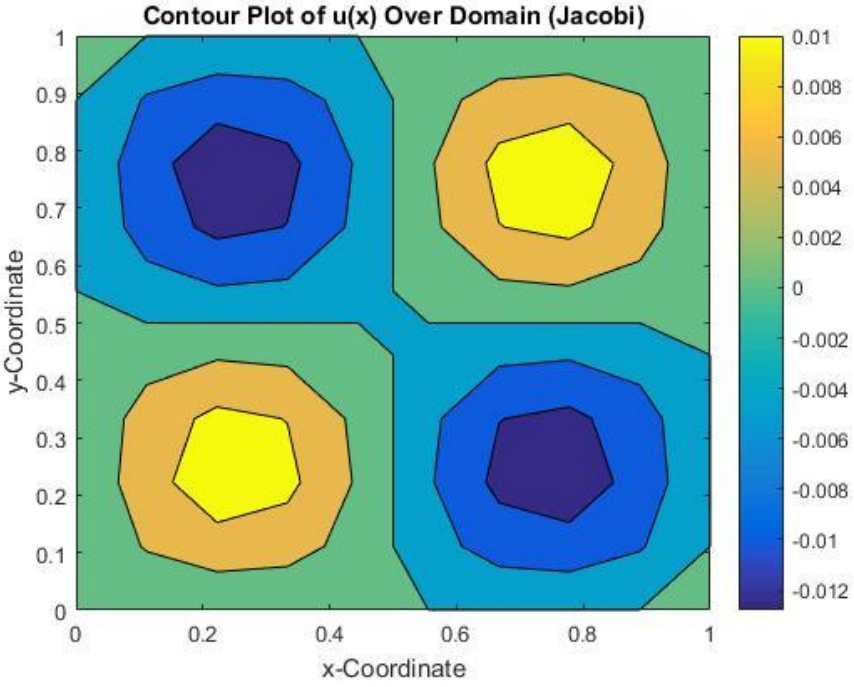
(d)

Fig 2: Contour plots of u(x) over domain for different grid sizes per axis (LU Solver), (a) 10, (b) 20, (c) 30, (d) 40

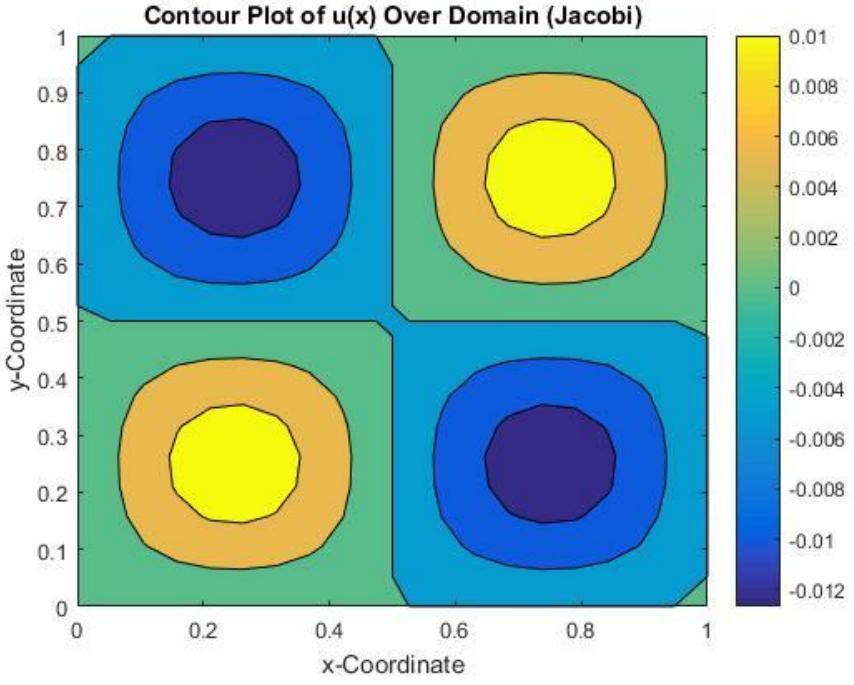
As can be seen from Figure 2, the contours, and hence, the solution for the boundary value problem seems to become smoother or finer as the number of grid points increase. The LU decomposition method scales as n^3 and hence, increasing the number of grid points results in solving a large matrix which is computationally intensive.

3.2 Numerical Solution using Jacobi Iterative Solver

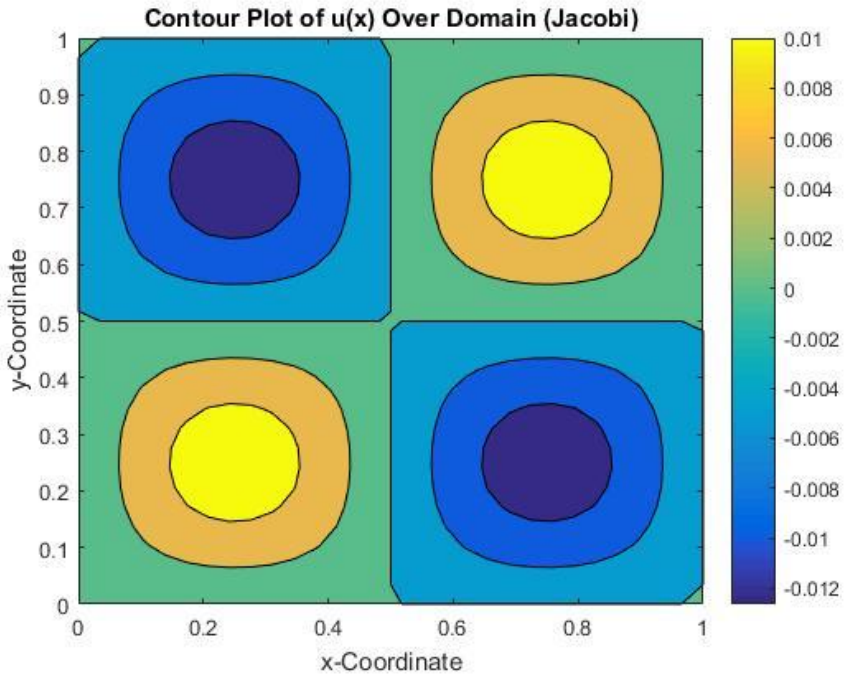
The same experiment is conducted using a Jacobi iterative solver represented in Equation 6. The contours for the variation of *u* over the domain are shown using 10, 20, 30 and 40 grid points per axis. Figure 3 shows the results for the same.



