Abstract

This course is an independent study on Numerical Solution of Partial Differential Equations by the Finite Element Method. This semester I have continued working with Vinh Nguyen. We are advised by Dr. Sigal Gottlieb. Our main focus this semester was to use finite element methods to map curves in space. MatLab and Octave were used throughout the semester. In order to map these functions in space. In order to start, we needed an understanding of how math can be used to relate piecewise linear functions to our actual functions.

1 Main Code

The following is the main code produced by Vinh Nguyen. I have gone through to depict what is happening throughout and also have commented throughout this code and the preceding codes. First I will present the code and then give a short description of each. For a prequel, the following will be the basis for our understanding.

Here we have just created a variable M which will be used to determine the matrix size. Later we will see how changing M will change errors produced between our real function $-\sin(\pi x)$ and our piecewise linear functions made from phi.m. Keep in mind that this code is for piecewise linear phi functions. From the above code we can see that function y is plotted and is shown in Figure 1.
To continue our analysis, we're working with equation (1.6), page 21 from the Numerical Solution of Partial Differential Equations. For simplicity, I will cut this into a matrix and the new equation that can be seen in Figure 1, for $M = 4$:

$$
\begin{bmatrix}
2 & -1 & 0 & 0 \\
-1 & 2 & -1 & 0 \\
0 & -1 & 2 & -1 \\
0 & 0 & -1 & 2 \\
\end{bmatrix}
\begin{bmatrix}
\mathcal{N}_1 \\
\mathcal{N}_2 \\
\mathcal{N}_3 \\
\mathcal{N}_4 \\
\end{bmatrix}
= 
\begin{bmatrix}
b_1 \\
b_2 \\
b_3 \\
b_4 \\
\end{bmatrix}
$$

This exact matrix was imported into LaTeX with the following code located in Section 2, Imported Matrix. This code allows us to change the size of $M$ which will change the size of the Matrix we produce. Proceeding this code is the updated code. This code is now entered as a function and can be changed if needed without affecting the original main code itself.

## 2 Import Matrix

```matlab
A=zeros(M,M);  \textit{\%\% Creates a Matrix of zeros with size MxM}
```
B = ones(M,M); % Creates a Matrix of ones with size MxM
C = 2*B; % Creates a Matrix of all 2s with size MxM
D = -1*ones(M-1,M-1); % Creates a Matrix of -1s with size (M-1)x(M-1)

for i = 1:M % Steps through i = 1 up to M
    A(i,i) = C(i,i); % Sets matrix A a diagonal of all 2s.
    for j = 1:M-1 % Steps through j = 1 up to M-1
        A(j,j+1) = D(j,j); % Sets the element term A(j,j+1) equal to D(j,j)
        A(j+1,j) = D(j,j); % Sets the element term (j+1,j) equal to D(j,j) which is -1
    end
end

A = (1/h)*A; % (1/h) is our coefficient in front of matrix

3 Call Functions

3.1 Revised Code Function phi

The following function phi will be used to produce a matrix of ones and zeros. Once we integrate we will see how the final matrix is created.

function [phi]=function_phi(x,M) % Call function with parameters
    % x is input variable, M is matrix size
    for j = 2:M-1 % Creates a for loop from 2 --> M-1
        for i = 1:M % Creates a for loop from 1 --> M
            if (x(i) >= x(j-1)) & (x(i) <= x(j)) % Creates a domain X_j-1 --> X_j
                phi(j,i)=(x(i)-x(j-1))/(x(j)-x(j-1)); % phi function
            elseif (x(i) >= x(j)) & (x(i) <= x(j+1)) % Creates a domain X_j --> X_j+1
                phi(j,i)=(x(j+1)-x(i))/(x(j+1)-x(j)); % phi function
            else
                phi(j,i)=0; % If value falls out of for loop
            end
        end
    end

phi(1,i)=0; % Set to phi function to 0
phi(1,1)=1;
phi(M,M)=1; % Boundary Conditions
The above coding, Function Phi, allows for us to create piecewise linear functions. These piecewise linear functions will later be used to help map a line in space. We are working with a 2-Dimensional space so our function will closely map our original \( \sin \) function. To reiterate, the following conditions should be stated and a short recap given.

