Celine Ho

MECE 5397

9 May 2018

Project A – Ahc2-3

**Abstract**

The purpose of this project is to write a code to solve the given two-dimensional Helmholtz equation while taking into account the given boundary conditions. The project implements both mathematical knowledge and coding knowledge from the course.

For the mathematical portion, the project demonstrates my understanding of the given two-dimensional Helmholtz equation and its corresponding boundary conditions. It features the step-by-step process of solving the equation and setting up the matrices. The project also shows detailed descriptions on why each step is performed. Two iterative methods are featured to solve linear equation systems. These methods include Gauss-Seidel and Successive Over Relaxation (SOR). Both methods are explained in detail, with the pros and cons listed for using each method. The mathematical model of this project contains too many iterations and large amounts of data that cannot be solved by hand, but can be solved in Matlab, That is where the coding portion of this project comes in.

For the coding section, I will use the mathematical model that I develop to write a Matlab code for this project. In addition to the project demonstrating my understanding of how to write codes in Matlab, it will also demonstrate how well I can implement coding practices such as checkpointing, managing changes with Git version control, and coding for visualization.

The report will begin by providing information about the Helmholtz equation to develop a mathematical model for solving the equation, specifically to develop matrix A. The next section will employ the boundary conditions to create matrix F and to define boundary values for matrix u. The next part will describe the Gauss-Seidel and SOR methods. The next section will describe the technical specifications of the computer used for this project. The last section will look at the results graphs and provide comments on the results.

**Mathematical Statement of Problem and Discretized Version of Equations**

Given two-dimensional Helmholtz Equation

Domain of interest is the rectangle

* ,

Boundary conditions:

* ,

Steps for solving:

* Substituted the following values into the given 2D Helmholtz Equation.
* Write the equation above in matrix form. Arrange ­ in single vector u.
  + Fix j and run over all i’s in column 1, Fix j+1 and run over all i’s in column 2, Fix j+2 and run over all i’s in column 3, etc.
* To save space, write the transpose of u () instead of u
* To find the coefficients of ‘u’ terms in the matrix above, look at the following graph that has a five-point stencil cross.



* From the graph above, the penta-diagonal matrix (matrix with five non-zero diagonals) is formed and will be labeled matrix A.



* From matrix A,
  + = middle point and is applied to the residual at the same point.
  + = increment in ‘i’ direction (going from left to right); decrement in ‘i’ direction (going from top to bottom of matrix)
  + = value “n” points away
  + = zero terms: shows absence of some terms because we are working with a five-point stencil (center stencil point with 4 neighboring points to form cross shape)
* Plug in into matrix A to get the below matrix



* Write equation similar to matrix above with each row written horizontally from each other. For the coefficients from matrix A, it is assumed that
  + The above values are written in matrix form A where
* The system can be rewritten in the form which is expanded below.
* The values in matrix F are given from the boundary condition
  + , where
  + More details on finding Matrix F and developing Matrix F in Matlab will be given in the next section, “Description of Numerical Method”.
* As shown above, we know that
  + 🡸 This can be rearranged to solve for [u]
  + where
  + 🡸 We will plot this in Matlab to get our solution.
* We will use the given boundary conditions for Matrix u to populate specific points in [u] before running the Matlab code. More details about how to do this are in the next section “Description of Numerical Method”.

**Description of Numerical Method and Pseudocode Description**

1. Pseudocode and description for creating the pentadiagonal matrix A that will be used is shown below.

N = 11; %set number of nodes

dx = 1/N; %distance between each node in x-space

onesmatrix = ones (N-2,1); %9x1 matrix of ones

A\_diag = eye(N-1) \* (-4 / dx^2) %initial value (-4) for diag entries of diag block

A\_diag = A\_diag+ diag(onesmatrix,1) / dx^2; %values for 1st diagonal of diag block

A\_diag = A\_diag+ diag (onesmatrix,-1) / dx^2; %values for diag below main diagonal