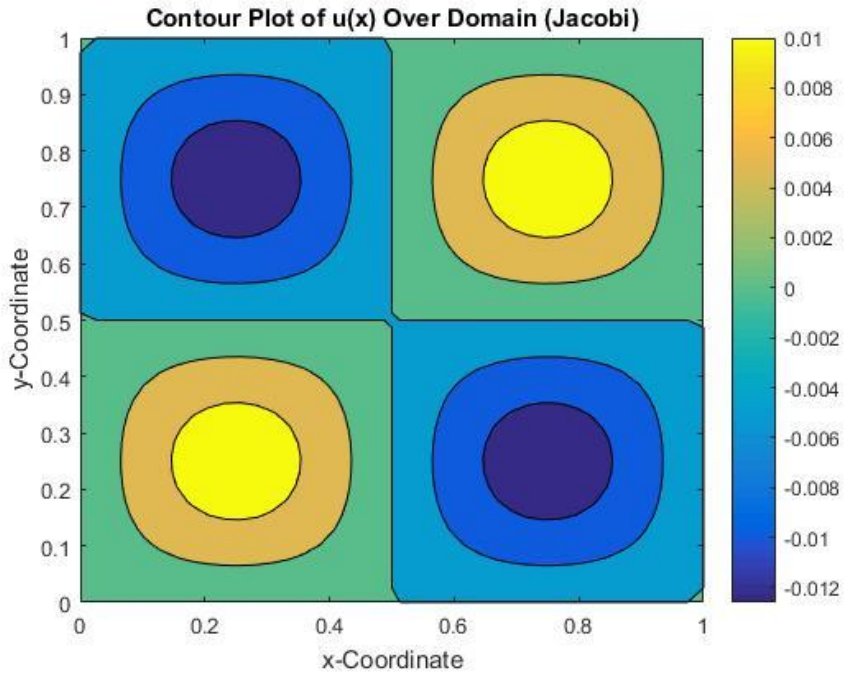
(a)



(b)



(c)



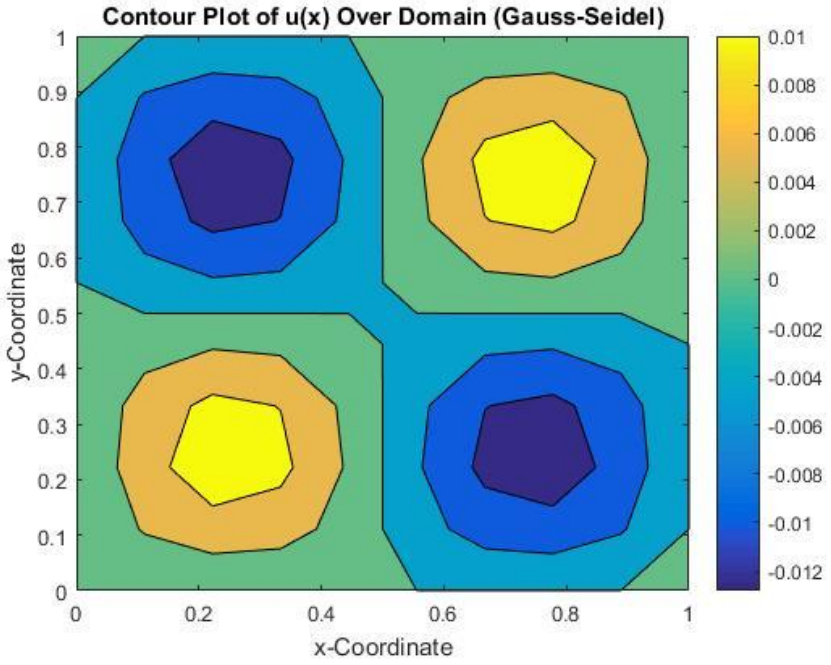
(d)

Fig 3: Contour plots of u(x) over domain for different grid sizes per axis (Jacobi), (a) 10, (b) 20, (c) 30, (d) 40

