Peter Cottle SID 19264824

ME 280A – 9/13/2011

Homework Assignment #2

Introduction:

Since Homework #1 dealt with the overall procedure of the Finite Element Method, this homework instead deals with the analysis of error convergence as a function of the order of the basis elements chosen. The differences between linear elements, quadratic elements, and cubic elements will be discussed as well as their various advantages and disadvantages in terms of computational intensity and memory storage. These considerations are very important when constructing enterprise-level finite element codes that must balance faster error convergence with larger memory overhead and CPU time. For the majority of the course we will be using linear elements for our FEM code, but this homework serves as a cursory (yet useful) examination of the effects of the basis element order.

Objectives:

The objectives of this homework build upon those from Homework #1; instead of simply constructing an FEM solver, we will instead alter our code to perform with any order of basis elements (among other minor differences). Thus, the objectives of this homework are:

- Modify our code to deal with non-zero Dirichlet boundary conditions. This means altering the RHS when eliminating rows and columns from the stiffness matrix. This is something I personally had to do after an error in Homework #1.
- Re-solve for the exact solution with these new boundary conditions.
- Modify the error calculation code to compute the defined error. This is something I had to personally do, for my error calculation code from Homework 1 was based on area.
- Modify the stiffness matrix code to handle 2nd and 3rd order basis functions.
- Modify the Right-Hand-Side calculation code to handle other basis functions.
- Perform rigorous error and performance analysis with this new code.

Procedure:

This homework was mainly split into two parts; the first was doing all of the code modification to obtain an FEM solver that works with different order basis functions, and the second was performing the error behavior analysis.

Modifying my previous code to work with Dirichlet boundary conditions and better error calculation was quite straightforward; the more difficult aspects of the code modifications are detailed below:

Calculating the RHS with 2nd and 3rd order basis functions:

- Define phi's over the zeta domain
    - For 2nd order basis functions, these were derived in class and thus easy to hard-code in. For 3rd order basis functions, I chose to use the polyfit command against various points of unity (and 0 elsewhere) for each basis function. This ensures that they behave in the desired way and prevents typing errors. This can also be extended to be used for any order basis function.
- Integrate these phi's against the loading function*
    - This involves transforming the loading function to the zeta domain. An x to zeta transformation function can be obtained by the relationship between x and zeta (detailed in the code and in the class notes). After the x coordinates are transformed, one must also multiply by the Jacobian to complete the transformation. Integration then can be done with Gaussian quadrature or trapz.
- Assemble the R matrix correctly
    - With 2nd or 3rd order basis functions, care must be taken to ensure the correct nodes overlap while the intermediary nodes remain independent.

*Note: I included two copies of the RHS calculation code; one that performs in the zeta domain and one that performs in the x domain. I only was able to get the x domain code to converge to a solution, but both integrals are theoretically and numerically identical; the only difference is that the x-domain integral cannot use Gaussian quadrature.

Calculating the Stiffness matrix with 2nd and 3rd order basis functions:

- Define phi primes over the zeta domain
    - This involves defining three or four phi primes instead of just two. These calculations are performed in the zeta domain with **Gaussian integration**; I used 8-point integration because the order of the integral can become quite large.
- Assemble the K matrix correctly. This involves ensuring the correct places in the stiffness matrix overlap.

The error analysis procedure was mainly taking the time to plot error as a function of N elements or element size h for any order P. Determining the relationship between the error and the element size for every order basis function was done both with curve-fitting and qualitative analysis.

Findings

Deriving the weak form

Because my first homework referred to the notes for the weak form derivation, I will instead detail that process here. First, we begin with our general form of the ODE that we wish to solve:

$$\frac{d}{dx}(A_1(x)\frac{du}{dx}) + A_2(x)u - A_3(x) = 0$$

Next, choose v which is any test function that ensures the integral below remains finite. Any v that satisfies this constraint is an 'admissible' function. Integrate the product of the above equation with v.

$$\int_\Omega [\frac{d}{dx}(A_1(x)\frac{du}{dx}) + A_2(x)u - A_3(x)]vdx = 0$$

Realizing that v is not unique (that there are many admissible functions), specify that this above equation holds for "all v's that are admissible."

Next, distribute the v inside the integral.

$$\int_\Omega [\frac{d}{dx}(A_1(x)\frac{du}{dx})v + A_2(x)uv - vA_3(x)]dx = 0$$

Split up the integral and re-arrange.

$$\int_\Omega \frac{d}{dx}(A_1(x)\frac{du}{dx})vdx + \int_\Omega A_2(x)uvdx = \int_\Omega vA_3(x)dx$$

Then, we examine the chain rule 'sleight of hand' that represents the term on the far left. This term can be thought of as a result of performing the chain rule on another differentiation product and taking one side of the results. The formal definition of the trick:

$$\frac{d}{dx}[(A_1(x)\frac{du}{dx})v] = [\frac{d}{dx}(A_1(x)\frac{du}{dx})]v + A_1(x)\frac{du}{dx}\frac{dv}{dx}$$

Solving for the term in our integral:

$$\frac{d}{dx}[(A_1(x)\frac{du}{dx})v] - A_1(x)\frac{du}{dx}\frac{dv}{dx} = [\frac{d}{dx}(A_1(x)\frac{du}{dx})]v$$

Substituting this term into our earlier integral:

$$\int_\Omega A_1(x)\frac{du}{dx}\frac{dv}{dx}dx - \int_\Omega A_2(x)uvdx = -\int_\Omega A_3(x)vdx + A_1(x)\frac{du}{dx}v|_{u(0)}^{u(L)}$$

We now have the weak form of the form:

$$B(u, v) = L(v)$$

Next, approximate u(x) as a series of basis functions. Do the same for v. Thus, we have approximations of the form:

$$u(x) \approx \sum_{i=1}^{N} \phi_i a_i \qquad v(x) \approx \sum_{j=1}^{N} \phi_j b_j$$

Substituting this in gives us the general equation form of:

$$\int_\Omega A_1(x) \sum_{i=1}^{N} \phi_i a_i \sum_{j=1}^{N} \phi_j b_j dx + terms = -\int_\Omega A_3(x) \sum_{j=1}^{N} \phi_j b_j dx$$

Which is where we get the general "Stiffness matrix * A = Loading vector" form from that so many undergrads are familiar with. This is the approximated form of the weak form where we get our matrix equations from. Hopefully this serves as an adequate summary of the derivation of the weak form that homework #1 did not include.

Re-solving with new Boundary Conditions:

We can pick up from the end of HW#1 after the double integration to solve for the new constant. Please refer to HW#1 for steps leading up to this equation.

$$u(x) = -\frac{5L^2}{\pi^2} cos(\frac{\pi k x}{L}) - \frac{25L^2(1-k^2)}{4\pi^2 k^2} sin(\frac{2\pi k x}{L}) + C_1 x + C_2$$

The first boundary condition remains the same:

$$u(0) = -\frac{5L^2}{\pi^2} cos(0) - \frac{25L^2(1-k^2)}{4\pi^2 k^2} sin(0) + C_2$$

$$u(0) = -\frac{5L^2}{\pi^2} - 0 + C_2 = 0$$

$$C_2 = \frac{5L^2}{\pi^2}$$

The second boundary condition calculation becomes:

$$u(L) = -\frac{5L^2}{\pi^2} cos(\frac{\pi k L}{L}) - \frac{25L^2(1-k^2)}{4\pi^2 k^2} sin(\frac{2\pi k L}{L}) + \frac{5L^2}{\pi^2} + C_1 L$$

$$u(L) = -\frac{5L^2}{\pi^2}(-1)^k - 0 + \frac{5L^2}{\pi^2} + C_1 L = 1$$

$$C_1 = \frac{5L}{\pi^2}(-1)^k - \frac{5L}{\pi^2} + \frac{1}{L}$$

Meaning the exact solution now becomes:

$$u(x) = -\frac{5L^2}{\pi^2}cos(\frac{\pi k x}{L}) - \frac{25L^2(1-k^2)}{4\pi^2 k^2}sin(\frac{2\pi k x}{L}) + \frac{5L^2}{\pi^2} + (\frac{5L}{\pi^2}[-1]^k - \frac{5L}{\pi^2} + \frac{1}{L})x$$

Modifying FEM code to work with quadratic and cubic basis functions

The process of modifying my code to work with different order basis functions was a time-intensive process (as well as something that occupied much of the GSI's personal time, my apologies). I eventually obtained code that satisfies these requirements; the full printout of code can be viewed in the appendix section.