A\_offdiag = eye (N-1) / dx^2; %identity matrix for off-diag blocks

%Put blocks in place in big matrix

A = sparse ((N-1) \* (N-1) , (N-1) \* (N-1));%Sparse 100x100 matrix

%no need to calculate for zeros

range=(i-1)\*(N-1);

range2=(i-2)\*(N-1);

for i = 1:N-1

A(range+1 : range+(N-1) , range+1 : range+(N-1)) = A\_diag;

%range of rows&columns corresponding to ith block

end

for i = 2:N-1 %Starts at 2. 1 less off diag block

%fill upper diag blocks

A(range2+1 : range2+(N-1) , range+1 : range+(N-1)) = A\_offdiag;

%%fill lower diag blocks (switch i&j entries)

A(range+1 : range+(N-1) , range2+1 : range2+(N-1)) = A\_offdiag;

end

* To develop Matrix F, j terms in adjacent rows are fixed while all i’s are run over as the rows go down. Next, j+1 is fixed while and all i’s are ran over in the next group of adjacent rows. So on and so forth. This is shown below.
* If I want matrix F to be of the size 100x100, that means that there will need 100 ‘i’ and ‘j’ terms (ixj=100x100). N=number of ‘i’ and ‘j’ terms=100).
* There will be 100 nodes between the boundaries = and =. The distance between each node is . The value of each x, y between and is below.
* Each value of x and y can be plugged into the given boundary condition
  + We can plug in and into which simplifies to
  + The analysis of matrix F above will be used to develop matrix F in Matlab.

1. The pseudocode for the known values of matrix F is shown below

F = zeros(100); %Set f to be zero everywhere to test it

N=100; %number of i and j terms

%F is filled up with %F(1,1),F(2,1),..F(100,1),F(1,2),F(2,2,),..F(100,2),..,F(99,100),F(100,100)

for j = 1:N

y = -pi + 2 / (N-1) \* (j - 1) \* pi; %how ‘j’ corresponds to ‘y’ location

for i = 1:N

k = i + (j - 1) \* 100; % F(k)=F(i,j) to populate F where k=i+(j-1)\*10

x = -pi + 2 / (N-1) \* (i - 1) \* pi; %how ‘i’ corresponds to ‘x’ location

F(k) =sin(pi\*(x+pi) / (2\*pi)) \* cos( pi/2 \* (2\*(y + pi) / (2\*pi) + 1));

end

end

* We will have to apply the boundary conditions for ‘u’ to populate specific spots in Matrix u before running the code.
  + [u]=100x1 matrix with unknown values (known values at boundary conditions)

[u] =

* From the boundary condition (BC), I know that
  + BC 1:
  + BC 2: , =
  + BC 3:
    - Found from
  + BC 4:
    - Found from =
  + BC 5:
  + BC 6:
    - Found from
    - Plug in values for and

From the boundary conditions above, I am trying to set every ‘u’ value where i=1, i=10, and j=1 (which corresponds to and ), to have the boundary conditions applied.

A similar example of what I am looking for is let’s say I want to perform operation

=

However, I want to apply the following boundary condition 🡺 If , then

I want to write a code where my final answer is

1. Pseudocode for boundary conditions of [u] listed above is shown below

%Applying Boundary Conditions

for j = 1

x = -pi + 2 / (N-1) \* (i - 1) \* pi; %how ‘i’ corresponds to ‘x’ location

u(i,j)=-pi\*(2\*pi)^2+(x+pi)\*((-2\*pi)+(2\*pi^2)); %From BC 6

for i = 1

y = -pi + 2 / (N-1) \* (j - 1) \* pi; %how ‘j’ relates to ‘y’ location

u(i,j)=y\*(pi-y)^2; %From BC 3

for i=10

y = -pi + 2 / (N-1) \* (j - 1) \* pi;

u(i,j)=(pi-y)^2\*cos(y); %From BC 4

end

end

end

m=100; %introduce non-existent grid point (m) for Neumann B.C.

for j = 10

y = -pi + 2 / (m-1) \* (j-1) \* pi; %how ‘j’ relates to ‘y’ location

syms y(u)

dy=diff (y(u) , u); %From BC 5

dy=0;

end

The entire Matlab code of the solution consists of the following:

%(1)Pseudocode and description for creating pentadiagonal matrix A found earlier in this section

%(2)Pseudocode for the known values of matrix F found earlier in this section

%(3)Pseudocode for boundary conditions of [u] found earlier in this section

%(4) To create 3D surface plot of final solution, below pseudocode is used

Lambda=1/2;

u=-(A+lambda) \ F; %minus sign to move F onto right-hand side

surf(u)

Applying Gauss-Seidel method gives

Applying Successive Over Relaxation (SOR) gives

The Jacobi method is shown below, and it is similar to the Gauss Seidel method which is used in this project.

.

The Jacobi method finds the values of to the kth iteration and does not change until the complete (k+1)th iteration has been calculated.

However, for the Gauss-Seidel method, the new values for are used immediately as they are known. Because of this, the Gauss-Seidel method converges and diverges at a faster rate than the Jacobi method. Since the Jacobi method finds values for each ‘u’ value at the same time, when each iteration of the Jacobi method shrinks the error by ½, each iteration of the Gauss-Seidel method shrinks the error by ¼. More Jacobi Method iterations are required to approximate the same solution as the Gauss-Seidel Method.

The SOR method is similar to the Gauss-Seidel method except it converges at a faster rate. For the SOR method, is the extrapolation factor, and the correctly chosen will accelerate the rate of convergence of the iterations which leads to the solution. If , the SOR method becomes the Gauss-Seidel method. From the description, it may look the SOR method is the preferred method, but it has a drawback because cannot be computed in advance. An estimate for must be tried and modified to find what value is most effective.

**Technical Specifications of Computer Used**

I used the computers in the UH Engineering Computing Center. The technical specifications of the computer were found by accessing the Linux System at UH. The command that I used to find each specification is written in parenthesis next to each technical specification.

Number of sockets: 4 (lscpu)

CPU model name: Intel (R) Xeon(R) CPU E5620 @ 2.40 GHz (less /proc/cpuinfo)

Number of cores/CPU: 1 (less /proc/cpu info)

Current CPU Clock frequency: 2394MHz (lscpu)

Max CPU clock frequency: 2394 MHz (cat /proc/cpuinfo | grep MHz)

L1d cache size:32kB (lscpu)

L1i cache size: 32kB (lscpu)

L2 cache size:256kB (lscpu)

L3 cache size: 12288KB (lscpu)

Total DRAM per CPU: 16335680 kB (cat /proc/meminfo)

PCI Channels version: VMware PCI Express Root Port rev 01 (lspci –vv)

Operating system version: Red Hat Enterprise Linux Server release 6.6 (cat /etc/issue.net)

C/Fortran compiler name and version: gcc version 4.4.7 20120313 (gcc –v)

• Properties of CPU (from Intel website)

* thermal design power (TDP): 80W
* <https://ark.intel.com/products/47925/Intel-Xeon-Processor-E5620-12M-Cache-2_40-GHz-5_86-GTs-Intel-QPI>