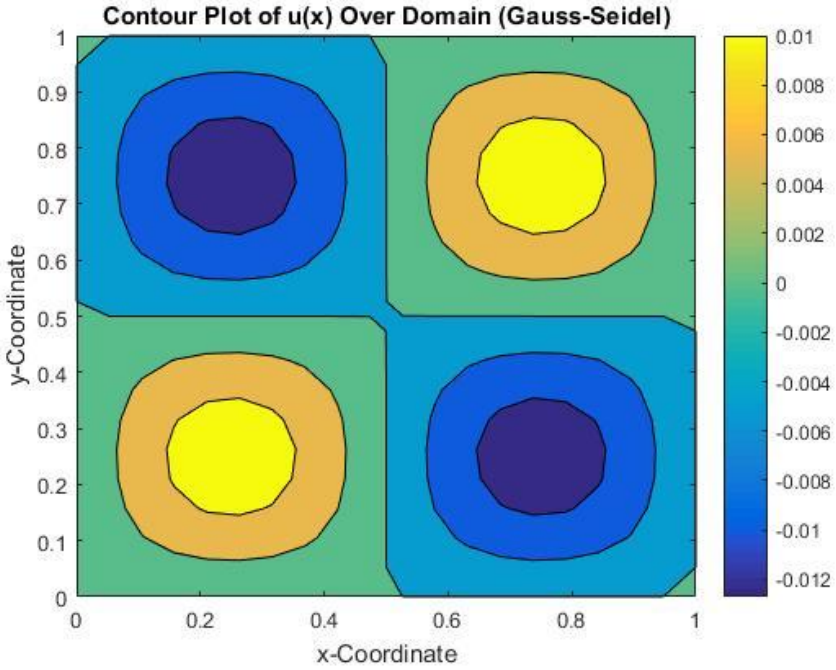
For the Jacobi solver, the solution contours look nearly the same as in the LU Decomposition solver. The Jacobi solver uses an initial guess to compute the value of *u* at the next iteration. The time scaling for the same is compared later on in this report.

3.3 Numerical Solution using Gauss-Seidel Iterative Solver

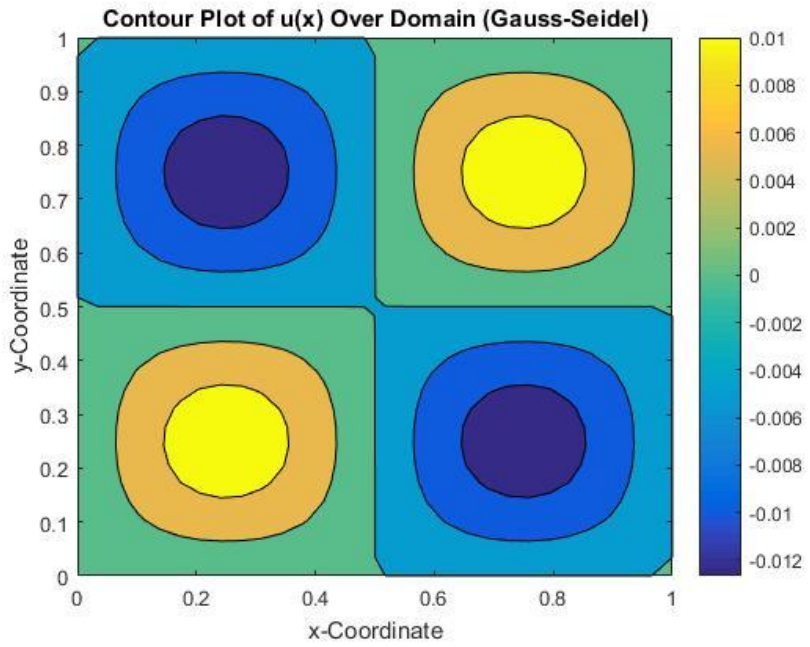
The experiment is then conducted using a Gauss-Seidel iterative solver represented in Equation 7. The contours for the variation of *u* over the domain are shown using 10, 20, 30 and 40 grid points per axis. Figure 4 shows the results for the same.



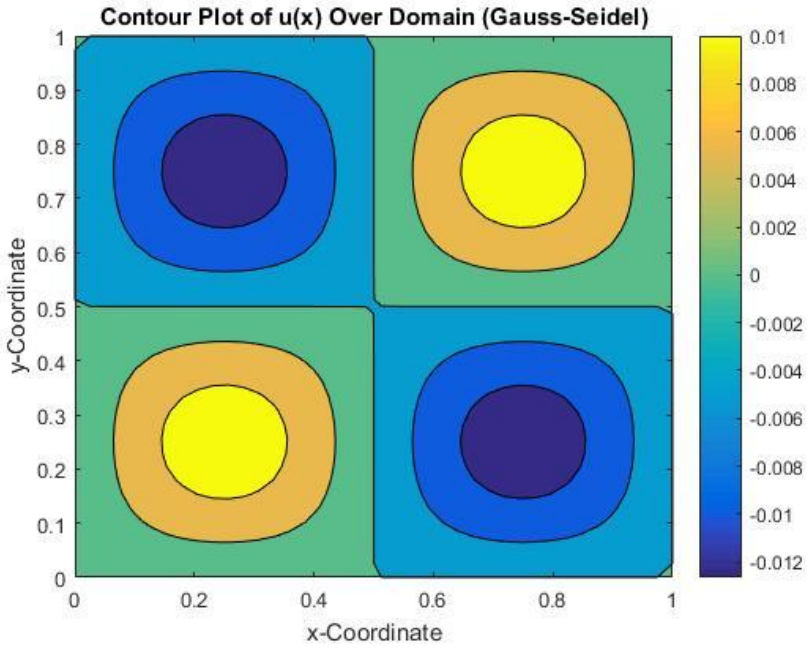
(a)



(b)



(c)



