Ilia Kheirkhah

ENM 5020

2/14/2023

Homework 1

Comparison of Doolittle Decomposition Algorithm with MATLAB Intrinsic LU Decomposition

Contents

Introduction

One of the most fundamental problems in mathematics, science, and engineering is finding the solutions to a linear system of equation, $\boldsymbol{A}\underline{x} = \underline{b}$. Solving this problem is fundamental to many numerical schemes ranging from the Finite Difference Method we explore to iterative methods to machine learning and various other fields. Due to the importance this equation, many people have spent time finding better ways to solve the system given certain structures for our $\boldsymbol{A}$ matrix.

Depending on what values each index takes in matrix $\boldsymbol{A}$, we can characterize the matrix differently. For example, the matrix could be spare if many of the terms of zero or dense if many of the terms are nonzero. The matrix could be banded if all nonzero values fall into a band of a certain width from the main diagonal. Given one knows the structure of their matrix, they can take advantage of the organization to provide faster solutions to a system of equations.

In this example, we will be focusing on one method of solving systems known as LU, or Lower-Upper, Decomposition. This approach works for all non-singular matrices and allows for a faster computation time than simply finding the inverse of $\boldsymbol{A}$**.** The fundamental idea of LU Decomposition is to factor our matrix into a lower triangular and an upper triangular matrix such that:

$$\boldsymbol{A = LU} \tag{1}$$

Lower triangular and upper triangular matrices in systems of equations can be solved simply and quickly using forward or backward substitution respectively. This means that LU Decomposition allows us to do the difficult part of factoring a matrix and apply the factorization to various values of $\underline{b}$. This allows us to quickly solve similar equations without repeating needless calculations.

There can be many ways to code a given mathematical technique. For this report, we have chosen to implement Doolittle's Algorithm for LU Decomposition along with partial pivoting, forward substitution, and backward substitution. This allows us to solve many systems of equations and handle nearly singular matrices fairly well in most cases. Doolittle's algorithm will factor any matrix into a lower and upper triangular matrix. This generality is a double edged sword. Although we can now tackle many problems with the same algorithm, we do not take advantage of the structure of provided matrices to lower the computation time or storage requirements of our calculation. Due to the vast amount of storage modern computers have relative to the scale of the problems we will be working on, this report will focus less on storage issues and more on computation time.

Assume that $\boldsymbol{A}$ is a real matrix with dimension n. This means that $\boldsymbol{A} \in \mathbb{R}^{n\times n}$. We will not assume anything about the structure of the matrix as LU Decomposition is structure-agnostic. If we were to a solve a system involving $\boldsymbol{A}$ by simply finding its inverse and multiplying, we would expect an operation count of

$$oc^1 = n^3 \quad \textbf{(2)}$$

Although it is rare to reduce the order of the operation count, one can show that LU decomposition has an operation count of:

$$oc = 2n^3/3 \quad \textbf{(3)}$$

This can be a significant difference in runtime for some routines.

This is the fundamental idea behind LU Decomposition. As we will explore in this report, it is a very powerful tool for solving systems of equations including those derived from Taylor Series representations of differential equation, allowing us to get approximate solutions to complex problems in relatively little time.

Methods

In this section we will explore in more detail the implementation and usage of the important tools we developed over the course of this assignment. We will begin with Doolittle's Algorithm with pivoting and forward/backward substitution. After this we will explore the technique used to translate a differential equation to a linear system of equations which we can then apply LU decomposition to. Finally, we will explore the errors associated with these methods. This section will remain purely theoretical as applications of this code and the associated graphics will be in the results section.

LU Decomposition with Doolittle's Algorithm

The first step in beginning this assignment was to create code that executes LU Decomposition with Doolittle's Algorithm. This was done by implementing the four functions we were provided pseudocode for. The primary function was called *Ludecomp*. The primary inputs of this function were our $\boldsymbol{A}$ matrix, our $\underline{b}$ vector, the size of our vector, and our tolerance. Tolerance is the value a diagonal element has to be, after pivoting, for the algorithm to claim the matrix is too close to singular to give us accurate an accurate solution. For the course of this assignment, we have set this tolerance to 10^{-6}.

Although the primary function we interface with is *Ludecomp*, it does very little of the decomposition itself. The primary work is done in the *Decompose* function. This loops through our matrix, adjusting the values as appropriate to return the upper and lower triangular matrices. It is important to notice that in this algorithm, the upper and lower triangular matrices are stored in a single matrix. This is allowable since they are almost independent of each other, overlapping only on the main diagonal. However, since we are using LU Decomposition, it is known that the lower triangular matrix will have all its main diagonal indices have values of 1. Given this, we can leave them out and allow the upper triangular matrix to also inhabit the main diagonal of our return matrix.

Over the course of the *Decompose* function, we will make use of the *pivot* function to ensure that all the diagonal terms are the maximum value they can be to postpone any singularity issues the

[1] Note here that *oc* stands for operation count.

matrix could have. This is done by doing a simple row swap after finding the maximum value below the main diagonal in a given column. The problem with this pivoting approach is that it can mix the order of the result variables. For example, if we swap rows three and four in $\boldsymbol{A}$**,** we must also remember to swap the same rows in $\underline{b}$ to keep all the equations true. This is accounted for by a change of row variable in *Decompose* which gets passed around through the other functions.

Once the Decomposition is complete, all that is left to finish the process is to backwards and forward substitute in to retrieve our values for $\underline{x}$. This is done in the *Substitute* function, which is only called if *Ludecomp* confirms that the decomposition was successful and non-singular. Once this is all complete, we have solved our system of equations and can process the results as necessary.

Finite Difference Method

For this assignment, we applied the LU Decomposition to find numeric solutions to the one-dimensional and two-dimensional diffusion problem:

$$\nabla^2 T + \alpha = 0 \text{ with } T = 0 \text{ on all boundaries} \quad \textbf{(4)}$$

We solved these problems on the domains $D = (0 \leq x \leq 1)$ for the one-dimensional case and $D = (0 \leq x \leq 1) \cup (0 \leq y \leq 1)$. Currently, our implementation of the LU Decomposition does not have the ability to solve continuous differential equations. As a result, we need a way to translate our provided differential equation into a set of linear equations which we can then solve. This is done by using the Finite Difference Method (FDM).

To apply Finite Difference Method, we discretize our domain into n spaces and $n + 1$ nodes. We then want find the values at each node and, provided we have enough nodes, we can say that we have something very close to the true solution. It is also useful to define $h = 1/n$ as the size of the spaces. This will be a useful tool making the equations of FDM and analyzing the error associated with FDM.