Solving the given boundary value problem to achieve Error < 0.01

After running my code against the given boundary value problem, these are the number of finite elements I needed in order to achieve error < 0.01:

Table 1: Number of elements N needed for error < 0.01 versus order

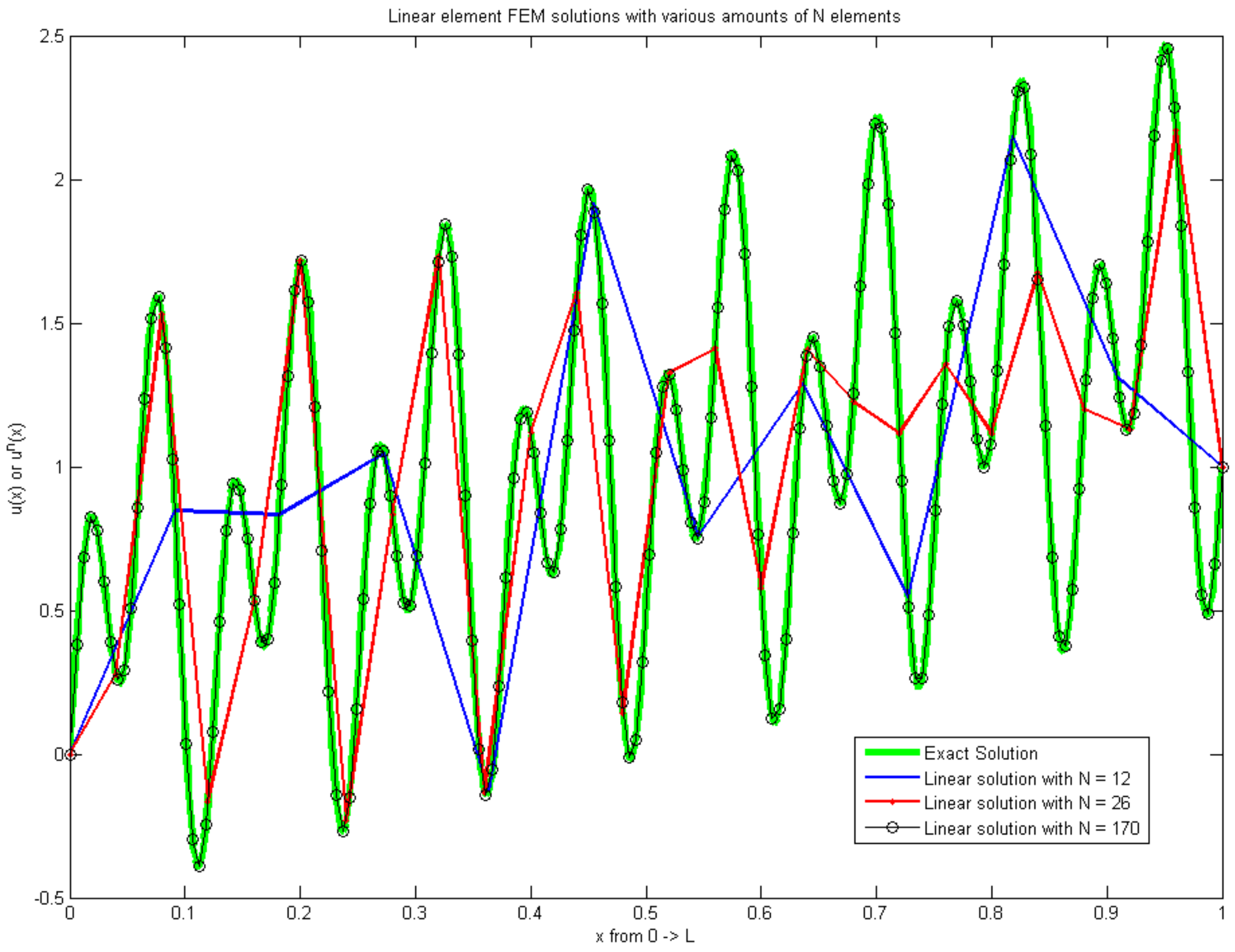
Order of the basis functions - P	N - elements needed for error < 0.01
1	~3001*
2	202
3	67

*Note: For linear elements, I used large mesh steps at the end (usually adding around 500 elements each iteration), so this number is an approximation.

Plotting the numerical solution for several values of N for each P (along with exact):

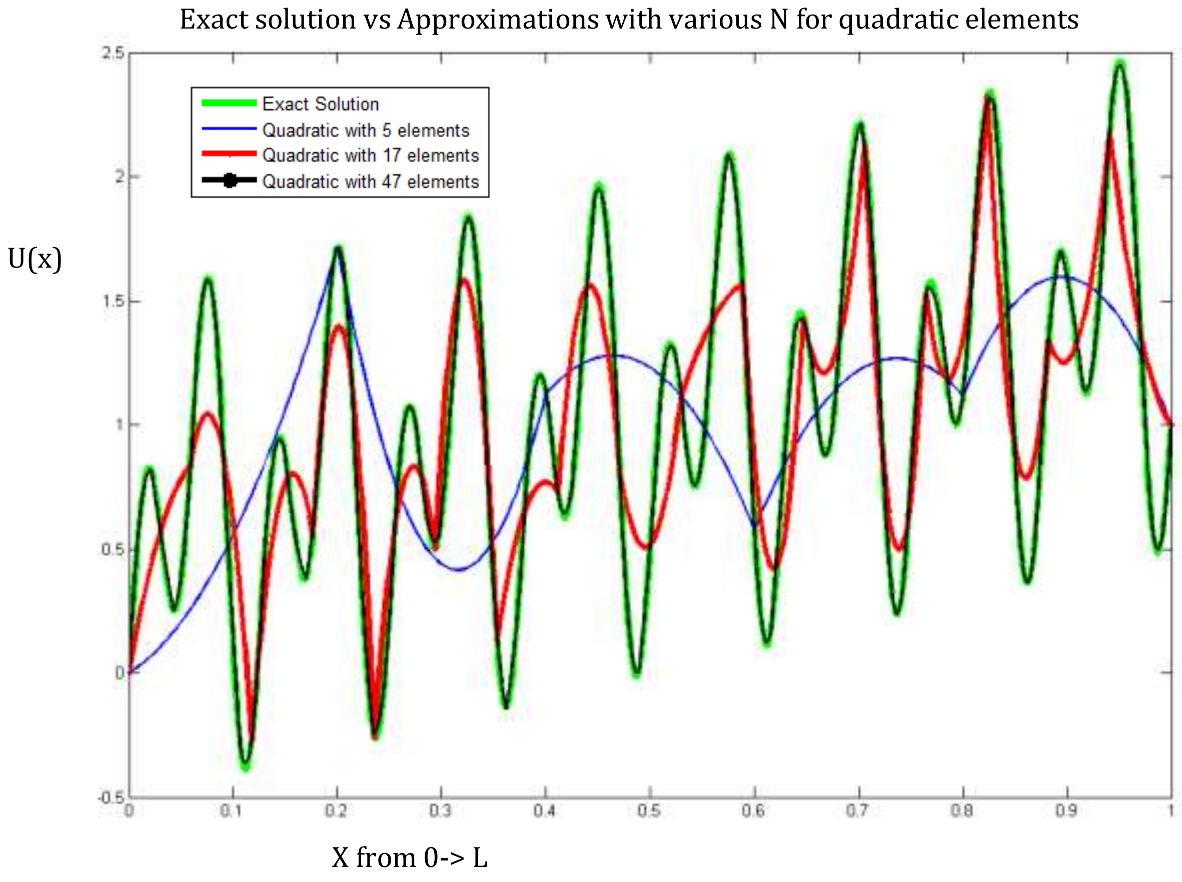
Linear:

Figure 1: Linear FEM solutions with various amounts of N elements



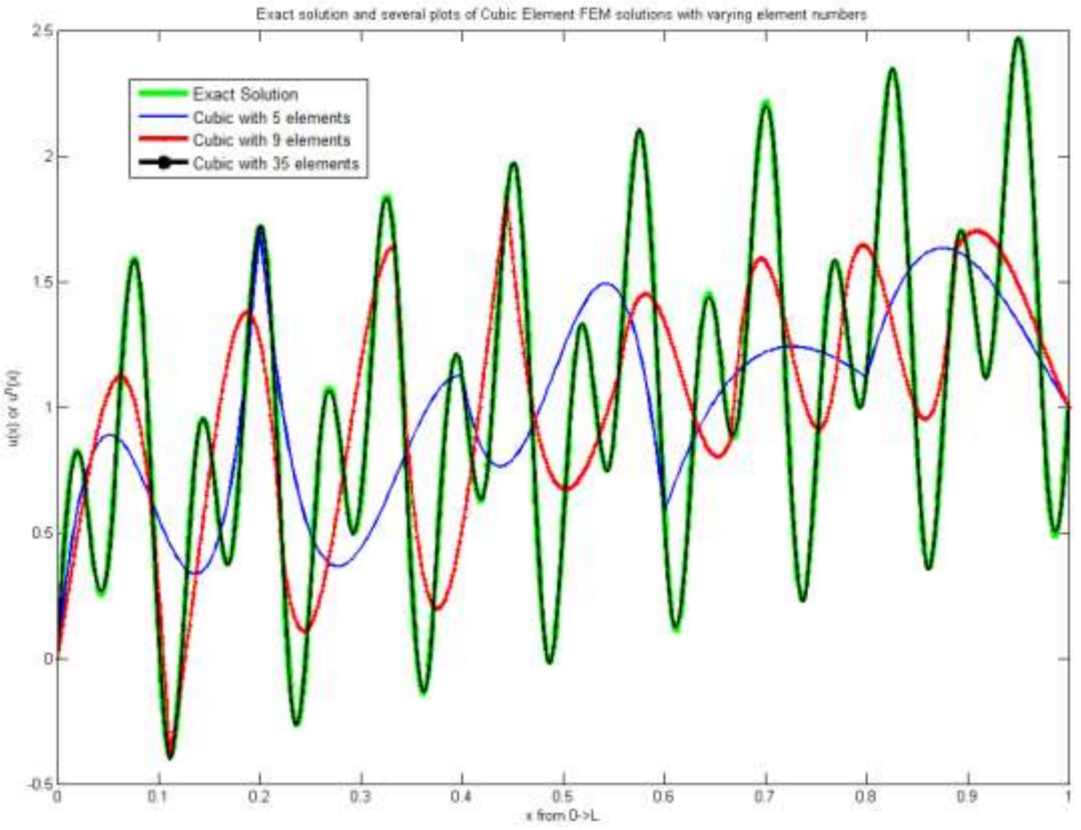
Quadratic:

Figure 2: Quadratic FEM solutions with various amounts of N elements



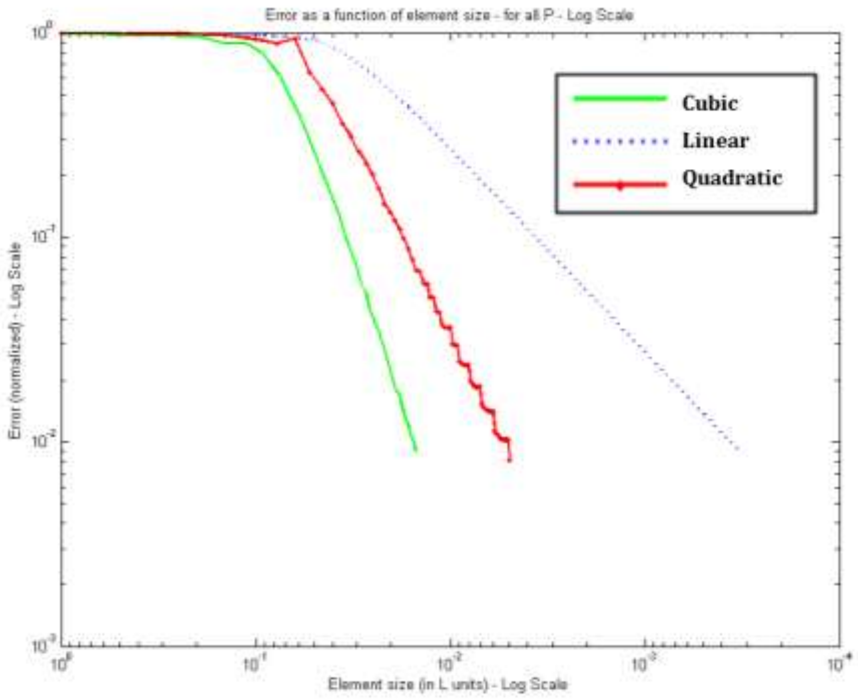
CUBIC:

Figure 3: Cubic FEM solutions with various amounts of N elements



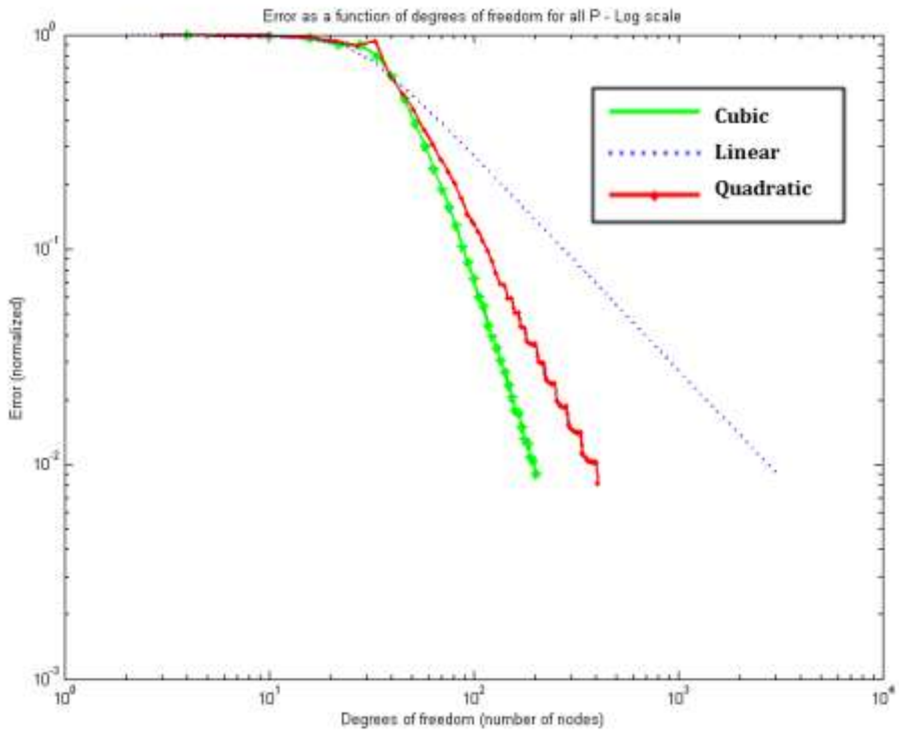
Plotting error as a function of element size h for each P (Note, I plotted on a loglog scale, which shows the behavior as a function of p better than linear scale):