### 3.2 Initial Conditions, Recap

\[
M=4; \quad \text{Matrix size}
\]
\[
y = -u'' = \sin(\pi x) \quad \text{Given Function}
\]
\[
x = \text{linspace}(0,1,M);
\]
\[
h = x(1,2) - x(1,1); \quad \text{Domain}
\]
\[
\text{phi} = \text{function}\_\text{phi}(x,M);
\]
\[
\text{MatrixCoefficient}\_\text{(change in x)} \quad \text{Creates piecewise linear Function phi}
\]

### 3.3 Creating the Slope

Now we need to build our derivatives. The derivative of a line is the slope of the line. If a line is vertical we will have an infinite slope. If a line is horizontal, we have a slope of 0. For our study we only encounter horizontal lines, therefore constituting our derivatives to be zero. Once we have stored our values for the number of indicated iterations, we can then begin to structure our theory.

\[
\text{function} \ [\text{delphi}] = \text{delphi}(x,M,h)
\]
\[
\text{for } j=2:M-1
\]
\[
\quad \text{for } i=1:M
\]
\[
\quad \quad \text{if } (x(i) >= x(j-1)) \& (x(i) <= x(j)) \quad \text{checks if there are intersecting lines}
\]
\[
\quad \quad \text{delphi}(j,i) = 1/(x(j)-x(j-1)); \quad \text{if the statement is true takes the slope}
\]
\[
\quad \quad \text{of line for } x_{(j-1)} \text{ to } x_j
\]

---

%%

% The above coding, Function Phi, allows for us to create piecewise linear functions. These piecewise linear functions will later be used to help map a line in space. We are working with a 2-Dimensional space so our function will closely map our original \( \sin \) function. To reiterate, the following conditions should be stated and a short recap given.

### 3.2 Initial Conditions, Recap

%%

% % % Initial Conditions % % % Recap

\[
M=4; \quad \text{Matrix size}
\]
\[
y = -u'' = \sin(\pi x) \quad \text{Given Function}
\]
\[
x = \text{linspace}(0,1,M);
\]
\[
h = x(1,2) - x(1,1); \quad \text{Domain}
\]
\[
\text{phi} = \text{function}\_\text{phi}(x,M);
\]
\[
\text{MatrixCoefficient}\_\text{(change in x)} \quad \text{Creates piecewise linear Function phi}
\]

%%

% % % Matrix size % % %

% % % Given Function % % %

% % % Domain % % %

% % % MatrixCoefficient\_\text{(change in x)} % % %

% % % Creates piecewise linear Function phi % % %

### 3.3 Creating the Slope

Now we need to build our derivatives. The derivative of a line is the slope of the line. If a line is vertical we will have an infinite slope. If a line is horizontal, we have a slope of 0. For our study we only encounter horizontal lines, therefore constituting our derivatives to be zero. Once we have stored our values for the number of indicated iterations, we can then begin to structure our theory.

% % % Creating the Derivative % % %

% % % This is the call function % % %

% % % variable x, constant M, h % % %

% % % Creates a for loop 2 \( \rightarrow \) M % % %

% % % Creates another for loop 1\( \rightarrow \) M % % %

% % % checks if there are intersecting lines % % %

% % % if the statement is true takes the slope % % %

% % % of line for \( x_{(j-1)} \) to \( x_j \) % % %
elseif (x(i) >= x(j)) & (x(i) <= x(j+1))
    delphi(j,i)=-1/(x(j+1)-x(j));
    elseif (x(i) > x(j) & x(i) < x(j+1)) % Checks if there are intersecting lines
        delphi(j,i)=-1/(x(j+1)-x(j)); % if the statement is true, takes the slope
        for x-j to x_{j+1}
        if no slope then zero
    end
end
delphi(1,1)=-1/h; % Sets delphi(1,1) equal to -1/h
delphi(M,M)=1/h; % Sets delphi(M,M) equal to 1/h

We are taking the slope of linear piecewise functions. We also know that the maximum height can only be one. The coefficient of h is defined as x_j - x_{j-1}. Now in order to create the matrix, we need to take the integral of delphi times delphi. As seen from the matrix equation, we need a way to solve for the coefficient b_n. The only way to find b_n is to set up the following code which will create our first matrix. This code takes the integral of phi(i)*phi(j).

