Dhruv Gupta
ENM 502
2.17.22
HOMEWORK ASSIGNMENT 1

Numerical Solution for PDE

Introduction

The assignment requires us to implement a code that performs LU decomposition on a matrix in order to solve a boundary value problem in the form of a Partial Differential Equation. The problem statement is a second order partial differential equation

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial x^2} + \alpha = 0 \quad s.t\ x \in [0,1] \cap y \in [0,1] \qquad (1)$$

with boundary conditions

$$T(x,y) = 0 \quad \forall \quad (0,y) \cap (1,y) \cap (x,0) \cap (x,1) \qquad (2)$$

The domain of the problem is visualized in Figure 1.

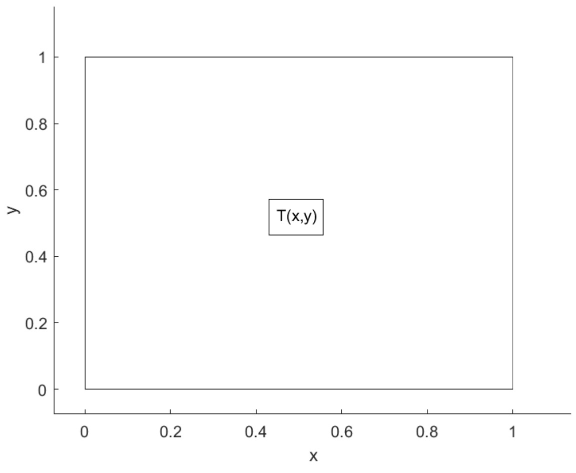


Figure 1: Problem Structure

The equation can be used to model heat conduction and other similar problems, where the source is time-independent (hence the constant α).

The numerical method being used is Finite Difference (FD) Approximation, in order to numerically solve the given differential equation. We use FD to approximate the derivatives wherever required and reduce the entire problem to a system of linear equations in the form of

$$Ax = b \qquad (3)$$

Our primary objective is to gauge the performance of our LU decomposer and solver across many resolutions, in 1D as well as 2D. The hypothesis is that the time taken by the LU routine will scale as N^3 (N is the grid size used in FD). We will also compare the performance with that of the MATLAB ***lu()*** routine. The overall behavior should be the same, but the MATLAB routine would be expected to run much more quickly given that it is presumably well-optimized. As a result, we also expect the 1-D solver to be quicker, given that the matrix (*A*) size is smaller for a given resolution when compared to the 2-D problem. We will also discuss the role of the bandwidth of said matrices in the complexity of the problem.

Problem Setup and Formulation

Finite Difference Approximation

In the context of the 2-D problem - the discretization approach being used is the FD method. We chop up the domain into a rectangular grid with (N_x + 1) and (N_y+1) points on the x-axis and y-axis respectively. Now, in this coarse grid, each point is

$$T(x_i, y_j) \quad s.t \quad i \in \{0, 1 \dots (N_x + 1)\}, \ j \in \{0, 1 \dots (N_y + 1)\} \qquad (4)$$

, and the grid spacing (h_x and h_y) is thus determined by the number of points we take on the axes, i.e, the resolution of the grid. The higher our resolution, the closer we should be to the solution obtained analytically. The spacing is

$$h_x = \frac{1.0}{N_x}, \ h_y = \frac{1.0}{N_y} \qquad (5)$$

Now that we've mapped our function to a grid, we can begin using Finite Difference (FD) to resolve the PDE. FD uses the Taylor series to approximate a derivative; here we use the central difference to find the second derivative approximation

$$f_x''(x_i) = \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{h_x^2} \qquad (6)$$

Substituting the approximation (6) we obtain our discretized PDE

$$T_x''(x_i, y_j) + T_y''(x_i, y_j) + \alpha = 0 \quad (7)$$

… - which are a series of equations for each point in the grid. However, dealing with two indices (i,j) would be cumbersome. Hence, the equation is remodeled such that (i,j) are combined into one index ($l = (j - 1)(N_x + 1) + i$), where we go through all the points starting from the bottom left (0,0) and making our way to the top-right (1,1). Since the equations are now representative of each point, we have $(N_x + 1) \times (N_y + 1)$ many equations in our system, and as many unknowns. The remodeled equations for $N_x = N_y$ are now

$$\frac{T_{l+1} - 2T_l + T_{l-1}}{h^2} + \frac{T_{l+(Nx+1)} - 2T_l + T_{l-(Nx+1)}}{h^2} = -\alpha \qquad (8)$$

… and these represent a system of linear equations which can be summarized as $A.T = b$. A contains the coefficients of the discretized PDE, while b will contain the right-hand side.