**Results**

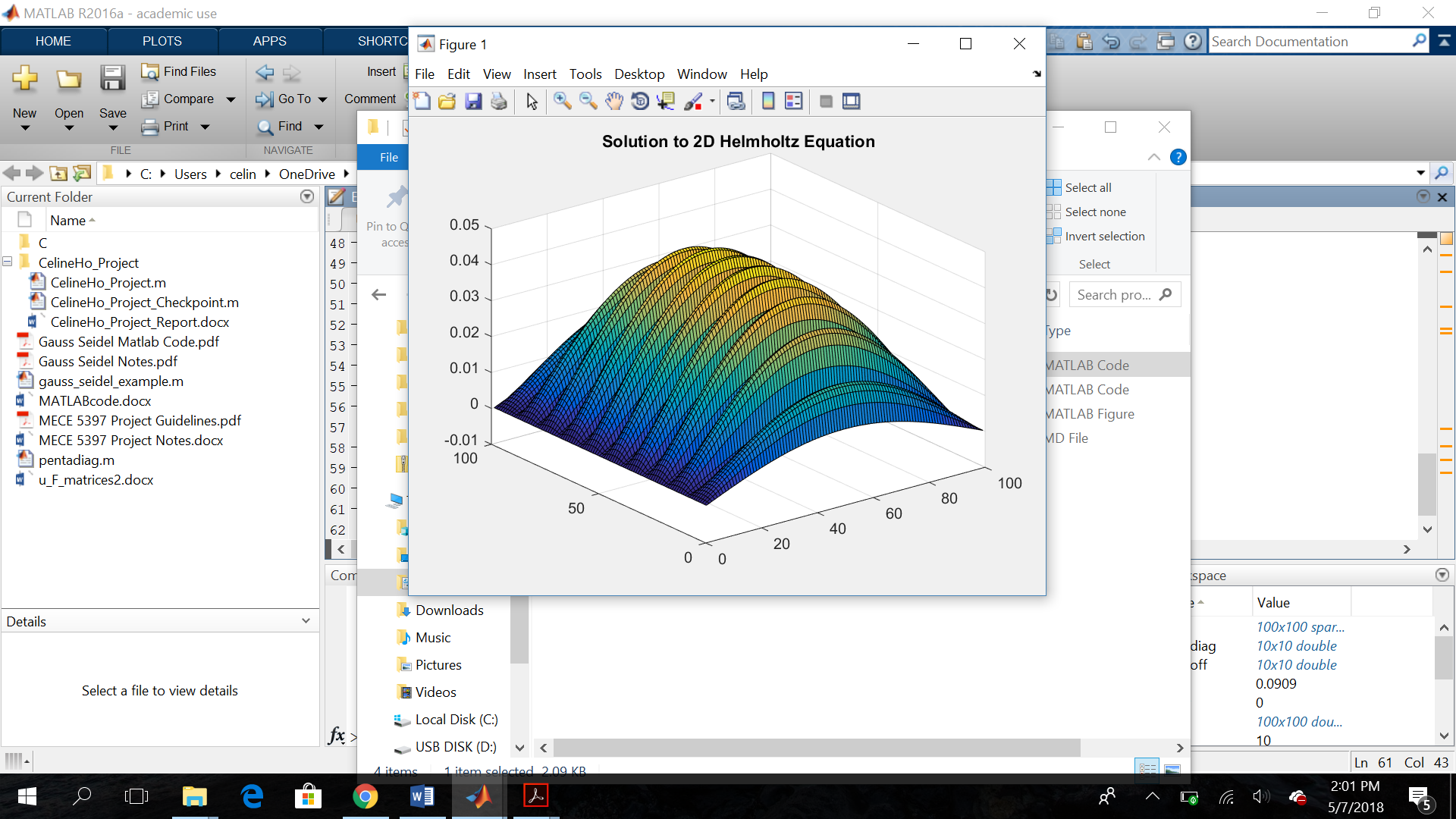


Figure 1: The parameters for the above simulation were the given boundary conditions from the original problem and . To shorten the run time of the code, N was set to equal a smaller value (N=11) for matrix A, which mean that i=10 and j=10. Therefore, the size of Matrix A is 100x100. N was set to equal 100 for Matrix F, which causes the size of Matrix F to also be 100x100.