Figure 4: Error as function of element size (to be fitted later)



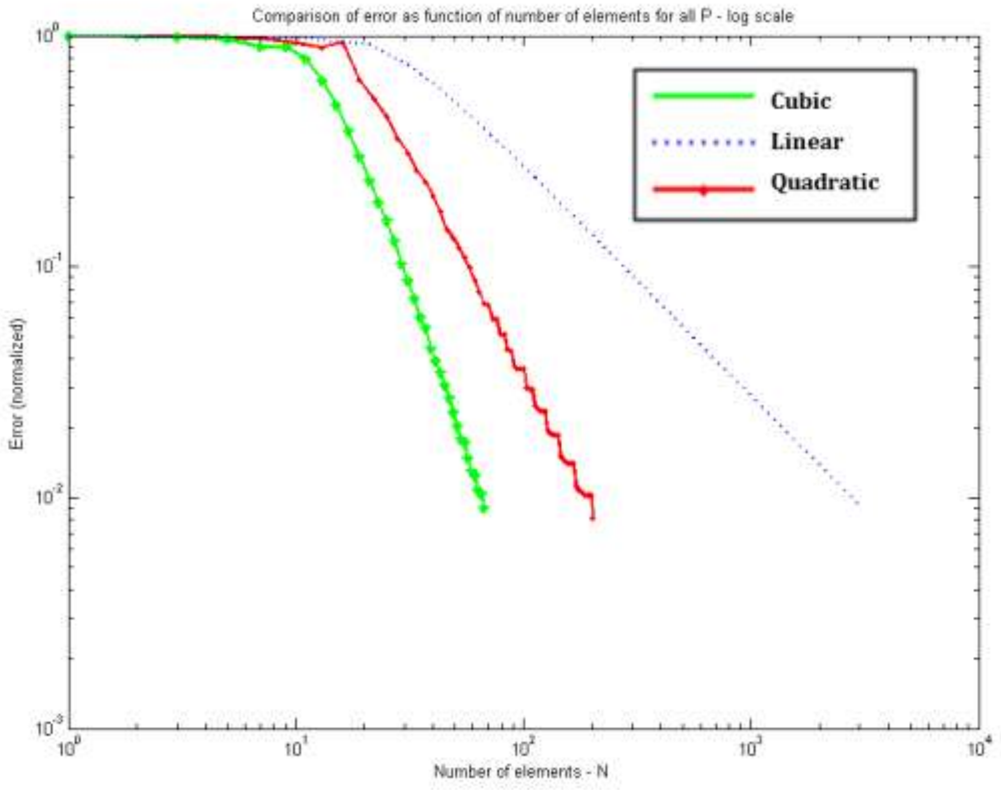
Plotting Error as a function of the number of degrees of freedom for each p:

Figure 5: Error as function of degrees of freedom



Plotting Error as a function of the number of elements for each p (this plot is similar but not equal to the function of element size plot and also not required):

Figure 6: Error as function of number of elements

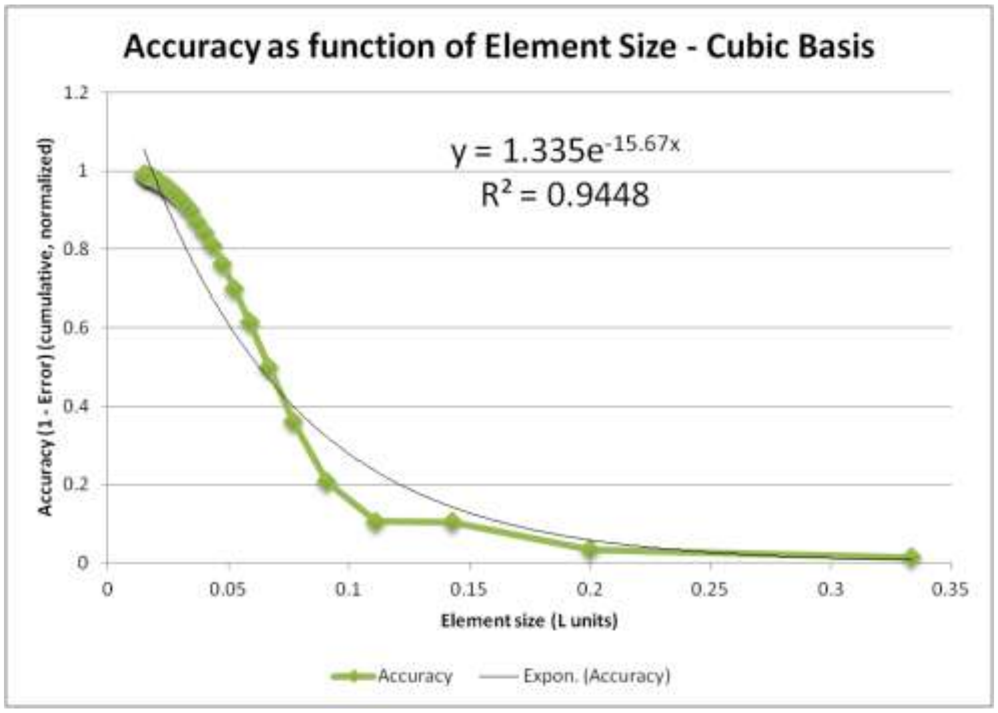


ANALYSIS:

To determine the relationship between error and element size, I used Excel's exponential trendline fit option on a data set of **Accuracy**, also known as **(1-error).** This gave me exponential decline towards 0 (rather than tangential growth). I used Excel instead of Matlab because Matlab 2009b doesn't have the curve fitting tools that Matlab 2011 has.

Regardless, I obtained the following graph for cubic elements:

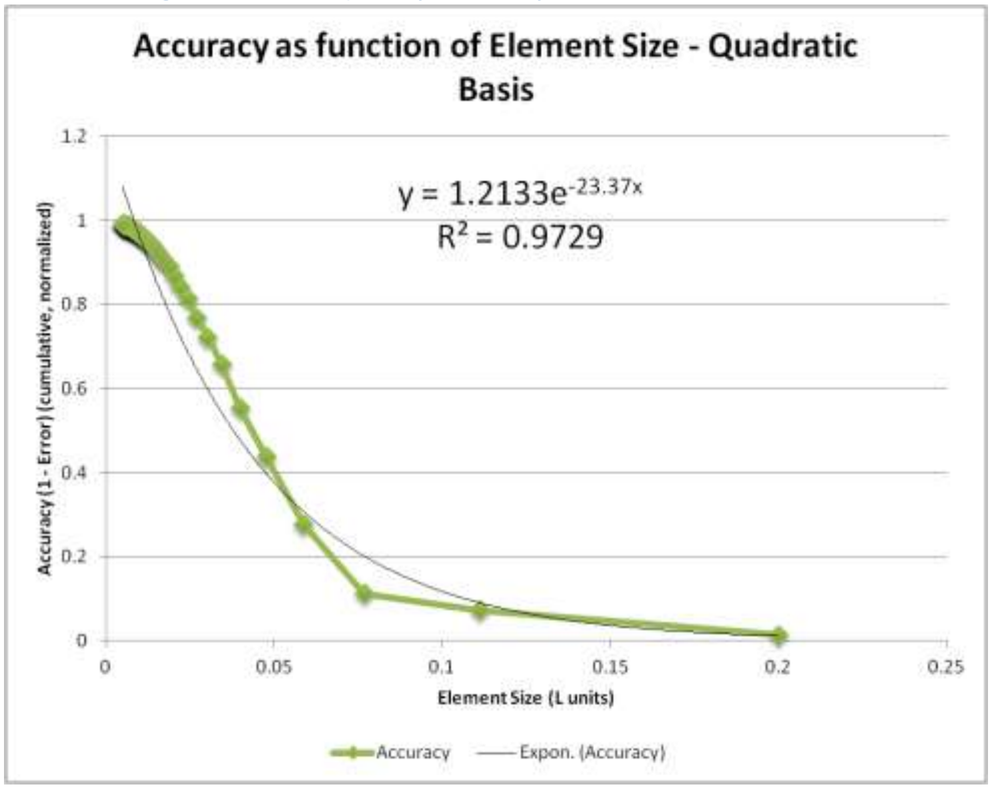
Figure 7: Accuracy as a function of Element Size - Cubic



As you can see, the error decay is exponential. The important point to note is the power of the exponent: **-15.67**. This will be important for our later comparisons.