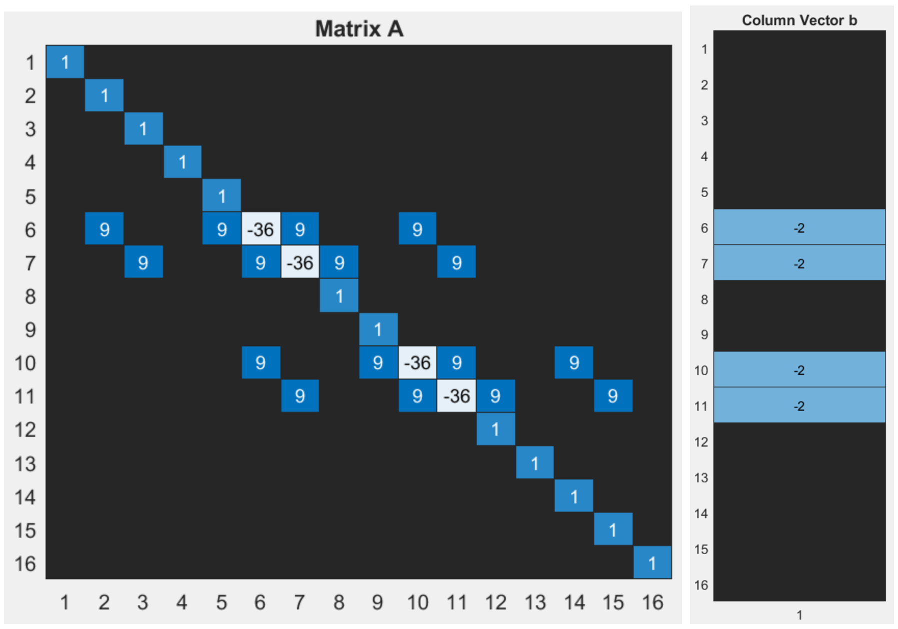
Tackling the boundary conditions

All values on the boundary are zero. In order to incorporate this into our matrix we use the equation

$$1.T_l = 0 \quad s.t \ \ l \in Boundary \qquad (9)$$

… to represent the boundary points. Below are the 2-D and 1-D systems illustrated in figure 2 and 3,

Figure 2: 2-D Formulation for Ax = b $[N_x = N_y = 3;\ \alpha = 2]$



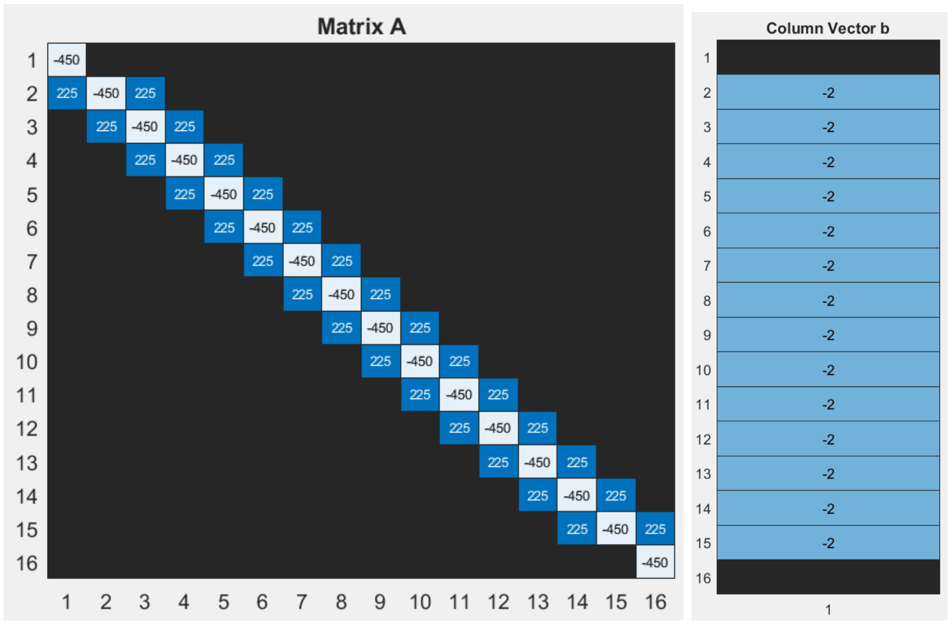
Matrix A

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1															
2		1														
3			1													
4				1												
5					1											
6		9			9	-36	9			9						
7			9			9	-36	9			9					
8								1								
9									1							
10						9			9	-36	9			9		
11							9			9	-36	9			9	
12												1				
13													1			
14														1		
15															1	
16																1

Column Vector b

	1
1	
2	
3	
4	
5	
6	-2
7	-2
8	
9	
10	-2
11	-2
12	
13	
14	
15	
16	

Figure 3: 1-D Formulation for Ax = b [$N_x = 15;\ \alpha = 2$]



… where the zeros in the system are represented by black cells. It is evident now that the matrix structure is extremely sparse, and this can be potentially exploited to solve the system more efficiently.

The LU decomposition starts by resolving the matrix A into a product of a lower and upper triangular matrix (L, U). The resulting system is solved using forward and backward substitution. Using MATLAB's *backslash*, the solution is obtained as

$$x = U\backslash(L\backslash b) \qquad (10)$$

tic and *toc* commands are used to measure the time taken by the user-defined LU routine and the intrinsic MATLAB routine. MATLAB is used to perform all of the analyses. I've tried my best to use functions as much as possible in order to make it more modular.
Error analysis uses the provided analytical solution and the grid-size-normalized L2-norm to gauge the accuracy of the results obtained. The *fit* module from the curve-fitting toolbox is used to find the scaling of the time taken by the routines and the error.