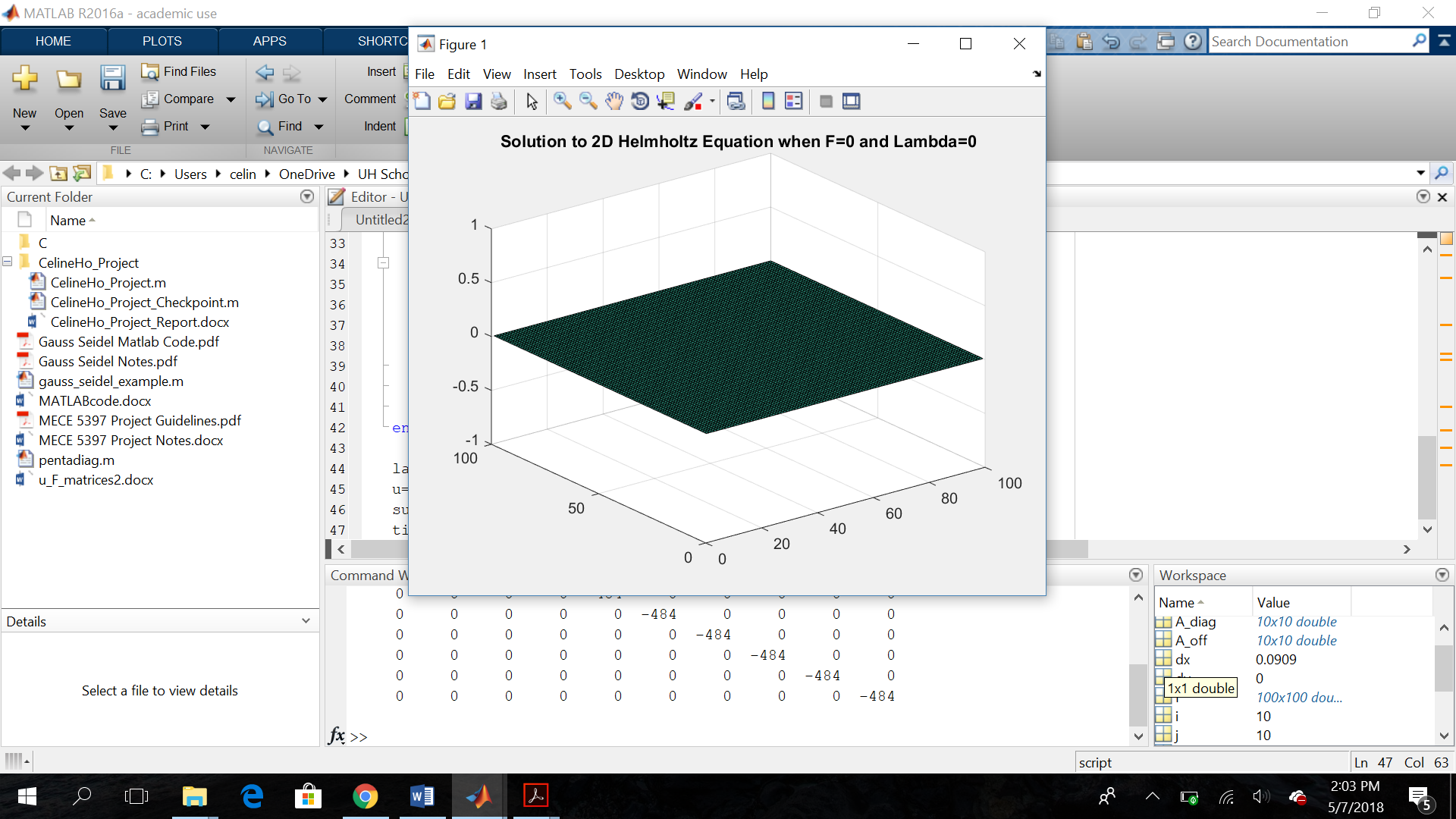


Figure 2: The parameters for the simulation above include the boundary conditions from the given problem, excluding the boundary conditions for F(x,y). Instead, F is set to be zero by replacing the code for F with F=zeros(100)so that F is a 100x100 matrix filled with only zero terms. in this simulation as well.

If more points are used for discretization, each cell in the grid becomes smaller, which increases the number of cells and also increases the accuracy of the solution. While using more points for discretization can increase the accuracy and decrease the error, a larger number of points used for discretization will cause the code to run longer. Also, there is an upper limit to the tolerance for generating a grid for the problem, where running the code will converge to the solution. However, when that upper limit is reached for the cell size in the grid, increasing the number of points for discretization no longer causes the solution to become more accurate, and may even move the output farther away from the solution.