Once we have discretized, we need a way to represent our first and second derivative. This is done by using the forward and backward Taylor series we can find approximations for derivatives. For our applications, the approximations we will use are:

$$\frac{dT_i}{dx} = \frac{T_{i+1} - T_{i-1}}{2h} + O(h^2) \quad \textbf{(5)}$$

$$\frac{d^2T_i}{dx^2} = \frac{T_{i+1} - 2T_i + T_{i-1}}{h^2} + O(h^2) \quad \textbf{(6)}$$

Notice that in both equations, we have an extra term $O(h^2)$. This is to say that the error associated with discretization in this way is proportional to h^2. We will discuss error in more detail in the next section.

When applying this to a one-dimensional function, the use is obvious. However, one needs to be slightly clearer when working in two dimensions. In this case, instead of being able to identify our location with one index, i, we need to have two indices, i and j. i represents our position in the x direction while j represents our position in the y direction.

If we want to take the partial derivative in the x direction, we would use indices $(i,j), (i+1,j)$ and $(i-1,j)$. This would be the same as moving horizontally through our grid. Similarly, when taking the partial derivative in the y direction, we would use indices $(i,j), (i,j+1)$ and $(i,j-1)$. This is akin to moving vertically through our grid. Notice here that applying these equations on the boundaries, namely when i or j is equal to 1 or $n+1$, we will run off the grid and be unable to find the derivatives we would need. However, this is where our boundary conditions come in. Instead of applying the provided differential equation on the boundaries, we enforce our boundary conditions.

A major problem with the current syntax we are using is that, in the two-dimensional case, we cannot have a variable depend on two indices and also create a square matrix with an easy to create structure. This means that we need to find a way to convert our (i,j) index system into a single variable index system (l). This can be done simply with the following mapping:

$$l = (j-1)(n+1) + i \quad \textbf{(7)}$$

Since we have a square array of grid points, the order in which we count does not matter. As such, we can say this is analogous to counting horizontally through each row until you reach the end, and then going back to the beginning of the row above in a sort of typewriter numbering scheme. Although it is necessary to use this numbering to solve our system, it is difficult to graph this numbering due to the nature of the language we are using. To ease this issue, we need to develop a way to go from the (l) system to the (i,j) system. This is done by taking advantage of the modulo operator, $mod(a,b)$ which returns the remainder of a/b. For example, the value of $mod(7,3)$ is 1. Using this operator, we can then say:

$$i = mod(l-1, n+1) + 1 \quad \textbf{(8)}$$
$$j = (l-k)/(n+1) + 1 \quad \textbf{(9)}$$

Together, these equations allow us to generate our $\boldsymbol{A}$ matrix for our discretized diffusion equation and then translate our solution back into a format we can easily plot and manipulate for error analysis.

It is important to note here that, in the two-dimensional case, our grid size will be $(n+1) \times (n+1)$. However, since each grid point has an associated equation, this means that our matrix will be $(n+1)^2 \times (n+1)^2$. We can say $(n+1)^2 = N$.[2] One can then notice that our runtime will scale with N^3 but our error will only scale with $1/n^2$. Our runtime will grow much faster than our error will drop in a two-dimensional situation. In a three-dimensional simulation using finite difference method, we would have to say $N = (n+1)^3$, exaggerating the runtime problem even more. This is something to be conscious of when setting up simulations to run.

[2] If N is ever used, it always refers to the dimension of the matrix or the total number of grid points, which will always be equal.

Error Calculation

As we have discussed before, we expect our error to scale as $O(h^2)$. In this report, we would like to validate this while also developing a method of analyzing error for other problems. Ideally, we would have an analytic solution to compare our numeric result to. This would allow us to obtain an exact error which we could graph to demonstrate the scaling trends. However, if we have an accurate, analytical result, there is no point in spending time to create and process a numeric solution. In many cases where numeric simulations are used, an analytic solution is not possible.

To solve this problem, we take what one might consider to be an obvious step. Instead of comparing our result to an analytic solution, we compare our result against a finer mesh. Given a sufficiency fine mesh, one can ignore the error associated with the reference result as it will be significantly smaller in magnitude than the actual error. The major problem with a reference solution is when the current grid and the reference grid to not line up nicely with one another. In this situation, some points on the less fine grid may not coincide with a point on the finer grid. This means that we have to introduce extra error to interpolate to our point of interest when calculating error. To make this easier, it is often good to pick numbers that are multiples of one another. For the sake of this assignment, I have chosen to use $n = 120$ as the reference 2D grid. This gives me a $121^2 = 14{,}641$ grid points to work with. In addition, due to the high number of divisors 120 has, we can get several data points to see good trends in the error graph without needing to calculate multiple reference grids.

It should be clear that our coarse grid of interest will never be the same size as our fine reference grid. This means that, when calculating error, we cannot simply subtract the two results. Instead, we need to find the proper indices that overlap. For example, consider a one-dimensional discretization with $n = 60$ and a reference solution with $n = 120$. Although the solutions coincide at $i = 1$, $i = 2$ on the coarse grid would coincide with $i = 3$ on the fine grid. Similarly, $i = 3$ on the coarse grid would coincide with $i = 5$ on the fine grid. This pattern continues and we can say $i_{fine} = 1 + 2i_{coarse}$. More generally, for a one-dimensional solution, we can say:

$$i_{fine} = 1 + \frac{n_{fine}}{n_{coarse}} i_{coarse} \tag{10}$$

One may think that this could also be applied to the $2D$ case while in the (l) numbering system. However, this does not work due to the jumps in the ordering system. Instead, it was found easiest to translate the solution back into the (i, j) numbering system. At this point, we can apply equation 10 to both the i and j components to get the proper indices. That is to say:

$$(i, j)_{fine} = (1 + Si_{coarse}, 1 + Sj_{coarse}) \tag{11}$$

$$S = \frac{n_{fine}}{n_{coarse}} \tag{12}$$

Notice that we make use of the same scaler value S in equation 11 as we do in equation 10.

After going through our solution and extracting the error, we can make an error vector. However, to compare errors among all the grid sizes, we will need to normalize in some way. Since we have chosen the L-2 Norm as the norm we will use to find magnitude, we need to find a compatible factor to normalize by. We know that the L-2 Norm is defined as:

$$e = \sqrt{\sum_{i}^{N} \varepsilon_i^2}$$

If we, for the sake of ease, assume that $\varepsilon_i$is some constant, one can see that:

$$e = \sqrt{N}\varepsilon$$

This means that if we want to normalize by some value, we should divide by $\sqrt{N}$ when using the L-2 Norm.

Results

In this section we will analyze the results generated by our LU decomposition when applied to the finite difference interpretation of the one- and two-dimensional diffusion equations. We will first look at how refining the grid influences the appearance of the numeric solution for various grid sizes in 1D and 2D. We will then explore how the solution time scales with refining grid sizes. Finally, we will take a more analytical view into our solutions and analyze the error of various sizes when varying both the grid size and the value of α.