Results and Discussion

LU Decomposer and Solver

This is the crux of the report. $Ax = b$ is solved using the LU routine we have defined following the pseudo-code. First, we verify that the solver is returning the correct result, by matching the *x* obtained with the solution from MATLAB's *backslash* operator ($x = A\backslash b$).
For the LU solver, one of the parameters required is tolerance (*tol*). For the purposes of our problem, a mostly banded-diagonal matrix, varying the tolerance does not affect the solution. Table 1 records the error in the solution using various tolerances.

Table 1: Errors For Various Tolerances

tol	Error (1D; n_x = 2000)	Error (2D; n_x = 50)
10^{-1}	6.68×10^{-14}	4.82×10^{-5}
10^{-2}	6.68×10^{-14}	4.82×10^{-5}
10^{-3}	6.68×10^{-14}	4.82×10^{-5}
10^{-4}	6.68×10^{-14}	4.82×10^{-5}
10^{-5}	6.68×10^{-14}	4.82×10^{-5}
10^{-6}	6.68×10^{-14}	4.82×10^{-5}

Once this was verified, we ran the simulations varying grid-size for 1-D and 2-D problems to obtain the solution for Temperature in both cases. The grid-sizes are chosen based on the time taken to run the simulations. The contour maps for both cases are shown in figure 4. It's clear that as we move from a grid resolution of 10 to 50 for the 2-D grid, the contour smoothens. The corresponding heat-map in figure 5 illustrates this better. However, in the case of the 1-D solution, we do not see a significant difference in the solution from a grid resolution of 100 to 3500, illustrated in figure 6.

Figure 4: Contour Maps for 2-D Numerical Simulations

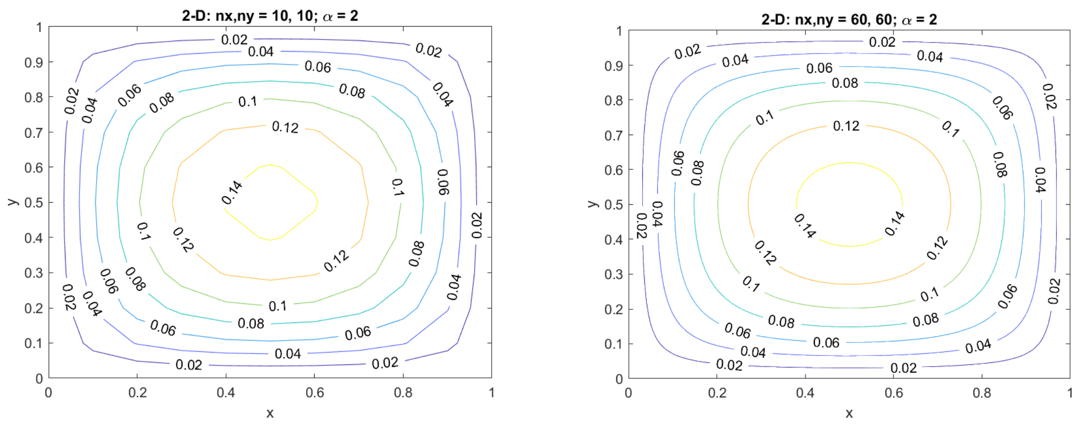


Figure 5: Heatmaps for 2-D Numerical Solutions

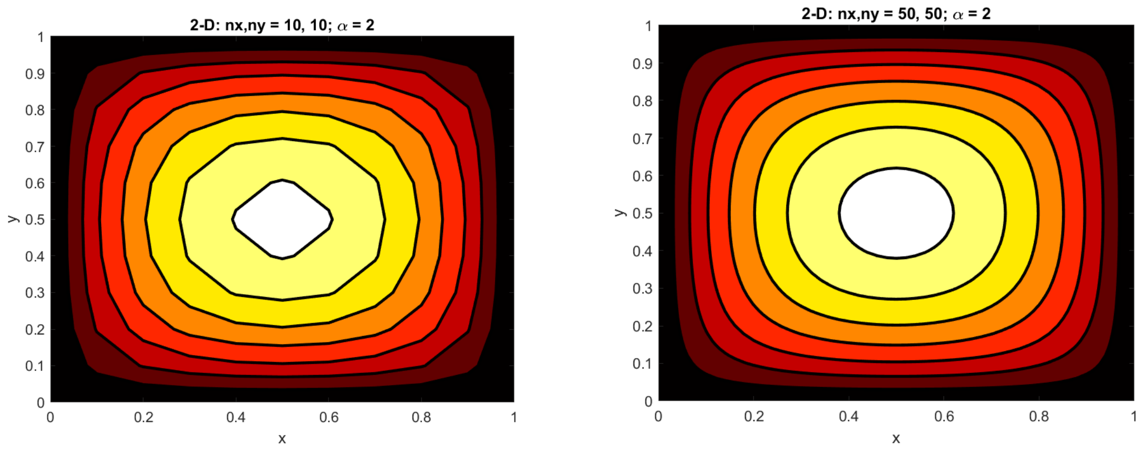
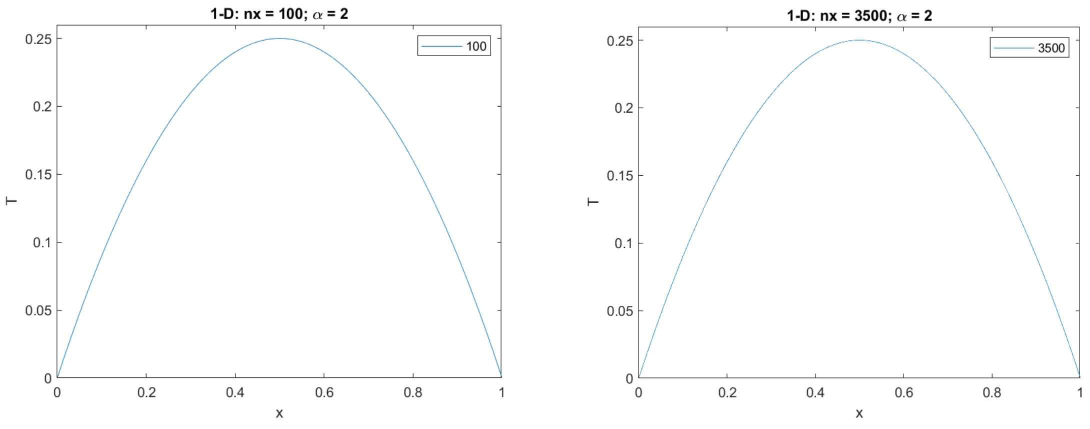