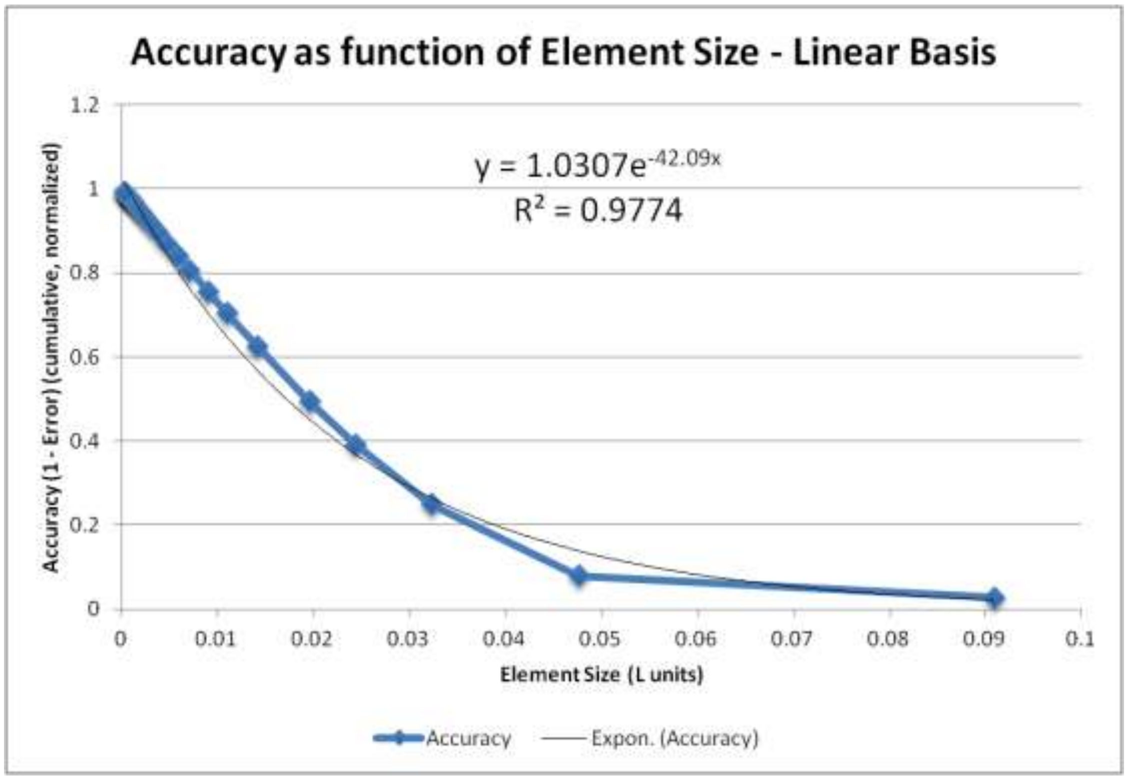
I performed the same analysis for quadratic:

Figure 8: Accuracy as a function of Element Size - Quadratic



Note the exponent of **-23.37.** Again, the same for linear:

Figure 9: Accuracy as a function of Element Size - Linear



With an exponent of **-42.09.**

A table to summarize the powers:

Table 2: Exponents of Accuracy Decay versus basis function power

Basis function power – P	Exponent of Accuracy Decay
1	-42.09
2	-23.37
3	-15.67

Because this is a graph of accuracy versus element size (with increasing element size), it can be somewhat counterintuitive to interpret these exponents. Essentially though, the exponent represents the **rate** at which the **accuracy** for a given basis element **decays** depending on **element size.** With this in mind, we see that quadratic element accuracy decays relatively slow compared to linear – it actually decays almost 3 times as slowly. Cubic elements essentially hold their accuracy for larger element sizes much better than linear elements.

Quadratic elements are (as we expect) somewhere in-between linear and cubic in terms of accuracy performance. It is important to note though that with an exponent of -23, quadratic elements perform much closer to cubic elements than linear elements. This might be an important factor in deciding which basis elements to use; essentially, quadratic elements have arguably the best accuracy relative to computation cost out of the three types of elements we have considered. Quadratic elements represent little added complexity compared to linear elements; there are many reasons for this. With only three basis functions per element, the total number of stiffness matrix numbers per element is 9. This is compared to 4 for linear elements and 16 for cubic. With 16 elements and 4x4 matrices overlapping, our matrix is not as sparse with cubic elements compared to quadratic elements.

Yet with quadratic elements, the overall size of the stiffness matrix decreases considerably relative to linear elements for minimal extra computation cost. The final stiffness matrix to obtain the desired error had 3052 degrees of freedom for linear elements; conversely, a matrix of only 403 degrees of freedom with quadratic elements satisfied the error requirements. Thus, quadratic elements represent a pleasant medium between overall matrix size and computation cost per element

Clearly there are tradeoffs between matrix sparseness, matrix size, loading computation cost, and memory overhead between the three types of basis functions we have considered here. Most likely in the future we will use linear elements due to their simplicity (especially after transitioning to 2 or 3 dimensions), but this homework served as a useful exercise in examining the benefits and drawbacks to each element type (as well as qualitatively measuring their performance).

APPENDIX:

Like in homework #1, the raw data for the above figures is not provided here but can be provided upon request. It is also quite easily generated from the included matlab code below:

```matlab
% Peter Cottle ME280a hw1!!
%clear all
clc
%close all

% Start / endpoints of the bar
domainStart = 0;
domainEnd = 1;

k = 16;
L = domainEnd - domainStart;

showextraGraphs = false;

order = 1;

% Exact solution
 exactSol = @(x) -1*(5*L^2)/(pi^2)*cos((pi*k.*x)/L) - (25*L^2*(1-
k^2))/(4*pi^2*k^2)*sin((2*pi*k*x)/(L)) + (5*L^2)/(pi^2) + x.*((-
5*L)/(pi^2) + ((-1)^k)*(5*L)/(pi^2));
 dudx = @(x) 1 * ( (5*k*L/pi) * sin((pi*k.*x)/L) - ((25*L*(1-
k^2))/(2*pi*k))*cos(2*pi*k.*x/L) + ((5*L)/(pi^2))*(-1)^(k) - 5*L/(pi^2));
loadingInZ = @(x) -1 * (k^2 * cos((pi * k * x)/L) + 5 * (1-k^2) *
sin((2*pi*k*x)/L));
A1inZ = @(x) 0.2;

% exactSol = @(x) x - x.^5;
% dudx = @(x) 1 - 5*x.^4;
% loadingInZ = @(x) 20* x.^3;
% A1inZ = @(x) 1;

%loadingInZ = @(x) 1;

exactSol = @(x) -1*(5*L^2)/(pi^2)*cos((pi*k.*x)/L) - (25*L^2*(1-
k^2))/(4*pi^2*k^2)*sin((2*pi*k*x)/(L)) + (5*L^2)/(pi^2) + x.*((1/L) + (-
5*L)/(pi^2) + ((-1)^k)*(5*L)/(pi^2));
dudx = @(x) 1 * ( (5*k*L/pi) * sin((pi*k.*x)/L) - ((25*L*(1-
k^2))/(2*pi*k))*cos(2*pi*k.*x/L) + ((1/L) + (5*L)/(pi^2))*(-1)^(k) -
5*L/(pi^2));
loadingInZ = @(x) -1 * (k^2 * cos((pi * k * x)/L) + 5 * (1-k^2) *
sin((2*pi*k*x)/L));
A1inZ = @(x) 0.2;
BC_left = 0;
BC_right = 1;



% get error norm to divide by
range = [0:0.0001:L];
eVals = 0.2 * dudx(range).*dudx(range);
eInt = trapz(range,abs(eVals));
eNorm = sqrt(eInt);
```

```matlab
function [ R ] =
computeLoading(currElement,myMesh,R,xInTermsOfZ,loadingInZ,order,exactSol)
%computeLoading computes and adds the R elements for a given element
% It takes in the phi functions in zeta domain (as function of x) and
% numerically integrates them with trapz


x_i = myMesh(1 + (currElement-1));
x_ip1 = myMesh(2 + (currElement-1));

if(order > 1)
    x_ip2 = myMesh(3 + (currElement-1));
else
    x_ip2 = 0;
end

if(order > 2)
    x_ip3 = myMesh(4 + (currElement-1)*order);
else
    x_ip3 = 0;
end

%jacobian = @(z,xi,xip1,xip2) z.*(xi + 2*xip1 + xip2) + 0.5 * (xip2 - xi);
%jacobian = @(z,xi,xip1,xip2) 0.5 * (xip2 - xi);
order
if(order == 1)
    phi1 = @(z) (-z+1)/2;
    phi2 = @(z) (z+1)/2;
    jacobian = @(z,xi,xip1,xip2) 0.5 * (xip1-xi);
elseif(order == 2)
    phi1 = @(z) z.*(z-1)/2;
    phi2 = @(z) (z-1).*(z+1)*-1;
    phi3 = @(z) (z+1).*z/2;
    jacobian = @(z,xi,xip1,xip2) z.*(xi - 2*xip1 + xip2) + 0.5 * (xip2 - xi);
elseif(order == 4) %TODO
    phi1 = @(z) 0;
end

% phi1 = @(z) z.*(z-1)/2;
% phi2 = @(z) (z-1).*(z+1)*-1;
% phi3 = @(z) (z+1).*z/2;
%
zspan = [-1:0.01:1];
figure(5)
clf
x_i
x_ip1
x_ip2
plot(zspan,xInTermsOfZ(x_i,x_ip1,x_ip2,zspan),'g*')

figure(2)
clf
```

```
if(order == 1)
    xInTermsOfZ = @(xi,xip1,xip2,z) 1/2*((xip1-xi).*z+xip1+xi);
elseif (order == 2)
    xInTermsOfZ = @(xi,xip1,xip2,z) xi.*(z-1)*0.5.*z + -xip1*(z+1).*(z-1)
+ xip2*(z+1).*z*0.5;
    %xInTermsOfZ = @(xi,xip1,xip2,z) z.^2*(0.5*xi + 0.5*xip2 - xip1) +
0.5*z*(xip2 + xi) + xip1;
end

% DONT FORGET ABOUT the -1 term (because it goes to the other side

% Initialize nodes and error
error = 10;
tol = 0.01;

% 2 for linear, 3 for quadratic, etc
N = order + 1;
%N = 3;
figure(1)
clf
plot([domainStart:0.01:domainEnd],exactSol([domainStart:0.01:domainEnd]),'
g', 'LineWidth',6);
hold on
if(showextraGraphs)
    figure(4)
    clf

plot([domainStart:0.01:domainEnd],exactSol([domainStart:0.01:domainEnd]),'
g', 'LineWidth',6);
    hold on
    errorForPlot = zeros(32,1);
end

xPointsForError = [0];
yPointsForError = [1];

% loop while error is below tolerance
while error > tol
%while N < 100
%for i=1:100
    % For "N" amount of nodes, we will have an NxN matrix. We specify that
    % there will be at most 3 nonzeros per row (or per column) in the last
    % argument
    kGlobal = spalloc(N,N,5*N);

    R = zeros(N,1);

    % Obtain the mesh
    myMesh = mesh(domainStart,domainEnd,N);

    % Loop over elements
    numElements = (length(myMesh) - 1) / order;

    for currElement=1:numElements

        % Get the K matrix
        %currElement
        kElement = miniK(myMesh,currElement,A1inZ,order);

        % Assemble into kGlobal
        kGlobal = assembleIntoGlobal(kElement,currElement,kGlobal,order);

        % Compute the loading factor...
        R =
computeLoading2(currElement,myMesh,R,xInTermsOfZ,loadingInZ,order,exactSol
);

    end

    if(showextraGraphs)

        figure(6)
        hold on
        asd = 1 + floor((N / 20));
        plot(myMesh,R*N,'LineWidth',asd)
        R
        figure(3)
        spy(kGlobal)
        full(kGlobal)
        %pause
    end



    % do the initial condition switchover, but the a's are 0 at the edges
    % so don't bother for now
    kCopy = kGlobal;

    % Impose the initial conditions by deleting 1st and last rows,
    % and deleting first and last columns
    kGlobal(N,:) = [];
    kGlobal(1,:) = [];

    %before deleting the columns, make sure to multiply them to dirichlet
    %BC's
    R_adderLeft = [0; full(kGlobal(:,1)) * BC_left; 0];
    R_adderRight = [0; full(kGlobal(:,N)) * BC_right; 0];
    R = R - R_adderLeft;
    R = R - R_adderRight;

    %delete these columns

    kGlobal(:,N) = [];
    kGlobal(:,1) = [];
```

```
hold on
plot(zspan,loadingInZ(xInTermsOfZ(x_i,x_ip1,x_ip2,zspan)),'r*')
plot(zspan,phi1(zspan),'b')
plot(zspan,phi2(zspan),'r')
if(order > 1)
    plot(zspan,phi3(zspan),'k*')
end
plot(zspan,jacobian(zspan,x_i,x_ip1,x_ip2),'g*')
%pause
figure(4)
clf
hold on
plot(zspan,phi1(zspan).*jacobian(zspan,x_i,x_ip1,x_ip2).*loadingInZ(xInTermsOfZ(x_i,x_ip1
,x_ip2,zspan)),'g','LineWidth',3)
plot(zspan,phi2(zspan).*jacobian(zspan,x_i,x_ip1,x_ip2).*loadingInZ(xInTermsOfZ(x_i,x_ip1
,x_ip2,zspan)),'b','LineWidth',3)
if(order>1)

plot(zspan,phi3(zspan).*jacobian(zspan,x_i,x_ip1,x_ip2).*loadingInZ(xInTermsOfZ(x_i,x_ip1
,x_ip2,zspan)),'r','LineWidth',3)
end


%pause


%
%
% [gaussPoints,gaussWeights] = getGauss();
% miniR = zeros(order+1,1);
% products = zeros(order+1,max(size(gaussPoints)));
%
%
% products(1,:) = (phi1(gaussPoints) .* jacobian(gaussPoints,x_i,x_ip1,x_ip2) .*
loadingInZ(xInTermsOfZ(x_i,x_ip1,x_ip2,gaussPoints)))';
% products(2,:) = (phi2(gaussPoints) .* jacobian(gaussPoints,x_i,x_ip1,x_ip2) .*
loadingInZ(xInTermsOfZ(x_i,x_ip1,x_ip2,gaussPoints)))';
%
% if(order > 1)
%     products(3,:) = (phi3(gaussPoints) .* jacobian(gaussPoints,x_i,x_ip1,x_ip2) .*