Numerical Solutions to BVP

The first problem we approached with our *Ludecomp* function was the heat equation. Specifically, we chose to solve this differential equation for when $\alpha = 2$. In this section, we will look at the solutions and how grid refinement influences the plot visually. We will later quantify any error present.

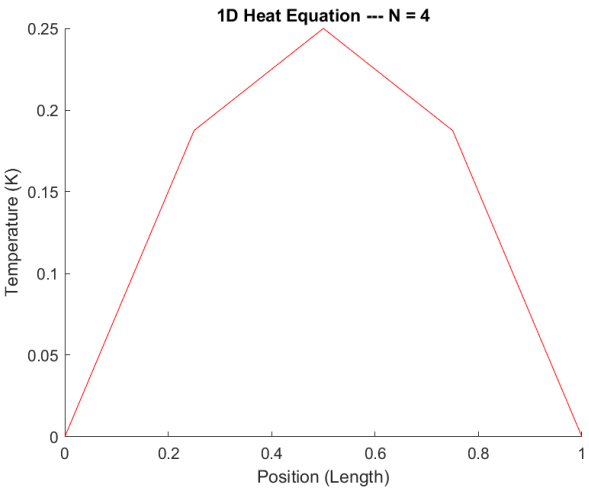

Figure 1a

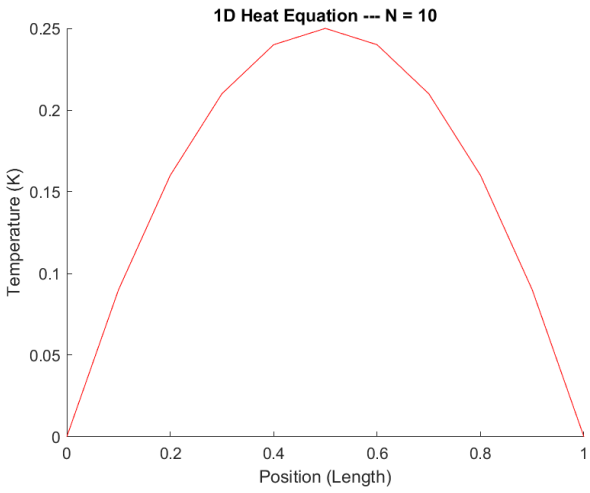

Figure 1b

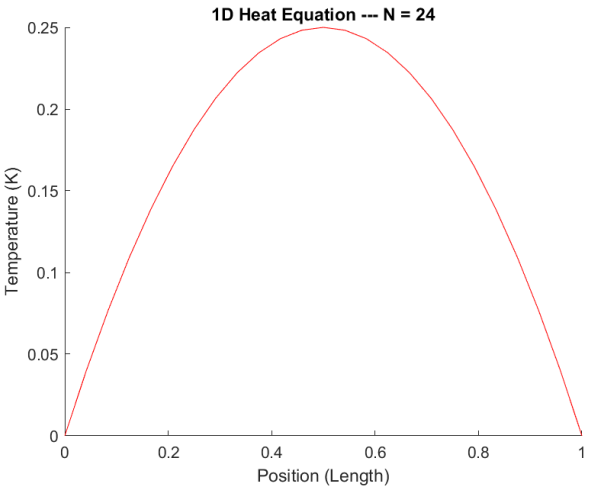

Figure 1c

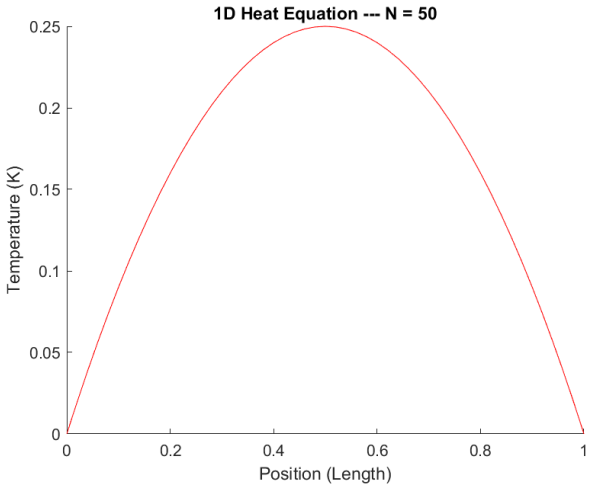

Figure 1d

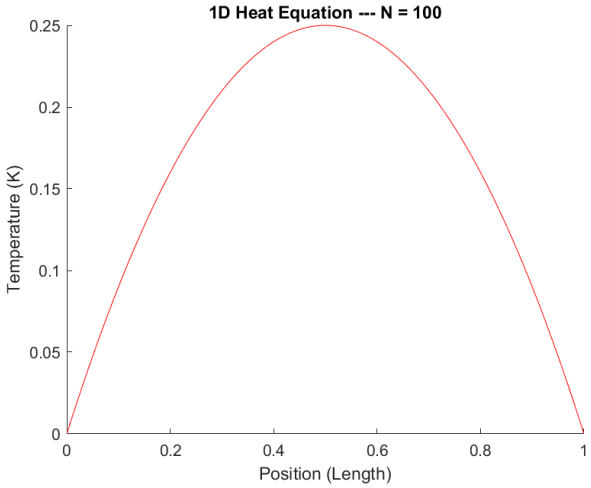

Figure 1e

As can be seen, the visual smoothness of the graphs increases very quickly. There is significant improvement from Figure 1a to 1b. There is noticeable but not as significant smoothing from Figure 1b to 1c. However, the change from Figure 1c to 1d is barely noticeable. The change from Figure 1d to 1e is imperceptible. As such, when going for the purpose of solving this type of differential equation is purely visual, less grid points can be warranted.

It is important to note that this analysis of visual smoothness is qualitative and only comparable to other similarly small α values on similar domains. If we were to increase the value of α or to increase the size of the domain, it should be expected that a finer mesh would be necessary for a similar smoothness.

One feature of interest is the fact that all solutions, regardless of grid refinement, located the maximum value properly. Provided, I ensured that there would be a grid point where I expected the maximum value to be. However, no grid size resulted in the predicted maximum value to be different than the true maximum value.

Just as we have solutions for one-dimensional problems, we can apply our code to a two-dimensional system with minor adjustments to how we construct our matrix. Implementing this code we can see the

results in Figure 2. This is where we see more refinement come with grid size. With each increasing grid size, we can see the corners of the counter be filled more and more. This is expected as we should only have the exact boundaries at zero temperature.

In addition, we see the shape of the graph become much more smooth. In Figure 2a, the center of the counter looks almost like a square. As we progress to a finer grid, we can see the center become much more round until Figure 2d where the center is much more smooth. It is interesting to note that although the centers of Figures 2b and 2c look fairly smooth, the coarseness of the grid shows itself in the outer counters which are noticeably polygonal, especially at the corners. Compare this to the entirely smooth visual of Figure 2d and you notice the importance of a fine grid.