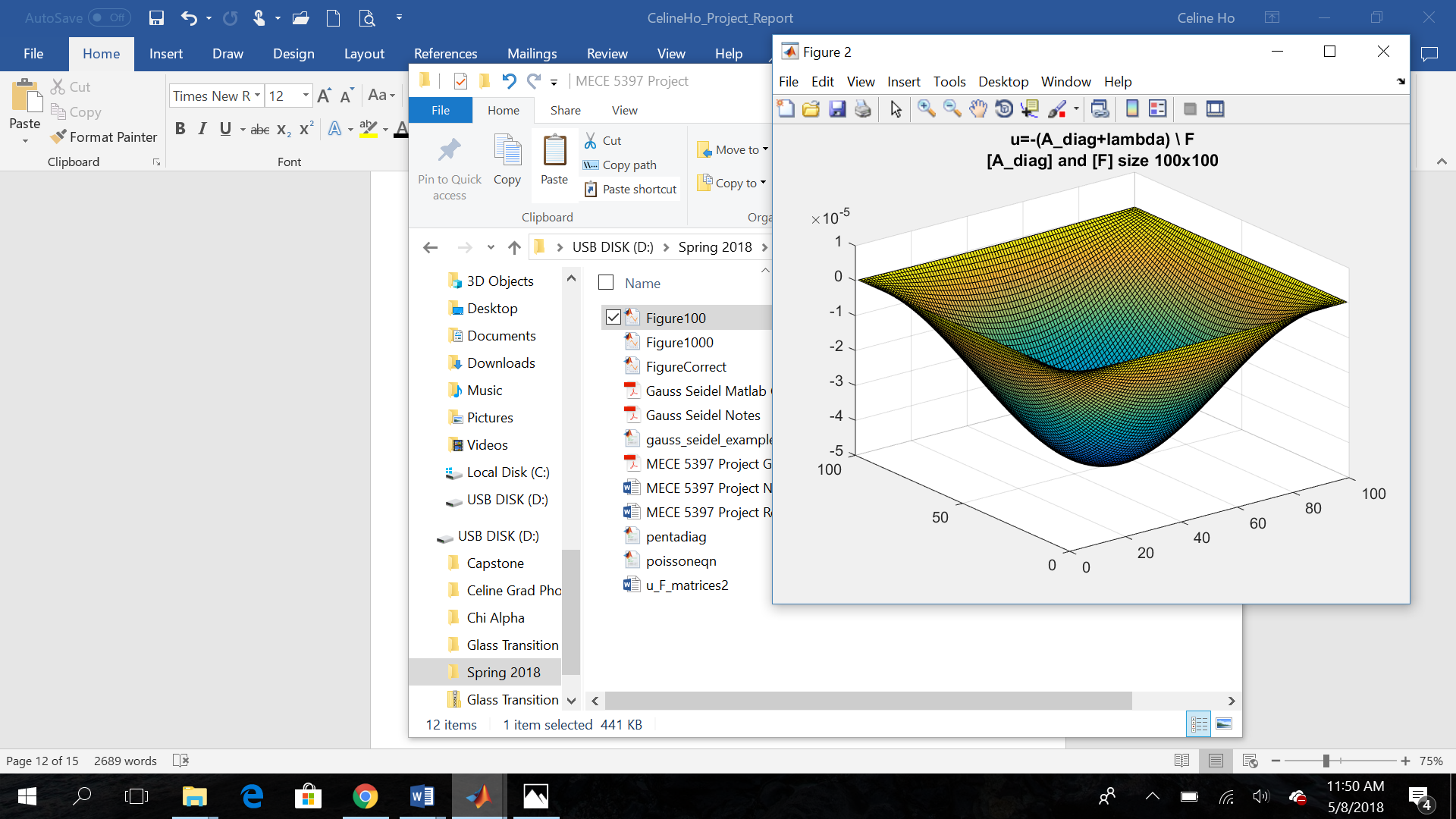


Figure 3: The grid convergence study above shows what happens when the tolerance upper limit for generating the grid in this problem is reached. I ran the code, but instead of running u=-(A+lambda) \ F, I ran u=-(A\_diag+lambda) \ F for the sake of observation. For the Figure 3 simulation, N=101 for matrix A and N=100 for matrix F, so that [A\_diag] and [F] will be the size of 100x100.