Figure 6: 1-D Numerical Simulation



Timing Scaling Analysis

Using the *tic* and *toc* commands we time our routine and compare it to the intrinsic routine. However, a single run wouldn't provide enough data to suggest a trend. Hence, we run all the simulations 5 times and note any major deviations. It is interesting to note that the time plots for the MATLAB routines are noisier than the user-defined routine in figure 8. The 1-D times do not show a lot of deviation either, apart from one exceptional run, shown in blue in figure 9. Again, the MATLAB routine time is very jittery. A really odd result is also seen in the MATLAB times, where the time taken for the first run is much larger than the subsequent ones. This could possibly be due to memory operations or such complications within MATLAB's lu() routine.

Figure 8: Time Taken By 2-D Simulations

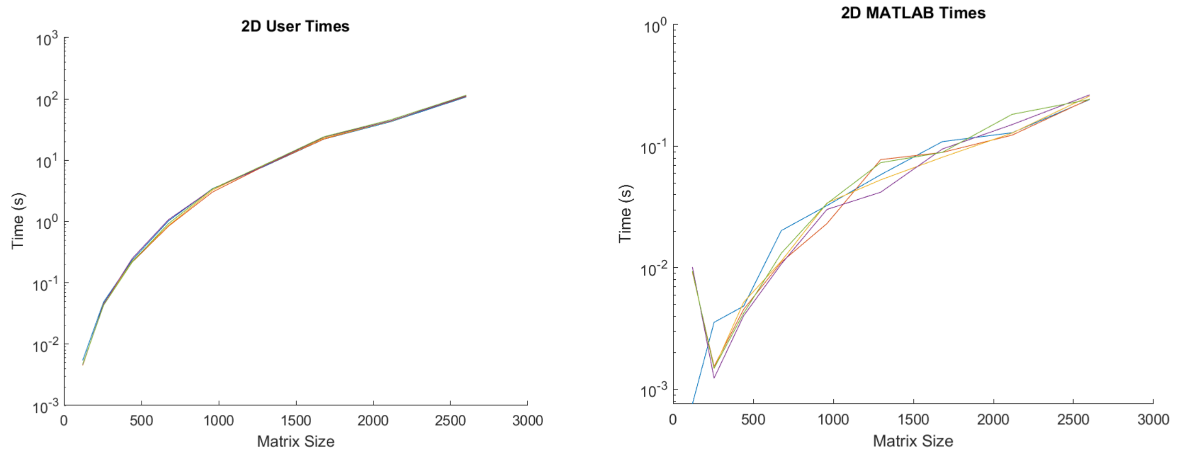
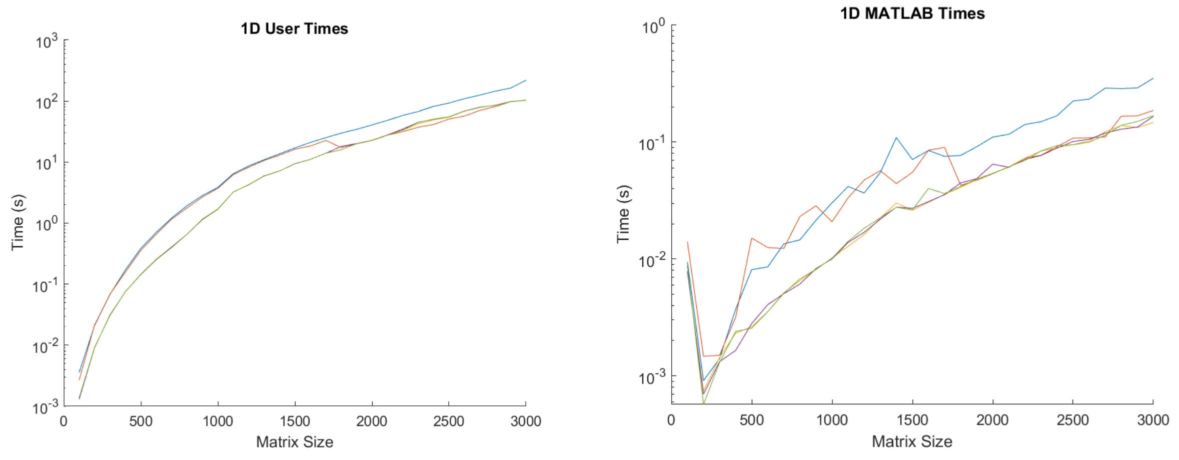