loadingInZ(xInTermsOfZ(x_i,x_ip1,x_ip2,gaussPoints)))';
% end
% if(order > 2)
%     products(4,:) = (phi4(gaussPoints) .* jacobian(gaussPoints,x_i,x_ip1,x_ip2) .*
loadingInZ(xInTermsOfZ(x_i,x_ip1,x_ip2,gaussPoints)))';
% end
%
% for(i=1:order+1)
% %     products(i)
% %     products(i).*gaussWeights
% %     pause
%     miniR(i) = sum(products(i).*gaussWeights);
%     %R(currElement*order + i - (order-0)) = R(currElement*order + i - (order-0)) +
miniR(i);
% end


%pause
%miniR
%pause
%pause



prod1 = phi1(zspan) .* jacobian(zspan,x_i,x_ip1,x_ip2) .*
loadingInZ(xInTermsOfZ(x_i,x_ip1,x_ip2,zspan));
R1 = trapz(zspan,prod1)

prod2 = phi2(zspan) .* jacobian(zspan,x_i,x_ip1,x_ip2) .*
loadingInZ(xInTermsOfZ(x_i,x_ip1,x_ip2,zspan));
R2 = trapz(zspan,prod2)

if(order > 1)
    prod3 = phi3(zspan) .* jacobian(zspan,x_i,x_ip1,x_ip2) .*
loadingInZ(xInTermsOfZ(x_i,x_ip1,x_ip2,zspan));
    R3 = trapz(zspan,prod3)
end



%
% %
if(order == 1)
    R(currElement) = R(currElement) + R1;
    R(currElement+1) = R(currElement+1) + R2
elseif (order == 2)
        R(currElement*2 - 1) = R(currElement*2 - 1) + R1;
    R(currElement*2 - 1 + 1) = R(currElement*2 - 1 + 1) + R2;
    R(currElement*2 - 1 + 2) = R(currElement*2 - 1 + 2) + R3
end





end



function [ R ] =
computeLoading2(currElement,myMesh,R,xInTermsOfZ,loadingInZ,order,exactSol)
%computeLoading computes and adds the R elements for a given element
% It takes in the phi functions in zeta domain (as function of x) and
% numerically integrates them with trapz

showextraGraphs = false;


x_i = myMesh(1 + (currElement-1)*order);
x_ip1 = myMesh(2 + (currElement-1)*order);


if(order > 1)
    x_ip2 = myMesh(3 + (currElement-1)*order);
else
    x_ip2 = 0;
end
```

```matlab
    % Do the same for R
    R(N,:) = [];
    R(1,:) = [];

    % Solve for A's based on this
    A = inv(kGlobal) * R;
    % Pad the a's for the initial conditions
    A = A;
    A = [0; A; 1];

    % Calculate error, aka integral of abs value of difference

    % display our N
    N


    %plotAnswer(myMesh,A,order,exactSol);
%     figure(1)
%     clf
%
plot(linspace(0,1,300),exactSol(linspace(0,1,300)),'g','LineWidth',3);
%     hold on
%     plot(myMesh,A,'b','LineWidth',2)
%
    error = calculateError(myMesh,A,dudx,A1inZ,eNorm,order,exactSol)

    figure(14)
    hold on
    plot(N,error,'b*');
    xPointsForError(end+1) = N;
    yPointsForError(end+1) = error;
    if(order == 3)
        N = N + order*2;
    elseif(order == 2)
        N = N + order*4;
    else
        N = N + 10*min(50,ceil(N/50));
    end

    pause(0.05)
    %pause
    if(showextraGraphs)
        %pause
        figure(4)
        clf
        figure(8)
        clf
        figure(9)
        clf
    end


end

% pause
% close all
% plot(range,eVals,'g', 'LineWidth',6)
% hold on
% plot(myMesh,A)
% title('FEM Analysis')
% legend('Exact solution','Solution with given N');
% close all
% plot(myMesh,A)
% hold on
%
plot([domainStart:0.01:domainEnd],exactSol([domainStart:0.01:domainEnd]),'
g', 'LineWidth',6);



function [ error ] = calculateError(
myMesh,A,dudx,A1inX,eNorm,order,exactSol )
%CALCULATEERROR calc error

% order here is linear, so loop through elements. aka just increment one
showextraGraphs = false;
error = 0;

if(order==1)
    for leftIndex=1:length(myMesh)-1
        rightIndex = leftIndex + 1;

        % get the x coordinate at this point in real world
        xLeft = myMesh(leftIndex);
        xRight = myMesh(rightIndex);

        % get the a's
        aLeft = A(leftIndex);
        aRight = A(rightIndex);

        % get the range over this, 100 is probably good
        xRange = linspace(xLeft,xRight,100);

        % get dudx over this
        dudxOnRange = dudx(xRange);

        % get the derivative of our approx
        % for order one is this easy
        slope = (aRight - aLeft) / (xRight - xLeft);

        dApproxOnRange = (xRange).*0 + slope;

        %calculate the integral thing. A1ofX here is just 0.2
        product = (dudxOnRange - dApproxOnRange).^2 * A1inX(xRange);
        integral = trapz(xRange,product);
        error = integral + error;
```

```matlab
if(order > 2)
    x_ip3 = myMesh(4 + (currElement-1)*order);
else
    x_ip3 = 0;
end



%construct the phi's in the x domain.
%for linear, this means subbing in (x - x_i) for z
%and having (xip1 - xi) for lengths... i think

if(order == 1)
    phi1 = @(x,xi,xip1) 1 + ((-1)/(xip1-xi)).*(x-xi);
    phi2 = @(x,xi,xip1) 1 + ((1)/(xip1-xi)).*(x-xip1);
elseif(order == 2)
    %use polyfit to make the equations, LOL!
    phi1points = [1; 0; 0];
    phi2points = [0; 1; 0];
    phi3points = [0; 0; 1];
    xPoints = [x_i;x_ip1;x_ip2];

    phi1coef = polyfit(xPoints,phi1points,2);
    phi2coef = polyfit(xPoints,phi2points,2);
    phi3coef = polyfit(xPoints,phi3points,2);

    phi1 = @(x,xi,xip1) 1 * (phi1coef(1)*x.^2 + phi1coef(2).*x + phi1coef(3));
    phi2 = @(x,xi,xip1) 1 * (phi2coef(1)*x.^2 + phi2coef(2).*x + phi2coef(3));
    phi3 = @(x,xi,xip1) 1 * (phi3coef(1)*x.^2 + phi3coef(2).*x + phi3coef(3));

elseif(order == 3) %TODO

    xPoints = [x_i;x_ip1;x_ip2;x_ip3];

    coeffs = getCubicCoeff(false,xPoints);

    phi1 = @(x,xi,xip1) coeffs(1,1)*x.^3 +  coeffs(1,2)*x.^2 + ...
                        coeffs(1,3)*x +     coeffs(1,4);

    phi2 = @(x,xi,xip1) coeffs(2,1)*x.^3 +  coeffs(2,2)*x.^2 + ...
                        coeffs(2,3)*x +     coeffs(2,4);

    phi3 = @(x,xi,xip1) coeffs(3,1)*x.^3 +  coeffs(3,2)*x.^2 + ...
                        coeffs(3,3)*x +     coeffs(3,4);

    phi4 = @(x,xi,xip1) coeffs(4,1)*x.^3 +  coeffs(4,2)*x.^2 + ...
                        coeffs(4,3)*x +     coeffs(4,4);

end
if(order == 1)
    xrange = linspace(x_i,x_ip1,100);
elseif(order == 2)
    xrange = linspace(x_i,x_ip2,200);
elseif(order == 3)
    xrange = linspace(x_i,x_ip3,300);
end



if(showextraGraphs)
    figure(4)
    plot(linspace(0,1,200),exactSol(linspace(0,1,200)),'g','LineWidth',3);
    hold on
    %plot(linspace(0,1,200),loadingInZ(linspace(0,1,200)),'k','LineWidth',2);
    plot(xrange,phi1(xrange,x_i,x_ip1))
    plot(xrange,phi2(xrange,x_i,x_ip1))
        plot(xrange,phi3(xrange,x_i,x_ip1))

        plot(xrange,phi4(xrange,x_i,x_ip1),'r','LineWidth',2)

end



product1 = phi1(xrange,x_i,x_ip1) .* loadingInZ(xrange);
integral1 = trapz(xrange,product1);

product2 = phi2(xrange,x_i,x_ip1) .* loadingInZ(xrange);
integral2 = trapz(xrange,product2);