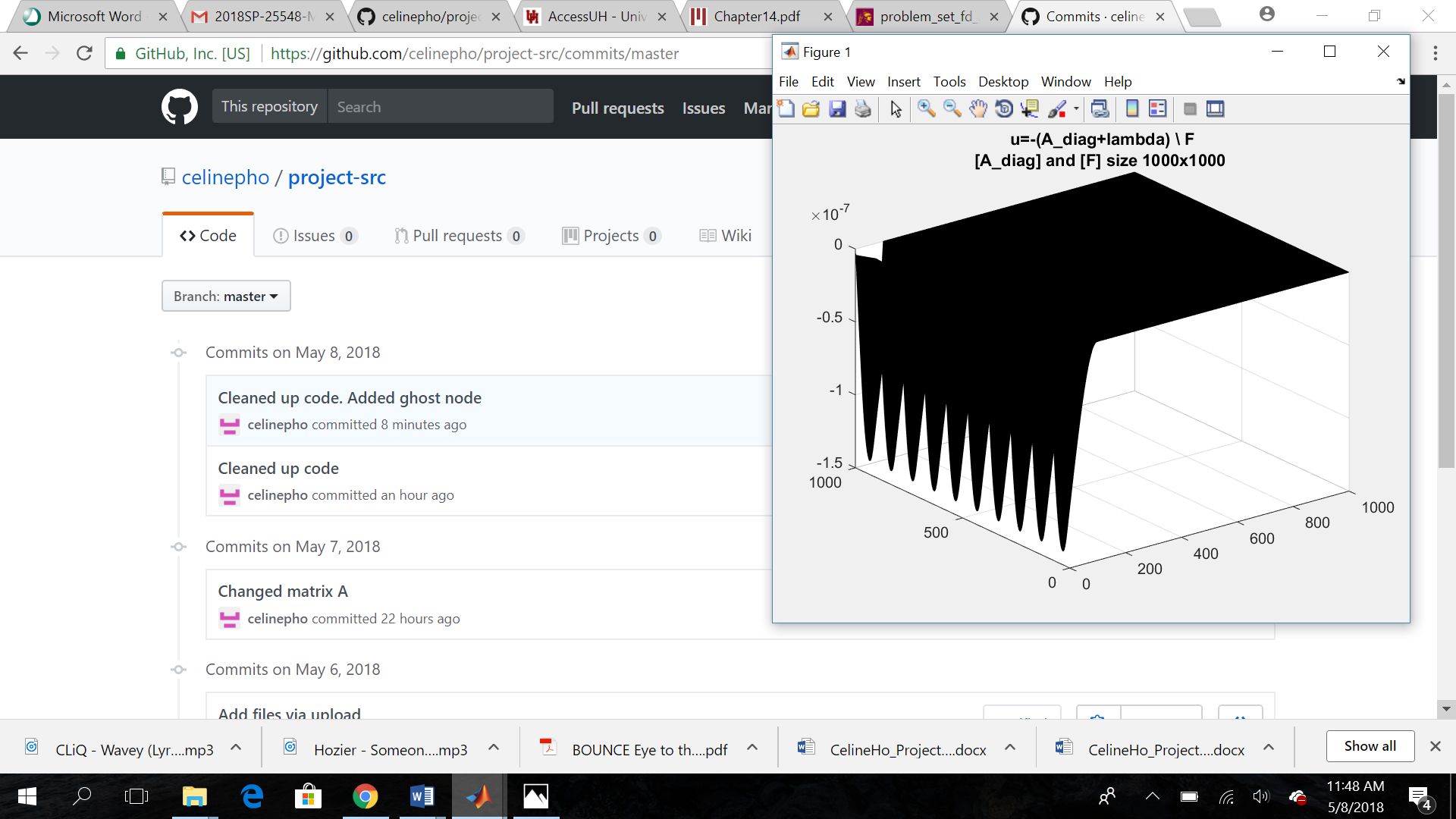


Figure 4: For the next simulation shown above, I set N=1001 for matrix A while setting N=1000 for matrix F. Therefore, [A\_diag] and [F] will be of the size 1000x1000.