Figure 9: Time Taken By 1-D Simulations



If we use the '*power1*' parameter to fit the data, the *fit* function fits the data to a power function $a.x^b$. The scaling factors are obtained in table 2.

Table 2: Time Scaling Factors

	2-D Scaling Factors	1-D Scaling Factors
MATLAB *lu()*	2.21	2.36
User-Defined *LU*	4.01	3.68

We expect the LU decomposition to scale as N^3 for a given matrix of grid size N, but these results are slightly deviating from the theory, shown in figures 10 and 11. The MATLAB routine is expected to be well-optimized and thus performs extremely well for both cases. While it was expected that due to the larger bandwidth of the 2-D matrix, the scaling would be poorer for the intrinsic solver, it turns out to not be the case. Meanwhile, the user-defined solver performs poorly for both the cases.

Figure 10: 2-D Time Scaling Fit

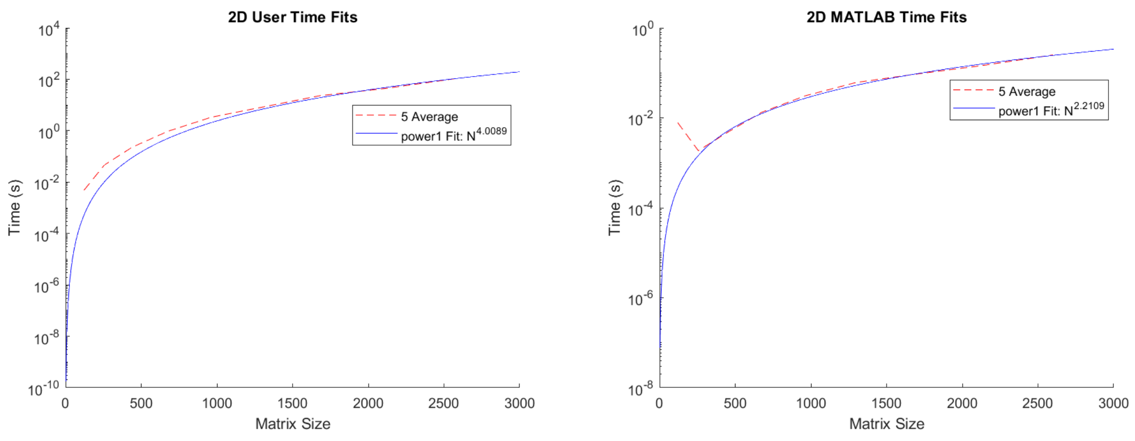
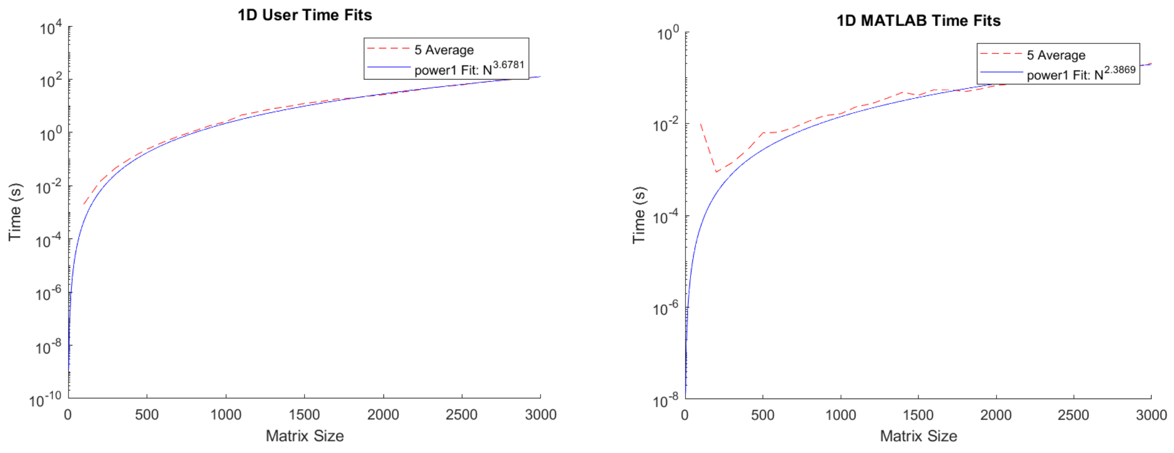