3.4 Integrated Phi

In order to take the integral of our phi function, a program called Maple was used to find our integrals. The following integrals are complexed and can be seen below.

```
function [intphi]=int_phi_f(x,M)
    intphi(1)=0; % Initial condition always 0
    for j=2:M-1
        intphi1(j)= ((pi*(x(j)-x(j-1))*cos(pi*x(j))) + sin(pi*x(j-1))-sin(pi*x(j)))/(pi^2*(x(j-1)-x(j))); % First Integral for X_{j-1} to X_j
        intphi2(j)= ((pi*(x(j)-x(j+1))*cos(pi*x(j)) -sin(pi*x(j))+sin(pi*x(j+1)))/(pi^2*(x(j)-x(j+1)))); % Second Integral for X_j to X_{j+1}
        intphi(j) = intphi1(j) + intphi2(j) % Adding both intphi1 + intphi2 will give our total integral
    end
    intphi(M)=0; % Sets our last integral to 0
```
4 Solving Coefficient Vector $b^n$

Now that we have solved for the first two parts of the proceeding matrix, we still have to solve for our coefficient variable vector $b^n$). Once we have solved, we can constructed our final test function and produce errors. An error of zero means we have completely mapped our test function.

$$\begin{bmatrix}
2 & -1 & 0 & 0 \\
-1 & 2 & -1 & 0 \\
0 & -1 & 2 & -1 \\
0 & 0 & -1 & 2
\end{bmatrix} \begin{bmatrix}
\mathbb{R}_1 \\
\mathbb{R}_2 \\
\mathbb{R}_3 \\
\mathbb{R}_4
\end{bmatrix} = \begin{bmatrix}
b_1 \\
b_2 \\
b_3 \\
b_4
\end{bmatrix}$$

% Computing $b_n$

```matlab
a=phi^1*int_phi; % Creates a variable for phi^1*int_phi
b=zeros(M,M); % We are solving for b_n
for i=1:M % Creating a matrix of all zeros
    for j=1:M % For loop from 1 to M-1
        b(j,i)=a(i)*phi(j,i); % Second for loop allows to step through matrix and compare previous nodes
    end % with preceding nodes
end % Strings b_n together
for i=1:M
    u_h(i)=-1*b(i,i); % completes our test function
end % by multiplying by -1.
```

Now that we have found our coefficient $b_n$, we can now graph our test function and see how close we are to our original function. Below in Figure 2, is the test function itself. From this graph we cannot tell if our test function closely maps our original function until we graph both of them together. This mapping can be seen in Figure 3. For $M=20$, we see how the test function (red) closely maps our original function (blue). The problem is that there is a
small gap between the two. The graphs were produced for $M = 20$. As we increase the size of $M$ we will notice that our errors decrease.

### 4.1 Computing the Error

```matlab
%% Computing Error

```% sin(pi*x))/pi^2; plot(x,u); hold on; plot(x,u_h,'r') err=abs(u_h-u); e1=norm(err,1)/M; e2=norm(err,2)/sqrt(M)

```% Actual Error Readings

```

<table>
<thead>
<tr>
<th>$M$</th>
<th>$e_1$</th>
<th>$e_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>0.0167</td>
<td>0.0167</td>
</tr>
<tr>
<td>25</td>
<td>0.0132</td>
<td>0.0132</td>
</tr>
<tr>
<td>35</td>
<td>0.0093</td>
<td>0.0093</td>
</tr>
<tr>
<td>50</td>
<td>0.0065</td>
<td>0.0065</td>
</tr>
<tr>
<td>100</td>
<td>0.0032</td>
<td>0.0032</td>
</tr>
<tr>
<td>200</td>
<td>0.00016</td>
<td>0.00016</td>
</tr>
</tbody>
</table>
Figure 2: Test Function $0 < x < 1$

Figure 3: Test with Original Function $0 < x < 1$
We can see from the produced error reading data that if we increase the number of iterations we can eventually get very close to zero. The error readings can be seen above for matrix size M equal to: 20, 25, 35, 50, 100, and 200. As we double the number of M, we cut our error by half. As we step through each max number of iterations, the errors start to get closer and closer to zero. As stated previously, if we set this code to run for an infinite number of iterations, we could see our test function match the original function almost perfectly.