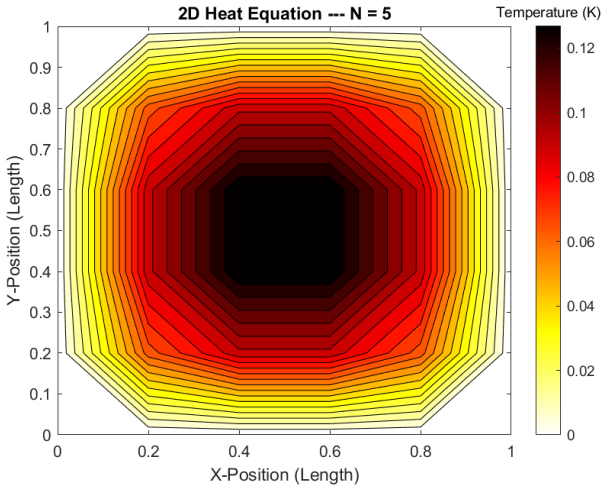

Figure 2a

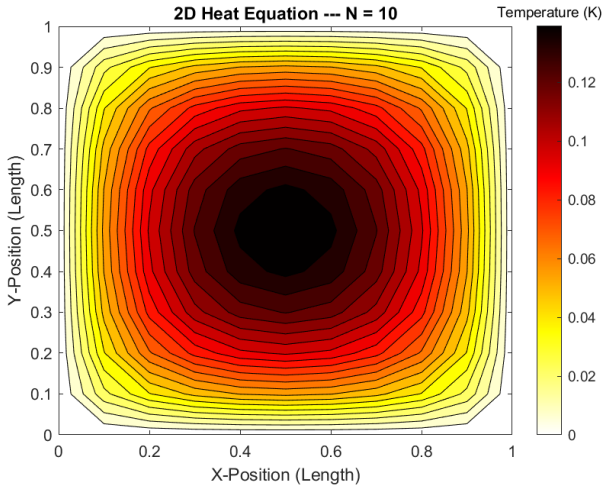

Figure 2b

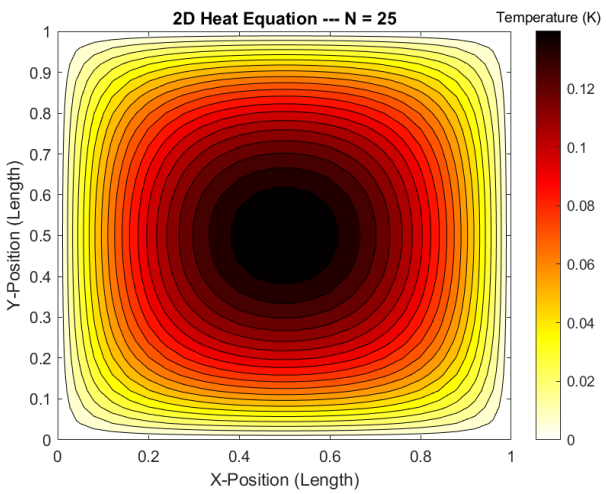

Figure 2c

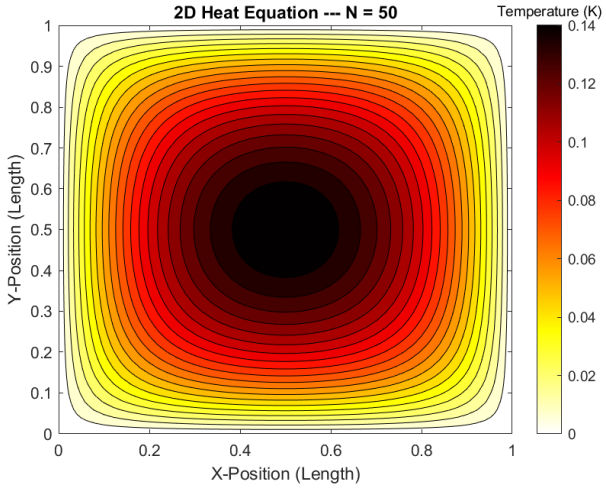

Figure 2d

As can be seen, refining the grid can have great impact on the smoothness of the graph. Regardless of what we can see with human vision, making the grid finer will result in piecewise curves that more and more closely approximate a truly smooth, continuous, solution. However, we cannot increase our grid size arbitrarily. Ignoring the problem of size, the more pervasive problem with increasing your grid size is time. The finer the grid, the longer the simulation will take. As we know from theory, and as we will confirm over the course of the next section of this report, we expect the simulation time to scale with the

number of grid-points cubed. That means doubling the number of grid-points you have leads to eight times increase in time. This is the primary reason why I chose not to have a 100x100 grid to show for the two-dimensional heat equation. The 50x50 grid took about a minute to solve. A 100x100 grid has four times as many grid points and will therefore take sixty-four times as much time. This means that solving a 100x100 grid point heat equation would require me to spend at least an hour, assuming my MATLAB could solve the problem without running into any runtime or storage issues.

Timing

As we discussed, our operation counter, and therefore our runtime, scales with $\frac{2}{3}N^3$ according to theory. In this situation, N is the total number of gridpoints. However, there is always a gap between theory and real application. As such, we want to ensure that Doolittle's Algorithm fits this mold, at least up to the power. Although the constant term is important, especially when it comes to comparing LU Decomposition against inverting a matrix, it is not particularly useful when it comes to estimating the real life expected runtime. It is much simpler to consider a reasonably sized time sample and multiply it by the appropriate scale depending on the relative size of the problems to get an approximate runtime, as I did at the end of the previous section. Although this is not perfectly accurate, it can give an order of magnitude approximation quickly to help in deciding if a simulation is feasible or not.

To ensure our approximations are accurate, we need to understand the scaling of our implementation of LU Decomposition. To do this, we took advantage of the built in Tic and Toc commands MATLAB provides for timing. We can the time against the number of grid points on a log-log scale. The reasoning behind this goes as follows. We know our operation count should obey the following relationship:

$$oc = 2n^3/3 \tag{13}$$

However, we cannot ensure the constant coefficient term and exponent are what our theory states. We can reasonably state that our operation count will take the form:

$$oc = cn^p \tag{14}$$

If we take the log of both sides, we can say:

$$\log_{10} oc = p \log_{10} n + \log c \tag{15}$$

Since we are using runtime as a proxy for operation count and picking the number of grid-points we use, one can notice this is the equation for a line. This means that our slope will be the scaling factor we expect. In our case, we expect this value to be approximately three for LU Decomposition algorithms.