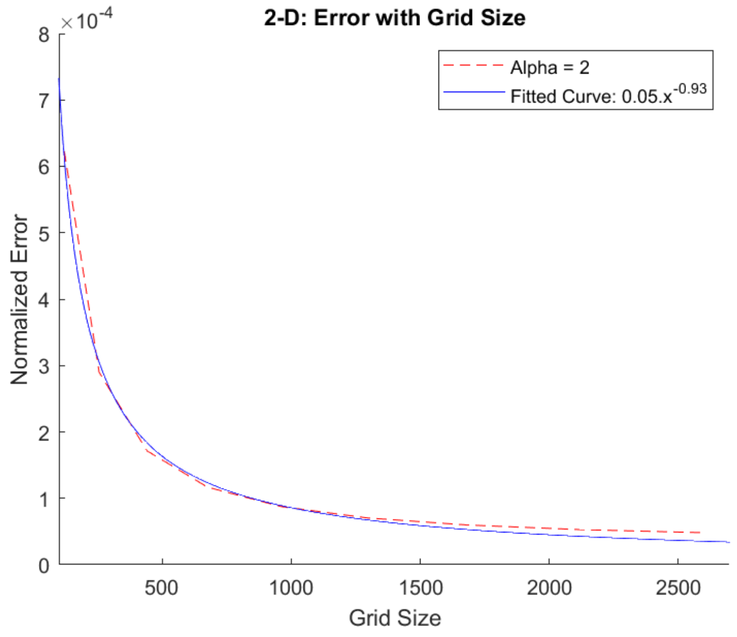
Figure 11: 1-D Time Scaling Fit



Error Analysis

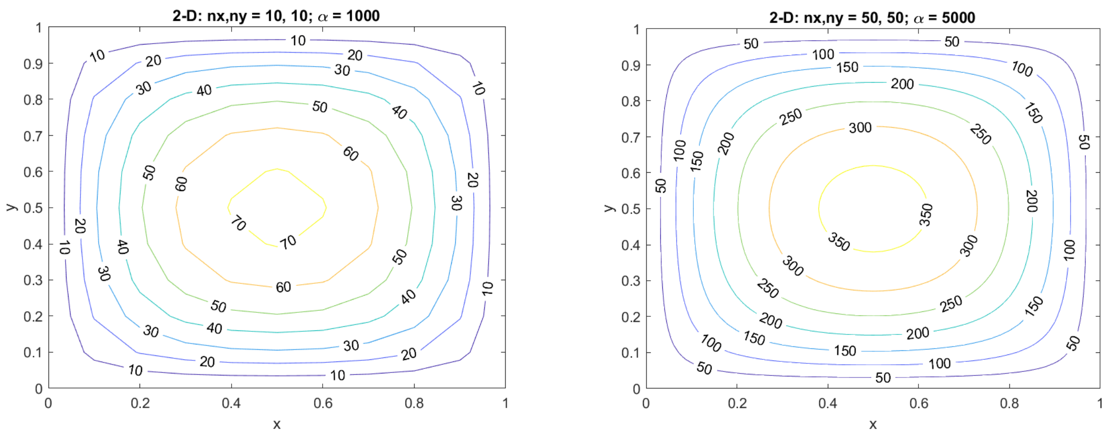
The normalized error in the solution is plotted against the grid-size in Figure 12. The scaling obtained when fit with a power function is ~ -1.0, following an asymptotic behavior. The asymptotic behavior is expected, since we can’t expect to reach the resolution of the analytic solution with a finite set of grid points.

Figure 12: Normalized Error With Grid-Size



For increased alpha, we don't see a stark difference in the contour maps of the solutions obtained, in figure 13. The grid sizes of 10 and 50 show similar behavior to when α was 2.

Figure 13: 2-D Solutions for Higher α



The error with alpha increases linearly, as seen in figure 14, for multiple grid resolutions. A lower grid-size, expectedly, gives a higher error. At the same time, the rate of increase in error is also greater for the lower resolution grids. We can reason out that the error should scale linearly with alpha since it is just a scaling factor in the equation (8)

$$\frac{T_{l+1} - 2T_l + T_{l-1}}{\alpha.h^2} + \frac{T_{l+(Nx+1)} - 2T_l + T_{l-(Nx+1)}}{\alpha.h^2} = -1 \qquad (11)$$

A higher alpha likely causes approximation errors during the backward and forward substitution.

It is clear from the log plot in figure 15 that the same behavior is followed across grid sizes.