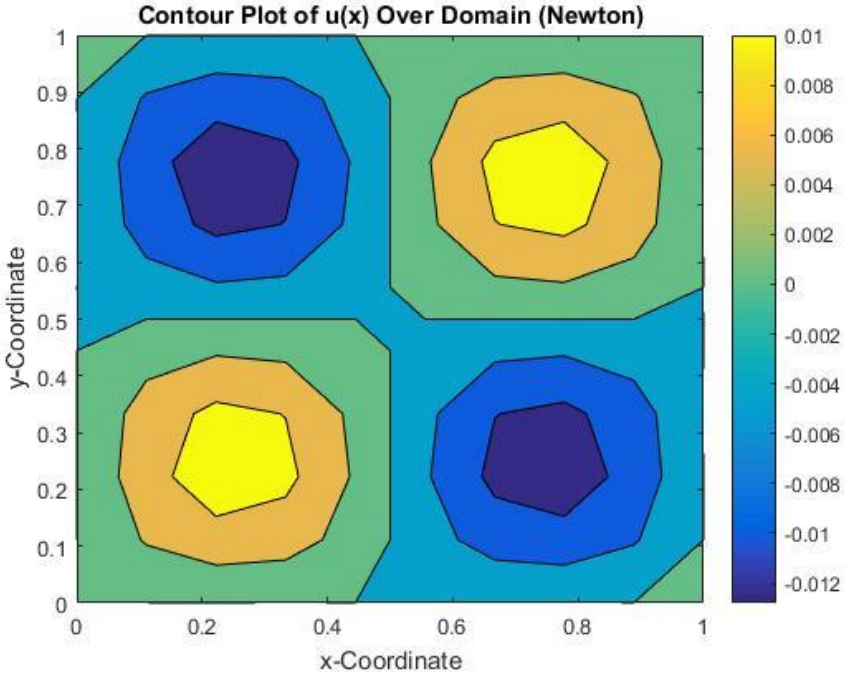
(d)

Fig 4: Contour plots of u(x) over domain for different grid sizes per axis (Gauss-Seidel), (a) 10, (b) 20, (c) 30, (d) 40

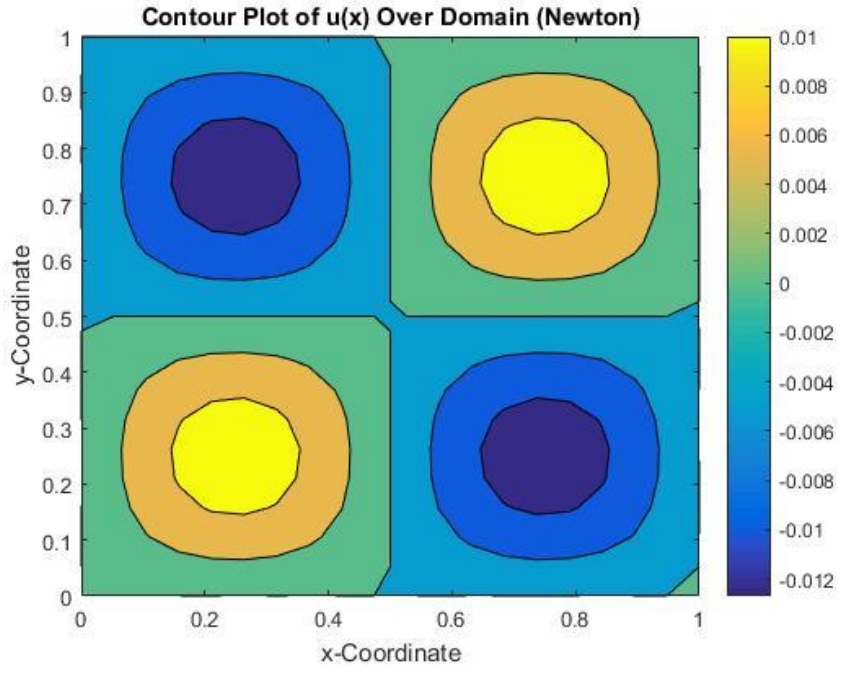
For the Gauss-Seidel solver, the solution contours look nearly the same as in the LU Decomposition solver and the Jacobi iterative solver. Even here, the contours become finer on increasing the number of grid points. The solver uses the new value of u as well as the newly computed values of u to arrive at a new guess for u. The time scaling for the same is compared later on in this report.

3.4 Numerical Solution using Newton's Method Solver

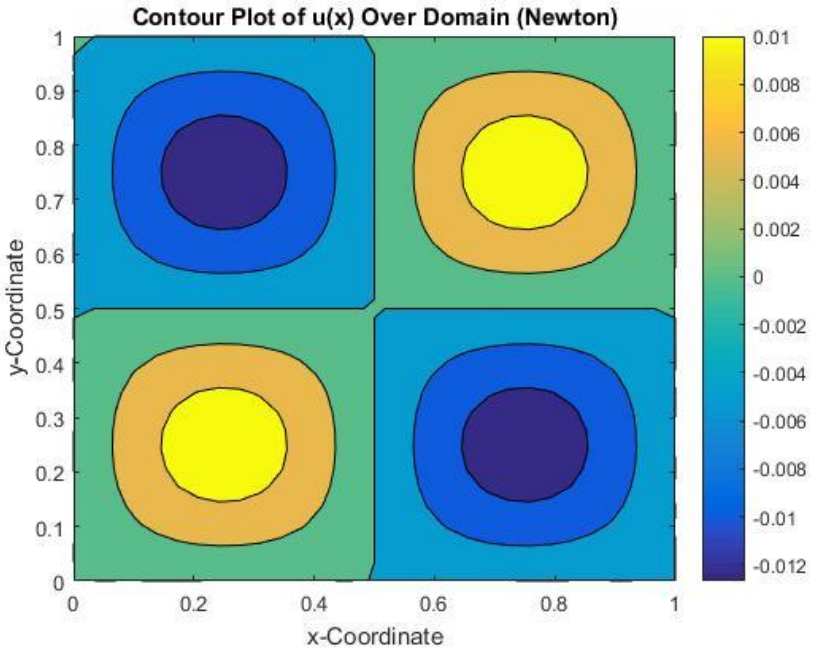
The Newton's Method algorithm is run to determine values of *u* using the *A* and *b* matrices. Tests are run for different grid sizes. The contours for the variation of *u* over the domain are shown using 10, 20, 30 and 40 grid points per axis. Figure 5 shows the results for the same.