It is important to note here that the matrices we create by applying the Finite Difference Method are banded matrices. More specifically, the one dimensional heat equation produces a tridiagonal matrix while the two dimensional heat equation produces a matrix of bandwidth $p = 2(n + 1) + 1 = 2n + 3$. In theory, we could take advantage of this matrix structure to speed up our solving algorithms. For a banded matrix, we expect the operation count to be:

$$oc = np^2 \tag{16}$$

This would be significantly better than LU Decomposition for our applications. However, for simplicity we were provided the code for LU Decomposition which works for all non-singular matrices. However, it is important to note the slight loss of time that would have ideally been saved if the application was for significantly larger systems.

Given all of this information, we can now look at the runtimes for both the one dimensional and two dimensional systems. We can also compare this runtimes with the built-in MATLAB LU solver by comparing the slopes of each log-log graph when fitted with a linear trendline.

Looking at the graphs which show our implementation of Doolittle's Algorithm, we see slopes of around 3.1 in Figure 3a and 3b, just over our expected value of 3. This shows that our implementation is accurate to theory. The slight increase in runtime is likely a result of hardware constraints and higher order terms that cloud the results at this scale.

On the other hand, MATLAB's LU solver is significantly faster than the theoretical best of cubic scaling. Having slopes of just under or around two for Figure 3c and 3d are unexpected for any LU decomposition algorithm. Although I have not analyzed the source code, I believe that a likely reason for this discrepancy is MATLAB taking some advantage of the structure of our matrix without us knowing. This could be a result of how MATLAB stores large matrices that contain many zeros. It is possible that MATLAB is skipping over many of the zeros, not because of how the algorithm itself is coded, but due to how MATLAB stores these seas of zeros.

Remember, when writing matrices, it is possible to write a matrix as follows:

$$[\overrightarrow{x_1} \quad \overrightarrow{x_2} \quad \overrightarrow{x_3}] \tag{17}$$

Where $\overrightarrow{x_i} \in \mathbb{R}^n$ are all column vectors. In this situation, we know that this matrix represents a $n \times 3$ matrix. We could do a similar scheme for filling out a recognizable matrix in a larger matrix. Consider things such as the identity matrix or matrices used in our work. If MATLAB uses a similar process for storing large sets of zeros, it would likely explain some of the time save we have, although I am not sure if this alone is enough to explain the entire difference.

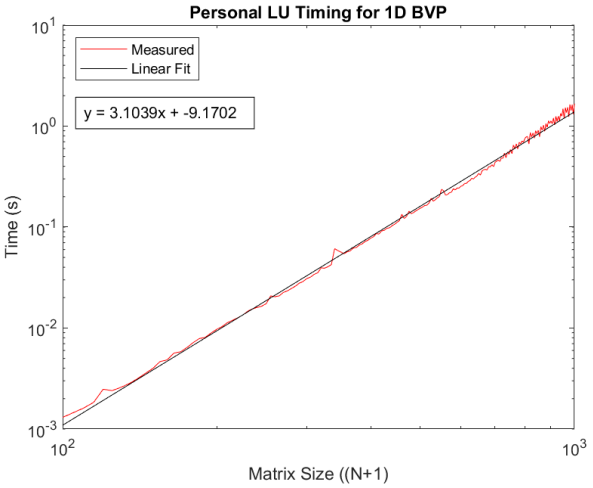

Figure 3a

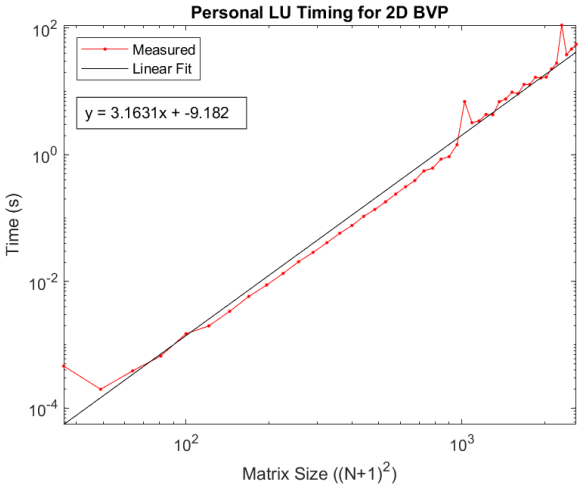

Figure 3b

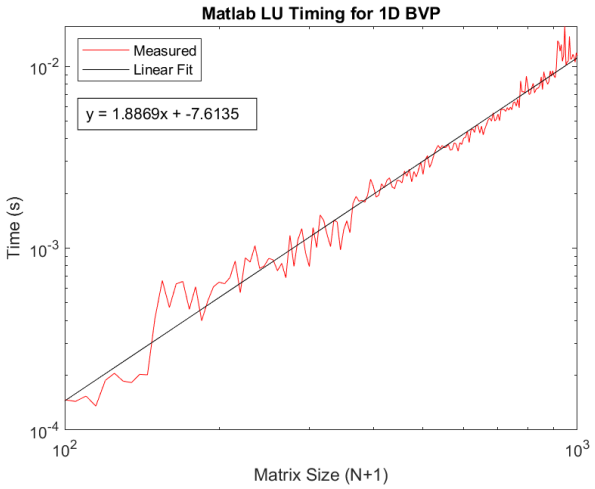

Figure 3c

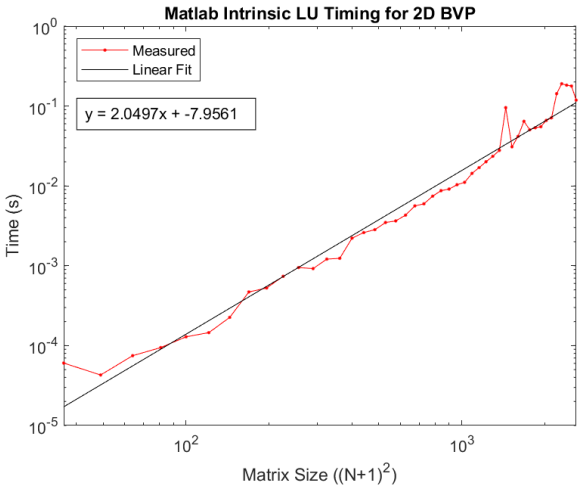

Figure 3d

Error Calculation

So far, we have shown that increasing the number of grid points makes the graph visually smoother at the cost of significant time scaling. However, as we discussed in our methods, the primary reason we want to make the grid finer is to reduce error in our simulation so we can get more accurate data to use in the real world.

When we apply this, we will often want to approximate how much error we have in a given mesh. However, when an analytical solution is not possible, our best course of action is to compare against a finer grid. This is exactly what we do here. As mentioned before, I chose a reference grid size of 121x121 grid-points. This allowed was within the scope of grid-points that my computer could solve and gave me a nine times finer mesh than what it was used to compare against. This corresponds to three times as many grid points in each direction for a two dimensional mesh. We know that our error scales with $1/n^2$ according to theory, where n in this situation is the number of grid points in one direction. This means that our reference mesh has 1/9th the error our finest mesh will have, resulting in at most an approximately ten percent error. For the sake of validating trends, I believe this is acceptable.