Figure 14: Error with Alpha

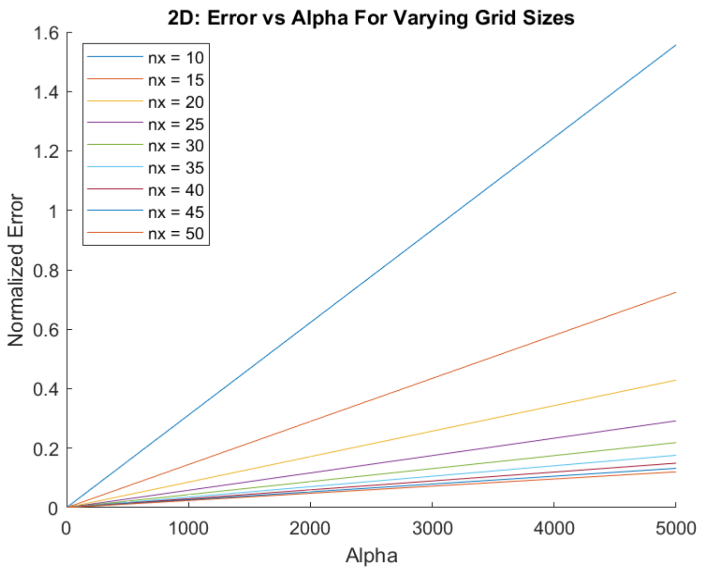
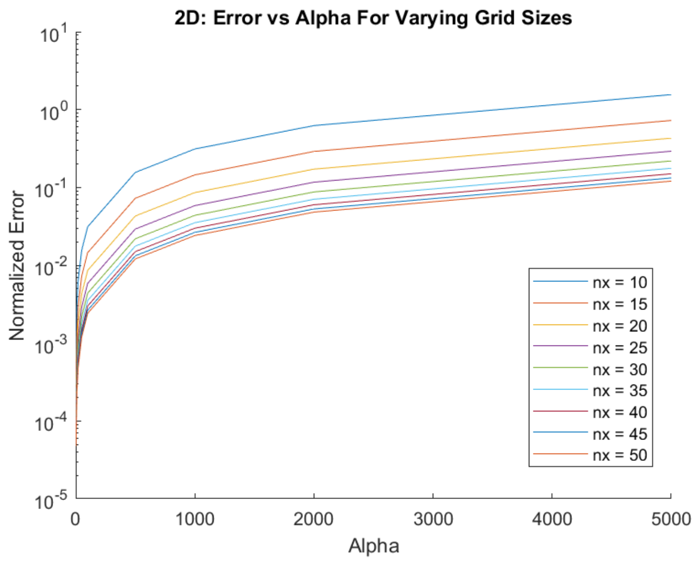


Figure 15: Log Error With Alpha



Conclusions

- LU Solver - The grid resolution improves the results obtained from the solver. A resolution of 10 (~100 grid points) for the 2-D problem gives us a coarse and blocky solution, while a resolution of 50 (~2500 grid points) gives us a much smoother result. The 1-D solution does not change too much with the increase in resolution.

- Time Scaling Analysis - The time plots follow a power law, with varying scaling factors across 1-D and 2-D grids and the intrinsic and user-defined solver. The user-defined solver performs within expectations of the theory (~ N^3), while the intrinsic solver performs much better.
- Error with grid-size - The error falls off asymptotically with the grid-size in the 2-D problem. It follows a N^{-1} scaling.
- Error with Alpha - The normalized error increases linearly with α for the 2-D problem. The same trend is followed across all the grid resolutions with no deviation.
- It's clear that a larger grid size gives us better results, however, a compromise needs to be made between the error and the time taken to perform the simulation. For the 2-D case, a grid size of around 30 (~1000 points) seems to be the best fit here. It is located on the elbow of the time plot and the error plot, hence any grid size beyond that would result in a small decrease in error while taking more time.
- The user-defined routine is not stress-tested for different matrices. Here, we only test it against two kinds of sparse matrices, where the intrinsic solver far outperforms the user-defined one.
- A more refined version of this solver would have to ensure that the problem does not scale as poorly for the 2-D case. The scaling was ~ 4, which can result in extremely poor performance for a larger grid size.
- A better algorithm would take advantage of the sparse structure of the matrix. The LU routine is a general method that can be applied to most solvable systems, but it may not be the most efficient.

Appendix: Code

Main Code For Running LU Solver and Recording Time and Error (2-D)

```
grid = 10:5:10;
lu_user = zeros(size(grid)); %record times
lu_mat = zeros(size(grid));
alpha = [0 2 10 20 50 100 500 1000 2000 5000];
error_alpha = zeros(size(alpha,2),size(grid,2));
x_all=zeros(size(grid,2),grid(end)+1);
for i=1:size(grid,2)
    for a=1:size(alpha,2)
        nx = grid(i);
        ny = grid(i);
        x_ = linspace(0,1,nx+1);
        y_ = linspace(0,1,ny+1);
        [A,b] = matrix(nx, ny,alpha(a));
        s=(nx+1)*(ny+1);
        x=zeros(s,1);
        tol=1e-5;
        err=0;

        tic
        x = ludecomp(A,b,s,tol,x,err);
        x_all(i,1:s) = x;
        lu_user(i) = toc;

        tic
        [l,u] = lu(A);
        z = u\(l\b);
        lu_mat(i) = toc;

        Tanal = analytic(nx,alpha(a),2);
        error_alpha(a,i) = norm(Tanal-x)/(nx+1);

        T_mat = reshape(x,[nx+1,nx+1]);
        %%PLOT
        figure('Visible','off');
        [C,h] = contour(x_,y_,T_mat);
        title("2-D: nx,ny = "+nx+", "+ny+"; \alpha = "+alpha(a));
        xlabel('x');
        ylabel('y');
        c.LineWidth = 2;
        clabel(C,h);
        saveas(gcf,"figures\alpha\2d"+nx+"alpha"+alpha(a)+".png");
    end
end
```