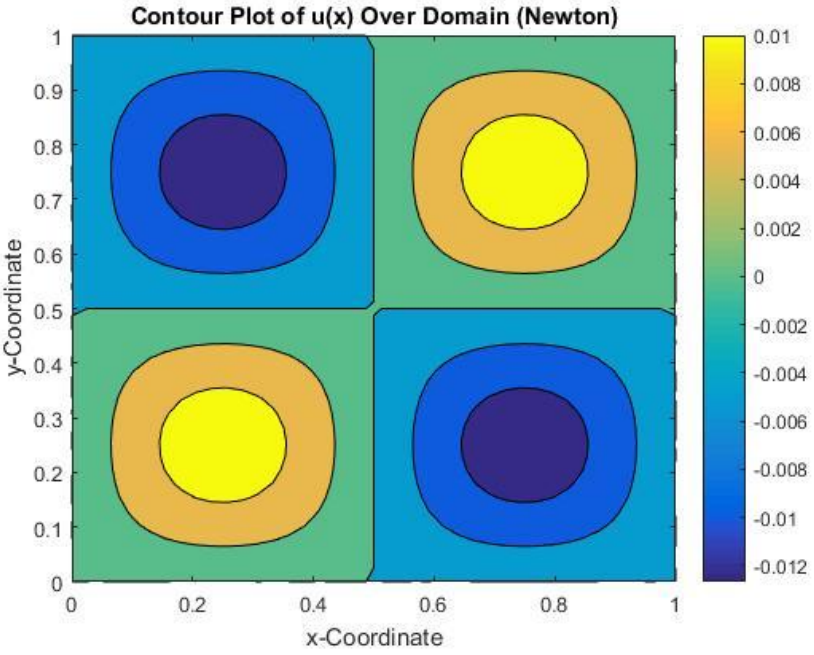
(a)



(b)



(c)



(d)

Fig 5: Contour plots of u(x) over domain for different grid sizes per axis (Newton), (a) 10, (b) 20, (c) 30, (d) 40

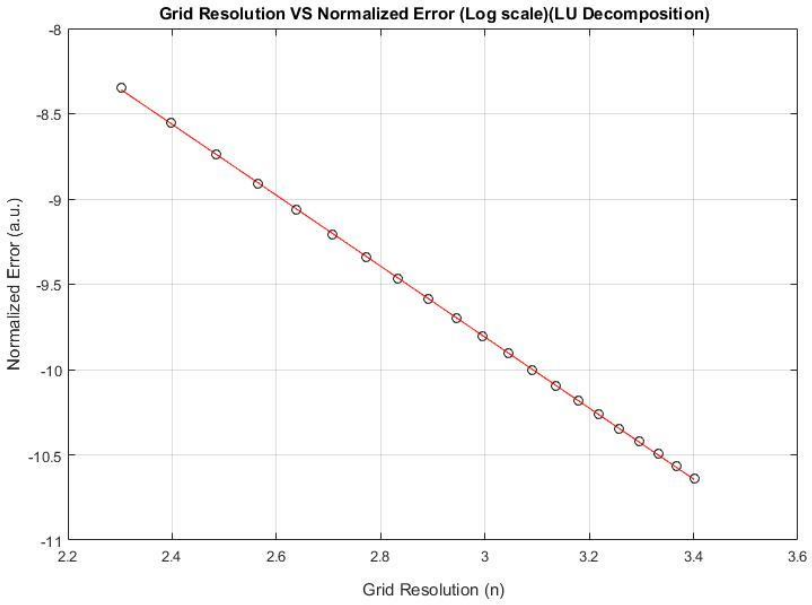
By using Newton's method, the contours still look nearly the same as the LU solver, and the Jacobi and Gauss-Seidel iterative solvers. The only difference while using Newton's method is that the solution is obtained in just one step. The operation count of Newton's method is usually n^3 times the number of iterations run. Moreover, if the method does not converge in nearly 12 steps, it can be safely said that the solution will not converge due to the nature of Newton's method.

3.5 Normalized Error versus Grid Resolution

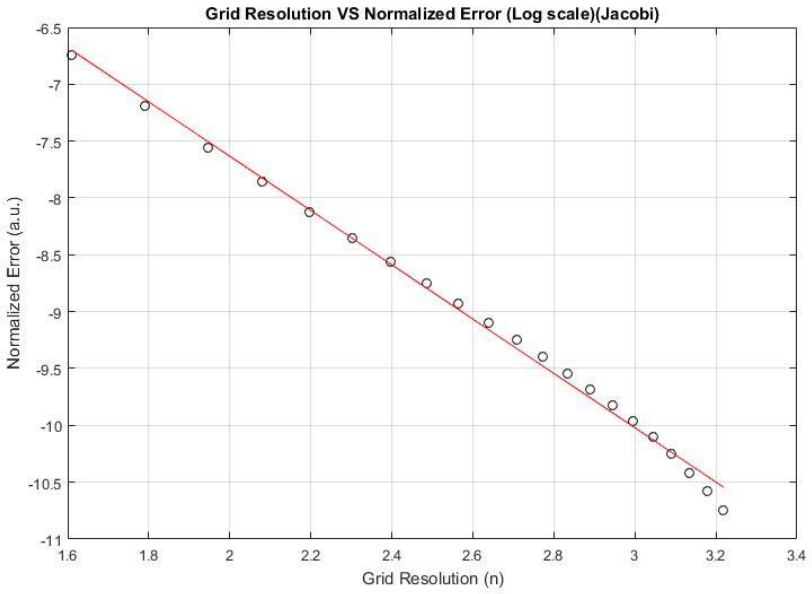
The variation of the normalized error with the grid resolution for the LU, Jacobi, Gauss-Seidel and the Newton solver is studied. The normalized error is defined as the root mean square error given by Equation 9:

$$L_2 = \sqrt{\frac{\sum (x_{exact} - x_i)^2}{n}} \tag{9}$$

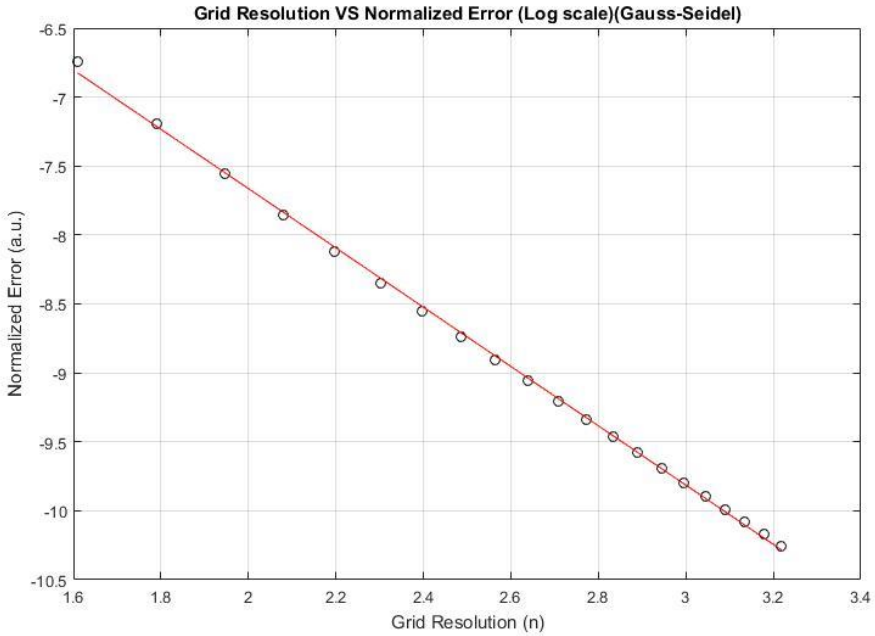
Plotting this normalized error as a function of the grid resolution for each solver, we get the plots as shown in Figure 6.



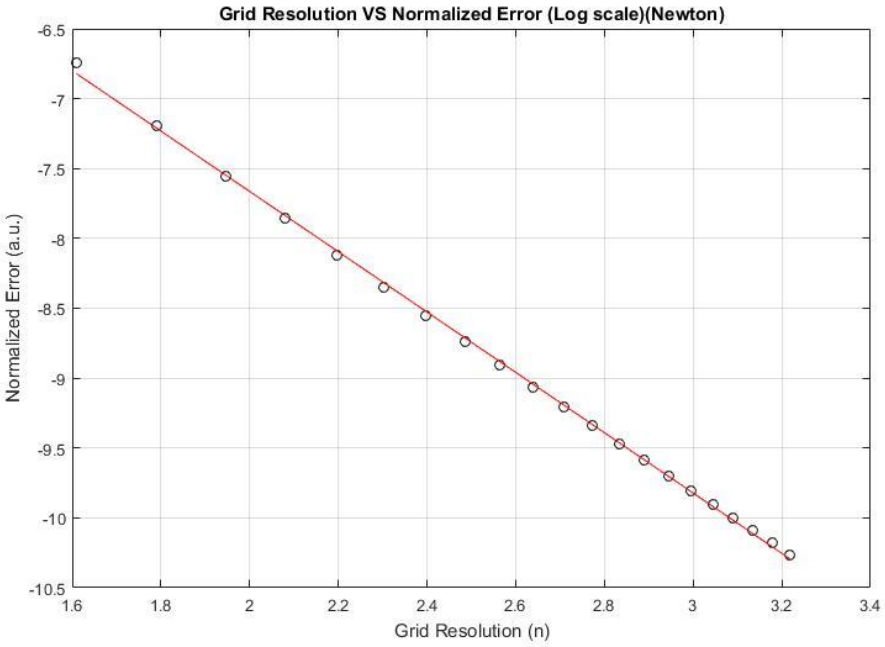
(a)



(b)



(c)



(d)

Fig 6: Grid resolution versus normalized error for, a) LU, b) Jacobi, c) Gauss-Seidel, d) Newton