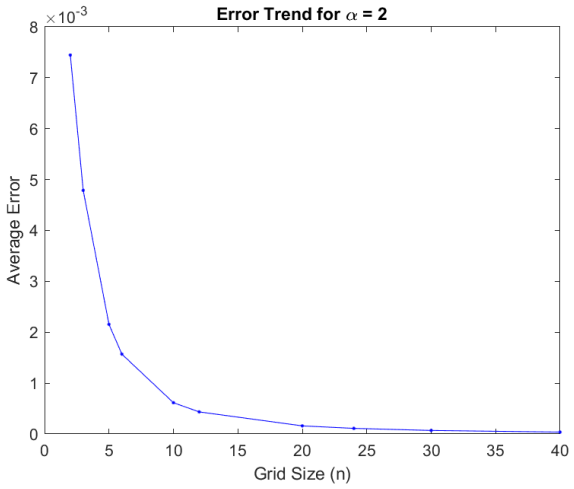

Figure 4a

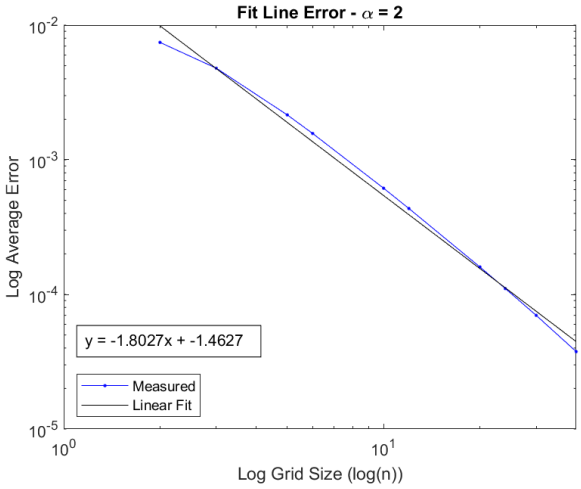

Figure 4b

Looking at Figure 4a, we can see the true trend of our error for $\alpha = 2$. Although it looks to be the correct shape, we can confirm the trend behavior by plotting on a log-log scale as we did to analyze time scaling. By identical logic, by fitting a line to our data we can extrapolate how our error scales from the slope. From Figure 4b, we can see the slope of our log-log data to be approximately -1.8. Although not exactly 2, this is only a 10% error from our expected result. It is likely that more grid points would result in the slope trending more negative and resulting in the -2 slope we expect. As such, I believe our theory is accurate.

Although we have been saying our error scales with $1/n^2$ orh^2, this loses some of the depth as we have excluded constant terms for the sake of simpler equations. These constant terms depend on the higher order derivatives of our solution and the maximum value of certain terms in our differential equation. The exact details are not important, but what is relevant for our discussion is that this constants can depend on α. As such, although our error will always scale with h^2, our normalized error may be larger in magnitude depending on the value of α. This can clearly be seen in Figure 5. The error we have in Figure 4 corresponds to the light blue line in Figure 5. However, as we increase α, we see that our errors all increase in magnitude. Although you cannot tell the curves apart for finer grids, it is clear to see that the value of α has extremely powerful implications for the error we expect to see in our simulations.

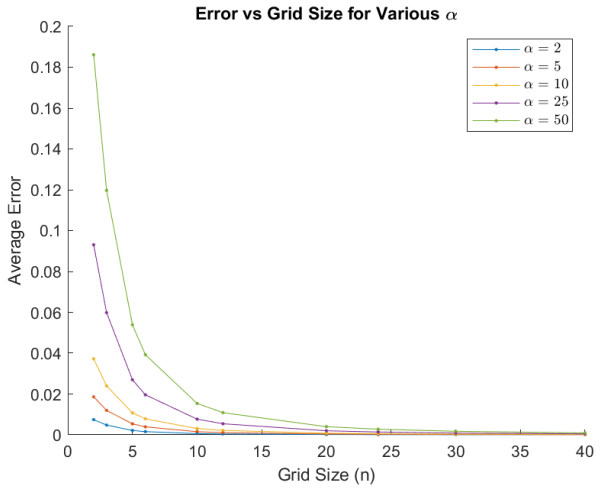

Figure 5

Discussion

Solving linear systems of equations shows up in an uncountable amount of real world applications. Understanding how they work allows for engineers to make judgements on the validity and possible uses of computational methods in the real world. We have seen that refining our grid leads to more accurate results. As such, engineers will like as fine a grid as they can get to be able to reliably apply their results to the real world. However, everything has a cost. For refining a grid, the cost is time. I believe one of the most important takeaways from this assignment is that solving linear equations takes some amount of time that scales with N^3. This means that a coarse simulation which took a couple hours can take days or weeks if the grid is refined carelessly, possibly leading to unsolvable systems not due to the nature of the problem but due to computational limits.

As we will explore over the remainder of this course, clever tricks can be used to reduce complex problems such as non-linear systems of equations into linear systems that we can then solve to get sufficiency accurate solutions. At the root of many more sophisticated computational schemes is often solving a set of linear equations. Given we do not have knowledge of the structure of our matrix a priori in these sophisticated schemes, LU decomposition becomes a great tool for solving almost any system of linear equations and this assignment has given us the tools necessary to do so.

Appendix

Main File

This is the file where we do all the problem solving. We have done Problems 1 and 2. Our Solver is complete and we can make the discretized heat equation. Let us solve our other stuff.

Contents

Problem 3 - Numerical Solutions for the 1D and 2D BVP

Here we solve the heat equation for alpha = 2 with various grid sizes. We graph our results and compare how they look.

```
% 1D solution
clear; clc; close all;
alpha = 2;
nset = [4,10,24,50,100];
fileTemplate = "1D_N=";
for i = 1:length(nset)
    N = nset(i);
    [A,b,x] = Heat1D(alpha,N,0,1,0,0);
    T = Ludecomp(A,b,N+1,10e-6,zeros(N+1,1));

    figure;
    hold on;
    plot(x,T,'r-');

    xlabel('Position (Length)')
    ylabel('Temperature (K)');
    title("1D Heat Equation --- N = "+num2str(N));
    xlim([0 1]);
    ylim([0 .25]);
    fileName  = fileTemplate + num2str(N)+".png";
    exportgraphics(gcf,fileName);

end

% 2D solution
nset2 = [5,10,25,50];
fileTemplate = "2D_N="
for i = 1:length(nset2)
    N2 = nset2(i);
    [Aheat2,bheat2,x,y] = Heat2D(2,N2,0,1,0,1);
    tres = Ludecomp(Aheat2,bheat2,(N2+1)^2,10e-6,zeros((N2+1)^2,1));

    t2d = zeros(N2+1);
    res = zeros(length(tres),3);
    for l = 1:length(tres)
        k = mod(l-1,N2+1)+1;
        j = (l-k)/(N2+1)+1;
        res(l,1)=l;
        res(l,2)=k;
        res(l,3)=j;
        t2d(k,j)=tres(l);
    end
```

```
    figure;
    contourf(x,y,t2d,20);
    colormap(flipud(hot));
    tempbar = colorbar;
    tempbar.Title.String = "Temperature (K)";
    xlabel('X-Position (Length)')
    ylabel('Y-Position (Length)');
    title("2D Heat Equation --- N = "+num2str(N2));

    fileName = fileTemplate + num2str(N2) + ".png";
    exportgraphics(gcf,fileName);
end
```