Forming the 2-D Problem Matrix

```
function [A,f] = matrix(nx,ny,alpha)
dx=1;
dy=1;
s=(nx+1)*(ny+1);
%Size - (nx+1)*(ny+1) x (nx+1)*(ny+1)
x=((nx/dx)^2)*ones(s-1,1); %left and right of diagonal
y=-2*((nx/dx)^2 + (ny/dy)^2)*ones(s,1); %diagonal
p=((ny/dy)^2)*ones(s-(nx+1),1);
A=diag(p,-(nx+1))+diag(x,-1)+diag(y)+diag(x,1)+diag(p,(nx+1));
f = -alpha*ones(s,1);

%Replacing boundary values
A(1:(nx+1),:) = eye((nx+1),s);
A(end-nx:end,:) = rot90(eye((nx+1),s),2);
f(1:(nx+1)) = 0;
f(end-nx:end) = 0;
for i = (nx+2):(ny*(nx+1))
    if mod((i-1),(nx+1))==0 || mod((i),(nx+1))==0
        A(i,:) = zeros(1,s);
        A(i,i) = 1;
        f(i)= 0;
    end
end
```

Forming the 1-D Problem Matrix

```
function [A,f] = matrix1D(nx,alpha)
dx=1;
s=(nx+1)*(1);
x=((nx/dx)^2)*ones(s-1,1); %left and right of diagonal
```

```
y=-2*((nx/dx)^2)*ones(s,1); %diagonal
A=diag(x,-1)+diag(y)+diag(x,1);
f = -alpha*ones(s,1);

%Replacing boundary values
A(1,2) = 0;
A(end,end-1) = 0;
f(1) = 0;
f(end) = 0;
```

Analytical Solution Calculation

```
function T = analytic(nx,alpha,dim)
h = 1/nx;
i=1;
if dim == 2 %For 2-D
    T = zeros((nx+1)*(nx+1),1);
    for x=0:h:1
        for y = 0:h:1
            T(i) = Tp(x,alpha)+Th1(x,y,alpha)+Th2(x,y,alpha);
            i=i+1;
        end
    end
else %For 1-D
    T = zeros((nx+1),1);
    for x=0:h:1
        T(i) = Tp(x,alpha);
        i=i+1;
    end
end
return;

function tp = Tp(x,alpha)
tp = -(alpha/2.0)*(x^2 - x);
return;

function sum = Th1(x,y,alpha)
sum = 0;
for n= 1:10
    sum = sum + (2*alpha*(((-1)^n)-1)*sinh(n*pi*y)*sin(n*pi*x)/(pi^3 * n^3
*sinh(n*pi)));
end
return;

function sum = Th2(x,y,alpha)
sum = 0;
for n = 1:10
    sum = sum + (2*alpha*(((-1)^n)-1)*sinh(n*pi*(y-1))*sin(n*pi*x)/(pi^3 * n^3
*sinh(-n*pi)));
end
return;
```

ludecomp.m

```
function x = ludecomp(a,b,n,tol,x,err)
o = zeros(n); %records row interchanges
s = zeros(n); %Stores max elements of each row
[a,o,err] = decomp(a,n,tol,o,s,err);
if err ~= -1
    x = substitute(a,o,n,b,x);
end
end
```

decomp.m

```
function [a,o,err] = decomp(a,n,tol,o,s,err)
for i = 1:n %Going through each row
    o(i) = i; %current row
    s(i) = abs(a(i,1)); %first element of row
    for j = 2:n %going through element of row
        if abs(a(i,j))>s(i)
            s(i)=abs(a(i,j)); %finding max element of each row
        end
    end
end
for k = 1:(n-1)
    o = pivot(a,o,s,n,k);
    if abs(a(o(k),k)/s(o(k))) < tol %If even the biggest element is still lower than
tolerance
        err = -1;
        disp(a(o(k),k)/s(o(k)));
        break
    end
    for i = (k+1):n %Start reduction operation
        factor = a(o(i),k)/a(o(k),k);
        a(o(i),k) = factor;
        for j = (k+1):n
            a(o(i),j) = a(o(i),j) - factor*a(o(k),j); %Subtraction rows
        end
    end
end
if abs(a(o(k),k)/s(o(k))) < tol %If last element is lower than tolerance. (Singular
Matrix)
    err = -1;
    disp(a(o(k),k)/s(o(k)));
end
end %function end
```

pivot.m

```
function [o] = pivot(a,o,s,n,k)
p = k;
big = abs(a(o(k),k)/s(o(k))); %current pivot / max
for ii=(k+1):n %Search in the following rows
    dummy=abs(a(o(ii),k)/s(o(ii))); %To check if bigger
    if dummy > big
        disp("Pivot")
        big = dummy;
        p = ii;
    end
end
%Reorder and record position using o
dummy = o(p);
o(p) = o(k);
o(k) = dummy;
end
```

substitute.m

```
function x = substitute(a,o,n,b,x)
for i = 2:n
    sum = b(o(i));
    for j = 1:(i-1)
        sum = sum - a(o(i),j)*b(o(j));
    end
    b(o(i)) = sum;
end
x(n) = b(o(n))/a(o(n),n);
for i = (n-1):-1:1
    sum = 0;
    for j = (i+1):n
        sum = sum + a(o(i),j)*x(j);
    end
    x(i) = (b(o(i)) - sum)/a(o(i),i);
end
end
```