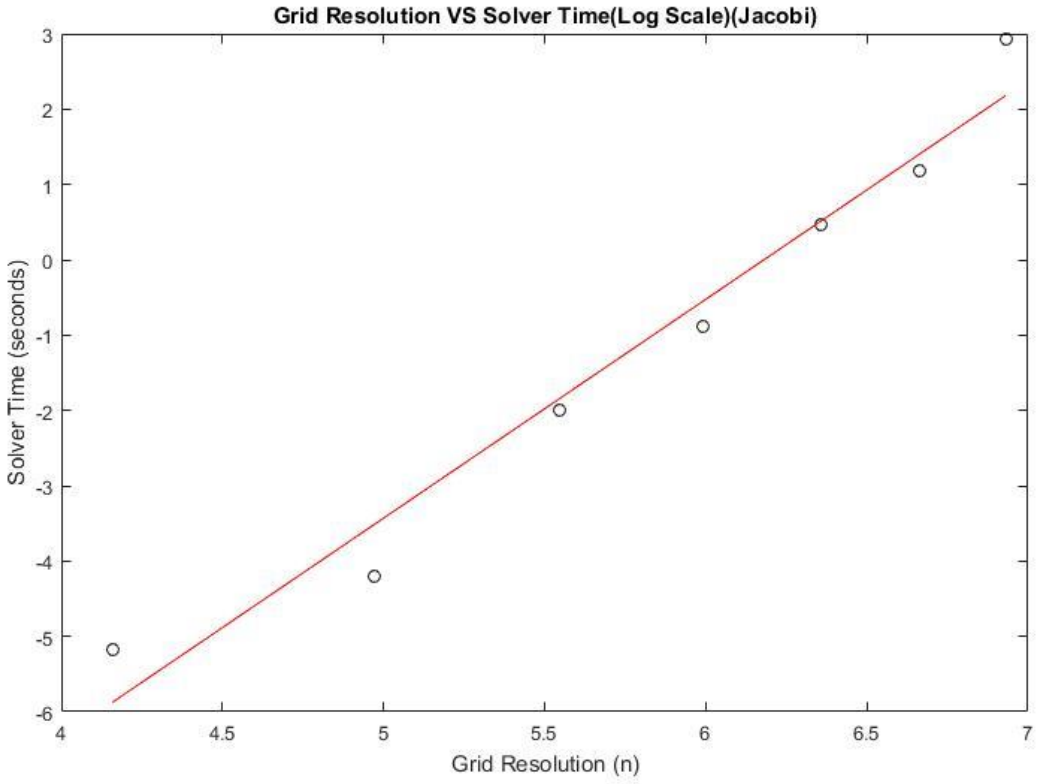
The exact solution to the boundary value problem is:

$$u(x) = \frac{\sin(2\pi x)\sin(2\pi y)}{8\pi^2} \tag{10}$$

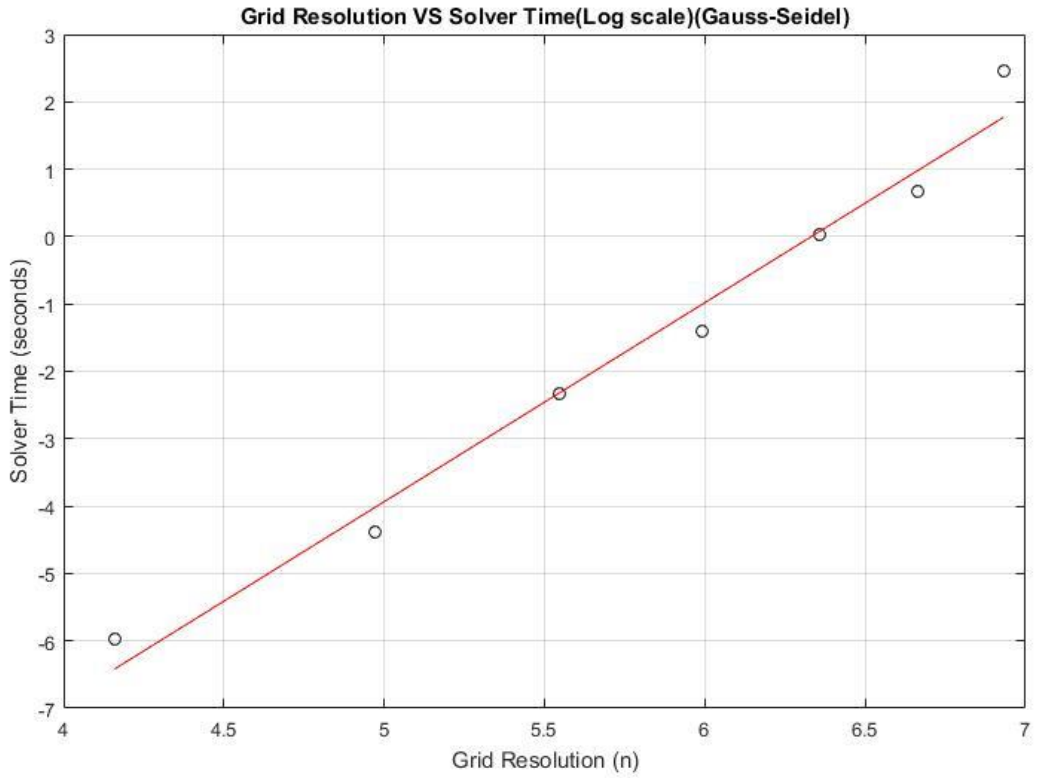
On plotting the normalized error versus the grid resolution, it is seen that the error scales as around -2. i.e. $-1/n^2$ with the scaling for LU decomposition, Jacobi, Gauss-Seidel and Newton's method being -2.0794, -2.1884, -2.1502 and -2.1585 respectively. Thus, as the resolution increases, the error keeps decreasing by an order 2.

3.6 Computational Scaling of Jacobi and Gauss-Seidel Iterative Solvers

The computational times for the Jacobi and the Gauss-Seidel iterative solvers are found for different grid resolutions to determine the scaling of each. The plots for the same are shown in Figure 7.



(a)



(b)

Fig 7: Grid Resolution VS Solver Time(Log Scale) for, a) Jacobi, b) Gauss-Seidel

It is observed that the scaling for the Jacobi and the Gauss-Seidel solvers comes to be around n^3, the exact value being 3.0514 for Jacobi and 3.9055 for Gauss-Seidel. Even for the direct solver, the scaling is of order 3. On comparing the two solvers, it is seen that the Gauss-Seidel solver is nearly twice as faster than the Jacobi solver which is because the Gauss-Seidel solver uses the new and the old values of u to solve the system of equations. Moreover, it is seen that both solvers will always converge to the right solution.

4. Conclusion

By running different test cases for different grid resolutions, we can conclude that the solution obtained tends to be better for higher number of grid points. This can be seen by the contour plots of each solving method. As the number of grid points increases, the contours tend to become much smoother and refined and provide definitive information regarding the variation of $u(x)$ over the whole domain.