Problem 4 - Timing of Personal vs. Matlab LU Command

Here we analyze how our implementation of LU decomposition compares against MATLAB's internal method. To compare, we will call our Ludecomp script and time it using the tic-toc functions of matlab. We will then find the solution by forcing matlab to use its LU function. This will allow us to compare our results. First we need to figure out how to force Matlab to call its LU solver instead of the banded solver. To do this, we will use the function we create at the end of this file. Given this function is implemented, we can vary the grid refinement from 50 to 2500 in increments of 50 to get a decent idea of the time trends. Our results follow.

```
clear;clc; close all;

%1D timing
alpha = 2;
nset = 100:5:1000;
times = zeros(length(nset),2);
for i = 1:length(nset)
    N = nset(i);
    [A,b,~] = Heat1D(alpha,N,0,1,0,0);
    tic
    [~] = Ludecomp(A,b,N+1,10e-6,zeros(N+1,1));
    times(i,1) = toc;

    [~,times(i,2)] = forceLU(A,b);
end
matrixSizes = nset;
logSize = log10(matrixSizes);
logTimesMe = transpose(log10(times(:,1)));
logTimesMat = transpose(log10(times(:,2)));

p1 = polyfit(logSize,logTimesMe,1);
p2 = polyfit(logSize,logTimesMat,1);

fitTimeMe = (matrixSizes.^p1(1))*10.^(p1(2));
fitTimeMat = (matrixSizes.^p2(1))*10.^(p2(2));

figure;
loglog(nset,times(:,1),'r-');
hold on;
loglog(matrixSizes,fitTimeMe,'k-');
xlabel('Matrix Size ((N+1)');
ylabel('Time (s)');
title('Personal LU Timing for 1D BVP');
a = annotation('textbox',[.15 .58 .2 .2],'String',"y = "+num2str(p1(1))+"x +
"+num2str(p1(2)),'FitBoxToText','on');
legend('Measured','Linear Fit','Location','northwest');
exportgraphics(gcf,"PersonalLU1D.png");
```

```
figure;
loglog(nset,times(:,2),'r-');
hold on;
loglog(matrixSizes,fitTimeMat,'k-');
xlabel('Matrix Size (N+1)');
ylabel('Time (s)');
title('Matlab LU Timing for 1D BVP');
a = annotation('textbox',[.15 .58 .2 .2],'String',"y = "+num2str(p2(1))+"x +
"+num2str(p2(2)),'FitBoxToText','on');
legend('Measured','Linear Fit','Location','northwest');
exportgraphics(gcf,"MatlabLU1D.png");

%2D timing
nset = 5:1:50;
times = zeros(length(nset),2);
for i = 1:length(nset)
    N = nset(i);
    [A,b,~] = Heat2D(alpha,N,0,1,0,1);
    tic
    [~] = Ludecomp(A,b,(N+1).^2,10e-6,zeros((N+1).^2,1));
    times(i,1) = toc;

    [~,times(i,2)] = forceLU(A,b);
end
matrixSizes = (nset + 1).^2;
logSize = log10(matrixSizes);
logTimesMe = transpose(log10(times(:,1)));
logTimesMat = transpose(log10(times(:,2)));

p1 = polyfit(logSize,logTimesMe,1);
p2 = polyfit(logSize,logTimesMat,1);

fitTimeMe = (matrixSizes.^p1(1))*10.^(p1(2));
fitTimeMat = (matrixSizes.^p2(1))*10.^(p2(2));

figure;
loglog(matrixSizes,times(:,1),'r.-');
hold on;
loglog(matrixSizes,fitTimeMe,'k-');
xlabel('Matrix Size ((N+1)^{2})');
ylabel('Time (s)');
title('Personal LU Timing for 2D BVP');
a = annotation('textbox',[.15 .58 .2 .2],'String',"y = "+num2str(p1(1))+"x +
"+num2str(p1(2)),'FitBoxToText','on');
legend('Measured','Linear Fit','Location','northwest');
exportgraphics(gcf,"PersonalLU2D.png");
figure;
loglog(matrixSizes,times(:,2),'r.-');
hold on;
loglog(matrixSizes,fitTimeMat,'k-');
xlabel('Matrix Size ((N+1)^{2})');
ylabel('Time (s)');
title('Matlab Intrinsic LU Timing for 2D BVP');
a = annotation('textbox',[.15 .58 .2 .2],'String',"y = "+num2str(p2(1))+"x +
"+num2str(p2(2)),'FitBoxToText','on');
legend('Measured','Linear Fit','Location','northwest');
exportgraphics(gcf,"MatlabLU2D.png");
```