5 The Theory

5.1 General integration

\[ u''(x) + f(x) = 0 \]

Let's say...

\[ u(0) = u(1) = 0 \]

\[ u''(x) = 0 \]

When the second derivative is equal to 0, the function is said to be linear.

\[ u(0) = b = 0 \]

\[ u(1) = a = 0 \]

With this substitution...

\[ -u''(x) = f(x) \]

\[ -\int_0^x u''(\varepsilon)d\varepsilon + c_1 = \int_0^x f(\varepsilon)d\varepsilon \]

\[ -\int_0^x \int_0^\zeta u''(\varepsilon)d\varepsilon d\zeta + c_1 x + c_2 = int_0^x f(\varepsilon)d\zeta \]

5.2 Least Squares Method

If we integrate the following:

\[ \int (f(x) - g(x))^2 \]

This should give you the best error reading between the actual function and the theoretical function. Now if we solve for a solution \( u \)...

we say \( u \) is in the space of function \( V \)

\[ u \in V \]

\[ F(u) \leq F(V) \]

This true for...

\[ u \in V \]
5.3 Partial Differential Equations

\[ u_t + u_x = 0 \]

\[ |u_t^h - u_x^h| \leq |V_t^h - V_x^h| \]

5.4 Definitions

Inner Product

\[ (v, w) = \int V(x)w(x)dx \]

Integral of their product over their domain

V is a space of functions V(x) which are continuous and bounded by:

\[ V(0) = V(1) = 0 \]

V' is a piecewise continuous function. It has a finite number of jumps in the space of V having a number of polynomials.

F(u) is my residual function.

\[ u'' + f = 0 \]

\[ u(0) = u(1) = 0 \]

for \( u \in V \), \( v \in V \) and \( w \in V \) we have...

\[ F(u) \leq F(v) \]

We now have:

\[ (v, w) = \int_0^1 v(x)w(x)dx \]

Continuous and bounded by:

\[ v(0) = v(1) = 0 \]

v' is a piecewise and continuous and also has a finite number of jumps.

F(v) residual

\[ F(v) = 1/2(v', v') - (f, v) \]

\[ = \frac{1}{2} \int_0^1 v'v' \, dx - \int_0^1 fv \, dx \]

\[ = \frac{1}{2} [v'v]_0^1 - \int_0^1 v''vdx - \int_0^1 fv \, dx \]

\[ = \frac{1}{2} [v'v]_0^1 - \int_0^1 v''vdx - \int_0^1 fv \, dx \]

10
\[
\int_0^1 (v''v + fv)dx
\]

\[
= \frac{1}{2} [\int_0^1 (v''v + fv)dx]
\]

\[
= -\frac{1}{2} [\int_0^1 v(v'' + f)dx]
\]

for ..

\[
u'' + f = 0
\]

\[
u(0) = u(1) = 0
\]

We find what \(v \in V\) is.

\[\int_0^1 (vu'' + fv)dx = 0\]

We find what \(v \in V\) is.

\[\int_0^1 (vu'' + fv)dx \leq \int_0^1 (wu'' + fw)dx\]

This is done for a \(w \in V\)

### 5.5 Finite Element Analysis

Finite Element Analysis (FEA) is a computer animated model used to design and run test creatively and cost effectively. In our case, FEA is used to analyze the errors produced between an original function and a finite element test function. We are using FEA to meet specifications provided by Dr. Sigal Gottlieb. FEA can be used to determine the design modifications required to meet our new condition. In general there are two types of analysis used for FEA; 2-D and 3-D modeling. 2-D modeling tends to be less accurate than 3-D modeling but for our purpose we will used 2-D modeling. It should be noted that there is no limit to the number of dimensions, as long as there is a computer that can handle such complexed problems, a solution can be found.