Moreover, as we calculate the normalized error as a function of the number of grid points or the resolution, the error tends to decrease as the resolution increases. This is verified in all solver cases. On increasing the number of grid points, the error keeps decreasing by an order of -2. As a result, it can be concluded that by resolving the domain into finer components, we can reduce the margin of error in the calculation of $u(x)$.

On plotting the variation of solver time and the grid resolution, it is seen that both the iterative solvers scale as n^3, which is the same as the direct method of solving the equations. Both iterative solvers always tend to converge with the Gauss-Seidel solver being nearly twice as faster than the Jacobi solver.

By running these series of tests, the above conclusions have thus been made. Since the system being used to compute these results took a considerable amount of time to solve high grid resolutions, the maximum grid resolution in the calculations has been taken as 40, thus resulting in 1600 points and equations. By increasing the number of points further, deeper insight to the scaling of the error and the solver time with the grid size can be gathered.

5. Appendix

The MATLAB code for the discretization of the boundary value problem and each individual solver is given as follows:

```
% Finite Difference

function [x,t] = finitedifference(nx,ny,tol)
   hx=1/(nx-1);
   hy=1/(ny-1);
   n = nx*ny;
   T = zeros(n);
   b = zeros(n,1);
   l2 = 0;

   for i = 1:nx
      for j = 1:ny
         l=(j-1)*(nx)+i;
         X(i,1)=(i-1)*hx;
         Y(j,1)=(j-1)*hy;

         if i==1 || i==nx
            T(l,l)=1;
            b(l,1)=0; %-sin(2*pi*X(i))*sin(2*pi*Y(j));

         elseif j==1 || j==ny
            T(l,l)=1;
            b(l,1)=0; %-sin(2*pi*X(i))*sin(2*pi*Y(j));

         else
            T(l,l)=(-2/(hx^2))+(-2/(hy^2));
            T(l,l+1)=1/(hx^2);
            T(l,l-1)=1/(hx^2);
            T(l,l+nx)=1/(hy^2);
            T(l,l-nx)=1/(hy^2);
            b(l,1)=-sin(2*pi*X(i))*sin(2*pi*Y(j));
         end
      end
   end

   [x,t] = ludecomp(T,b,tol,n);
   [x,t] = Jacobi(T,b,tol,n);
   [x,t] = GaussSeidel(T,b,tol,n);
   [x,t] = Newton(T,b,n);
   [l2] = l2norm(x,n,nx,ny,X,Y);
   figure; cont(x,nx,ny,X,Y);
   xlabel('x-Coordinate');
   ylabel('y-Coordinate');
   title('Contour Plot of u(x) Over Domain');
   colorbar;
end
```

```
% Jacobi

function [x,t] = Jacobi2(T,b,tol,n)
   x0 = zeros(n,1);
   x = zeros(n,1);
   maxiter = 10^4;
   c=1;
   tic;
   while c < maxiter
      for i = 1:n
         x(i)=((b(i) - T(i,[1:i-1,i+1:n])*x0([1:i-1,i+1:n]))/T(i,i));
      end
      if max(abs(x-x0))<tol
         break
      end
      c=c+1;
      x0 = x;
   end
   t=toc;
end


% Gauss-Seidel

function [x,t] = GaussSeidel(T,b,tol,n)
   x0 = zeros(n,1);
   x = zeros(n,1);
   maxiter = 10^4;
   c = 1;
   tic;
   while c < maxiter
      for i = 1:n
         x(i)=((b(i)-T(i,[1:i-1])*x([1:i-1])-T(i,[i+1:n])*x0([i+1:n]))/T(i,i));
      end
      if max(abs(x-x0))<tol
         break
      end
      c=c+1;
      x0 = x;
   end
   t=toc;
end


% Newton

function [x] = Newton(T,b,n)
   x = zeros(n,1);
   tic;
   R = T*x-b;
   dRx = T;
   deltax = -dRx\R;
   x = x + deltax;
   toc;
end
```

```
% L2 Norm

function [l2] = l2norm(x,n,nx,ny,X,Y)

   for i = 1:nx
      for j = 1:ny
         l = (j-1)*(nx)+i;
      exact(l,1) = sin(2*pi*X(i))*sin(2*pi*Y(j))/(8*(pi^2));
      end
   end

   for i = 1:l
      error(i,1) = exact(i,1)-x(i,1);
      error2(i,1) = error(i,1)*error(i,1);
   end
   sum = 0;
   for i = 1:l
      sum =  sum + error2(i,1);
   end
   mean = sum/n;

   l2 = sqrt(mean)
end


% Contour

function cont(x,nx,ny,X,Y)

   c = zeros(ny,nx);

   for i = 1:nx
      for j = 1:ny
         l=(j-1)*(nx)+i;
         c(j,i) = x(l);
      end
   end
   contourf(X,Y,c)
end
```

```
% Scaling

function scaling(k,tol)
   n = zeros(k+1,1);
   m = zeros(k+1,1);
   l = zeros(k+1,1);
   n(1) = 5;
   for i = 1:k+1
      [x,l2] = finitedifference(n(i),n(i),tol);
      n(i+1)=n(i)+1;
      l(i) = log(n(i));
      m(i) = log(l2);
      plot(l(i),m(i),'ok');
      hold on;
   end
   q = polyfit(l,m,1)
   y = polyval(q,l);
   plot(l,y,'-r')
   title('Grid Resolution VS Normalized Error (Log scale)(Jacobi)');
   xlabel('Grid Resolution (n)');
   ylabel('Normalized Error (a.u.)');
   grid on;
end
```