Problem 5 - Error

```
clear; clc; close all;
% Vary Grid Size, alpha = 2
```

```
%nset = [5,6,10,12,15,20,24,30,40,60];
nset = [2,3,5,6,10,12,20,24,30,40];
nref = 120;
alpha = 2;

[A,b,x,y] = Heat2D(alpha,nref,0,1,0,1);
Tref = A\b;
Tref2D = zeros(nref+1);

for l = 1:length(Tref)
    i = mod(l-1,nref+1)+1;
    j = (l-i)/(nref+1)+1;

    Tref2D(i,j)=Tref(l);
end

normalizedError1 = zeros(length(nset),1);
for i = 1:length(nset)
    [A,b,x,y] = Heat2D(alpha,nset(i),0,1,0,1);
    Tres = Ludecomp(A,b,(nset(i)+1)^2,10e-6,zeros((nset(i)+1)^2,1));
    Tres2D = zeros(nset(i)+1);
    for l = 1:length(Tres)
        k = mod(l-1,nset(i)+1)+1;
        j = (l-k)/(nset(i)+1)+1;

        Tres2D(k,j)=Tres(l);
    end
     errorArray = zeros(nset(i)+1);
     scaler = nref/nset(i);
     sum = 0;
     for j = 1:(nset(i)+1)
         for k = 1:(nset(i)+1)
             errorArray(j,k) = (Tres2D(j,k) - Tref2D(1+scaler*(j-1),1+scaler*(k-
1))).^2;
             sum = sum + errorArray(j,k);
         end
     end
     sum = sqrt(sum);
     normalizedError1(i) = sum/((nset(i)+1));
end

logSize = log10(nset);
logError = log10(normalizedError1);

p1 = polyfit(logSize,logError,1);
fitLine =  (nset.^p1(1))*10.^(p1(2));

figure;
loglog(nset,normalizedError1,'b.-');
hold on;
plot(nset,fitLine,'k-');
title("Fit Line Error - \alpha = 2");
xlabel("Log Grid Size (log(n))");
ylabel("Log Average Error");
a = annotation('textbox',[.15 .12 .2 .2],'String',"y = "+num2str(p1(1)) ...
    +"x + "+num2str(p1(2)),'FitBoxToText','on');
legend('Measured','Linear Fit','Location','southwest');
exportgraphics(gcf,"ErrorFitGraph.png");

figure;
plot(nset,normalizedError1,'b.-');
title("Error Trend for \alpha = 2");
xlabel("Grid Size (n)");
```

```
ylabel("Average Error");
exportgraphics(gcf,"ErrorGraph.png");
% Vary Alpha Error
alphaSet = [2,5,10,25,50];
figure;
hold on;
for i = 1:length(alphaSet)
    alpha = alphaSet(i);
    normalizedError = zeros(length(nset),1);
    [A,b,x,y] = Heat2D(alpha,nref,0,1,0,1);
    Tref = A\b;
    Tref2D = zeros(nref+1);

    for l = 1:length(Tref)
        k = mod(l-1,nref+1)+1;
        j = (l-k)/(nref+1)+1;

        Tref2D(k,j)=Tref(l);
    end

    for j = 1:length(nset)
        [A,b,~] = Heat2D(alpha,nset(j),0,1,0,1);
        Tres = Ludecomp(A,b,(nset(j)+1)^2,10e-6,zeros((nset(j)+1)^2,1));

        Tres2D = zeros(nset(j)+1);
        for l = 1:length(Tres)
            k = mod(l-1,nset(j)+1)+1;
            m = (l-k)/(nset(j)+1)+1;

            Tres2D(k,m)=Tres(l);
        end
        errorArray = zeros(size(Tres2D));
        scaler = nref/nset(j);
        sum = 0;
        for k = 1:(nset(j)+1)
            for m = 1:(nset(j)+1)
                errorArray(k,m) = Tres2D(k,m) - ...
                    Tref2D(1+scaler*(k-1),1+scaler*(m-1));
                errorArray(k,m) = errorArray(k,m).^2;
                sum = sum + errorArray(k,m);
            end
        end
        sum = sqrt(sum);
        normalizedError(j,1) = sum/(nset(j)+1);
        disp(num2str(nset(j))+"donef");
    end
    plot(nset,normalizedError,'.-','DisplayName',"$\alpha$ = "+num2str(alpha));
    title('Error vs Grid Size for Various \alpha');
    xlabel('Grid Size (n)');
    ylabel('Average Error');
end

legend('Interpreter','latex');

exportgraphics(gcf,"AlphaError.png");
```

Helper Functions

```
function [res,time] = forceLU(A,b)
    tic
    [L,U,P] = lu(A);
    y = L\(P*b);
```

```
    res = U\y;
    time = toc;
end
```

Ludecomp Function

This function is the primary working function that we will make use of. Although it does very little in itself, it calls all the necessary subroutines that make LU decomposition work in our pseudocode implementation.

```
function [x] = Ludecomp(A,b,n,tol,x)
    o = zeros(n,1);
    s = zeros(n,1);
    er = 0;
    [A,er,o] = Decompose(A,n,tol,o,s,er);
    if(not (er==-1))
        [~,x] = Substitute(A,o,n,b,x);
    end
end
```

Decompose

This function does the actual Doolittle decomposition algorithm, calling on pivot when necessary.

```
function [A,er,o] = Decompose(A,n,tol,o,s,er)
    for i = 1:n
        o(i)=i;
        s(i)=abs(A(i,1));
        for j = 2:n
            if(abs(A(i,j))>s(i))
                s(i)=abs(A(i,j));
            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)
            er = -1;
            disp(A(o(k),k)/s(o(k)));
            break;
        end

        for i =  (k+1):n
            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);
            end
        end
    end
    if(abs(A(o(k),k)/s(o(k)))<tol)
        er = -1;
        disp(A(o(k),k)/s(o(k)));
    end

end
```

Pivot

This function implements the concept of pivoting in LU decomposition.

```
function o = Pivot(a,o,s,n,k)
    p = k;
    big = abs(a(o(k),k)/s(o(k)));
    for ii = (k+1):n
        dummy = abs(a(o(ii),k)/s(o(ii)));
        if(dummy>big)
            big = dummy;
            p = ii;
        end
    end
    dummy = o(p);
    o(p)=o(k);
    o(k)=dummy;
end
```

Substitute

This function does the forward and backward substituion to finish of the Ludecomp function.

```
function [b,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
```

Heat1D

This function provides us with our heat equation matrix, b vector, and grid point locations.

```
function [A,b,x] = Heat1D(alpha,n,xmin,xmax,Txmin,Txmax)
    b = zeros(n+1,1)-alpha;
    h = (xmax - xmin)/n;
    b(1) = Txmin/h^2;
    b(end) = Txmax/h^2;
    x=xmin:h:xmax;

    A = zeros(n+1);
    for i = 1:(n+1)
        if(i==1 || i == n+1)
            A(i,i) = 1;
        else
            A(i,i-1) = 1;
            A(i,i) = -2;
            A(i,i+1) = 1;
        end
    end
    A = A./h^2;
```

```
end
```

Heat 2D

This function provides us with our matrix and b vector for a two dimensional heat diffusion problem along with the x and y positions of all the grid points. We assume a boundary condition of T=0 everywhere.

```
function [A,b,x,y] = Heat2D(alpha,n,xmin,xmax,ymin,ymax)
    A = zeros(n*n);
    b = zeros(n*n,1);

    hx = (xmax-xmin)/n;
    hy = (ymax-ymin)/n;

    x = xmin:hx:xmax;
    y = ymin:hy:ymax;

    [x,y] = meshgrid(x,y);

    for i = 1:(n+1)
        for j = 1:(n+1)
            l = (j-1)*(n+1)+i;
            if(i== 1 || j == 1 || i == n+1 || j == n+1)
                A(l,l) = 1;
                b(l) = 0;
            else
                A(l,l-1) = 1/hx^2;
                A(l,l) = -2/hx^2;
                A(l,l+1) = 1/hx^2;
                A(l,l-n-1) = 1/hy^2;
                A(l,l) = A(l,l) - 2/hy^2;
                A(l,l+n+1) = 1/hy^2;

                b(l)=-alpha;
            end
        end
    end
end
```