As observed above, setting matrices ‘A\_diag’ and ‘F’ to be the size 1000x1000 is too large. The upper limit has already been reached, so increasing the number of points and cells does not produce the correct simulation.

The CFL condition expresses that information from a given cell or element within the mesh of the solution must only spread to its side-by-side neighbor. The information cannot jump to a cell that is not immediately next to it if it is to meet stability conditions (Caminha, 2018).

* From the “Mathematical Statement” section of this problem, it is shown that the following values are substituted into the given 2D Helmholtz Equation
* and are the length between each node in different spatial directions. From the Gauss-Seidel method, the above equation is shown to be
* After a Taylor series expansion is performed on the Gauss-Seidel scheme above, the diffusion coefficients for numerical diffusion are introduced.
  + Where Courant number and
  + in order for numerical viscosity to be positive.

From the “Mathematical Statement” section of this report, the 2D Helmholtz equation is discretized to this function.

* From the discretized function, the formal order of spatial accuracy is the scheme is second order in x-space and second order in y-space. This is because the leading terms of the truncation error include the factors and .

The next step is to calculate the observed order of accuracy, which may not match the formal order of accuracy due to errors in the code. To do this, the discretization error should be found by calculating the difference between the exact solution and the solution from the code (“Numerical Errors,” 2017).

* Discretized Error =
* =coefficient of leading error term; =observed order of accuracy

The expected theoretical behavior of the graph was a smooth wave, similar to the simulation examples shown below.

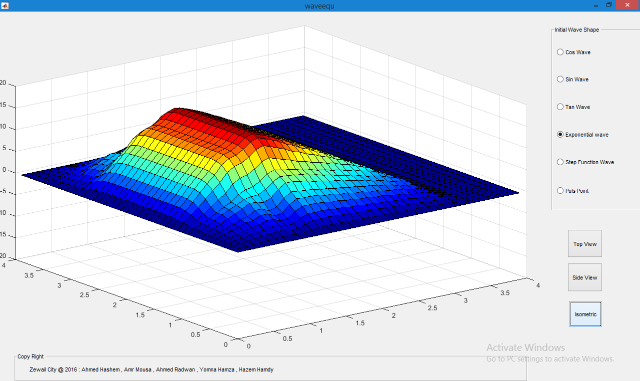
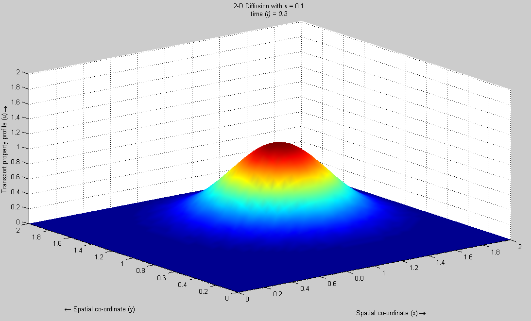


Figure 5 on left: [www.mathworks.com/matlabcentral/mlc-downloads/downloads/submissions/38088/versions/1/screenshot.png](http://www.mathworks.com/matlabcentral/mlc-downloads/downloads/submissions/38088/versions/1/screenshot.png)

Figure 6 on right: [www.mathworks.com/matlabcentral/mlc-downloads/downloads/submissions/55117/versions/3/screenshot.png](http://www.mathworks.com/matlabcentral/mlc-downloads/downloads/submissions/55117/versions/3/screenshot.png)

The solution that was found was not one continuous wave, but rather, a group of multiple smaller waves plotted next to each other as seen in Figure 1.

**References**

Caminha, G (2018, March 13). The CFL Condition and How to Choose Your Timestep Size.

Retrieved from <http://www.simscale.com/blog/2017/08/cfl-condition/>

Numerical Errors. (2017, July 19). Retrieve from

<https://www.sharcnet.ca/Software/Ansys/17.2/en-us/help/cfx_ref/i1001058.html>