if(order == 3)
    product4 = phi4(xrange,x_i,x_ip1) .* loadingInZ(xrange);
    integral4 = trapz(xrange,product4);
end

if(order > 1)
    product3 = phi3(xrange,x_i,x_ip1) .* loadingInZ(xrange);
    integral3 = trapz(xrange,product3);
    if(showextraGraphs)
        figure(8)
        hold on
        plot(xrange,product1,'g')
        plot(xrange,product2,'b')
        plot(xrange,product3,'r')
        plot(xrange,product4,'k')
        figure(9)
        hold on
        plot(xrange,phi1(xrange,x_i,x_ip1),'g','LineWidth',3)
        plot(xrange,phi2(xrange,x_i,x_ip1),'b','LineWidth',3)
        plot(xrange,phi3(xrange,x_i,x_ip1),'r','LineWidth',3)
        plot(xrange,phi4(xrange,x_i,x_ip1),'k','LineWidth',3)
    %integral2 = 0.5*(integral1 + integral3);
    %pause
    end
end
```

```matlab
    end
end



if(order==2)
    numElements = (length(A) - 1)/2;
    for(i=1:numElements)
        currElement = i;
        x_i = myMesh(1 + (currElement-1)*order);
        x_ip1 = myMesh(2 + (currElement-1)*order);
        x_ip2 = myMesh(3 + (currElement-1)*order);


        ypoints = [A(1 + (currElement-1)*order);...
                   A(2 + (currElement-1)*order);...
                   A(3 + (currElement-1)*order)];


        coefficients = polyfit([x_i;x_ip1;x_ip2;],ypoints,2);


        miniRange = [x_i:0.001:x_ip2];


        %take derivative...
        dApproxOnRange = coefficients(1) * miniRange * 2 +
coefficients(2);


        dExactOnRange = dudx(miniRange);


        product = (dExactOnRange - dApproxOnRange).^2 * A1inX(miniRange);
        integral = trapz(miniRange,product);
        error = integral + error;


        if(showextraGraphs)
            figure(12)
            plot(miniRange,yvals);
            figure(13)
            hold on
            plot(miniRange(2:end),dApproxOnRange,'g');
            plot(miniRange(2:end),dExactOnRange,'r');
            pause(0.05)
        end
    end
end


if(order==3)
    numElements = (length(A) - 1)/3;
    for(i=1:numElements)
        currElement = i;
        x_i = myMesh(1 + (currElement-1)*order);
        x_ip1 = myMesh(2 + (currElement-1)*order);
        x_ip2 = myMesh(3 + (currElement-1)*order);
        x_ip3 = myMesh(4 + (currElement-1)*order);



        ypoints = [A(1 + (currElement-1)*order);...
                   A(2 + (currElement-1)*order);...
                   A(3 + (currElement-1)*order);...
                   A(4 + (currElement-1)*order)];


        coefficients = polyfit([x_i;x_ip1;x_ip2;x_ip3;],ypoints,3);


        miniRange = [x_i:0.001:x_ip3];


        %take derivative...
        dApproxOnRange = coefficients(1) * miniRange.^2 * 3 + ...
                         coefficients(2) * miniRange * 2 + ...
                         coefficients(3);


        dExactOnRange = dudx(miniRange);


        product = (dExactOnRange - dApproxOnRange).^2 * A1inX(miniRange);
        integral = trapz(miniRange,product);
        error = integral + error;


        if(showextraGraphs)
            figure(12)
            plot(miniRange,ypoints);
            figure(13)
            hold on
            plot(miniRange,dApproxOnRange,'g');
            plot(miniRange,dExactOnRange,'r');
            pause(0.05)
        end
    end
end







error = sqrt(error) / eNorm;


end



function [ kGlobal ] =
assembleIntoGlobal(kElement,currElement,kGlobal,order)
%assembleIntoGlobal: This function takes a current element's K matrix
%   and assembles it into the global sparse matrix


%Inputs:
%kElement: mini 4x4 for this element
%currElement: The current element for
%kGlobal: sparse matrix to mess with
```

```matlab
if(order == 1)
    R(currElement) = R(currElement) + integral1;
    R(currElement+1) = R(currElement+1) + integral2;
elseif(order == 2)
    R(currElement*2 - 1) = R(currElement*2 - 1) + integral1;
    R(currElement*2 - 1 + 1) = R(currElement*2 - 1 + 1) + integral2;
    R(currElement*2 - 1 + 2) = R(currElement*2 - 1 + 2) + integral3;
elseif(order == 3)
    R(currElement*3 - 2) = R(currElement*3 - 2) + integral1;
    R(currElement*3 - 2 + 1) = R(currElement*3 - 2 + 1) + integral2;
    R(currElement*3 - 2 + 2) = R(currElement*3 - 2 + 2) + integral3;
    R(currElement*3 - 2 + 3) = R(currElement*3 - 2 + 3) + integral4;
end



function [ coefficients ] = getCubicCoeff(inZeta,xPoints)
%GETCUBICCOEFF Summary of this function goes here
%   Detailed explanation goes here


if(inZeta)
    xpoints = [-1; -1/3; 1/3; 1];
else
    xpoints = xPoints;
end


yPoints1 = [1; 0; 0; 0];
yPoints2 = [0; 1; 0; 0];
yPoints3 = [0; 0; 1; 0];
yPoints4 = [0; 0; 0; 1];


coef1 = polyfit(xpoints,yPoints1,3);
coef2 = polyfit(xpoints,yPoints2,3);
coef3 = polyfit(xpoints,yPoints3,3);
coef4 = polyfit(xpoints,yPoints4,3);


coefficients = [coef1; coef2; coef3; coef4];


end



function [ gaussPoints,gaussWeights ] = getGauss()
%GETGAUSS Summary of this function goes here
%   Detailed explanation goes here


gaussPoints = zeros(5,1);
gaussWeights = zeros(5,1);
% weight1 = 128/255;
% weight23 = (322 + 13*sqrt(70))/(900);
% weight45 = (322 - 13*sqrt(70))/(900);
%
% point1 = 0;
% points23 = (1/3) * sqrt(5 - 2 * sqrt(10/7));
% points45 = (1/3) * sqrt(5 + 2 * sqrt(10/7));


gaussPoints = [0.1834346424956498049394761;-0.1834346424956498049394761;
               0.5255324099163289858177390;-0.5255324099163289858177390;
               0.7966664774136267395915539;-0.7966664774136267395915539;
               0.9602898564975362316835609;-0.9602898564975362316835609];


gaussWeights = [0.3626837833783619829651504;0.3626837833783619829651504;
                0.3137066458778872873379622;0.3137066458778872873379622;
                0.2223810344533744705443560;0.2223810344533744705443560;
                0.1012285362903762591525314;0.1012285362903762591525314];


%   gaussPoints = [-0.9061798459386639927976269;
%         -0.5384693101056830910363144;
%         0.0000000000000000000000000;
%         0.5384693101056830910363144;
%         0.9061798459386639927976269];
%
%   gaussWeights = [0.2369268850561890875142640;
%        0.4786286704993664680412915;
%        0.5688888888888888888888889;
%        0.4786286704993664680412915;
%        0.2369268850561890875142640];



end



function [nodeLocs] = mesh(domainStart,domainEnd,N)
%MESH: This function will mesh a 1D domain (from 0->L or start->finish)
%into N number of nodes, all equally spaced. It assumes linear basis
%functions so it will not need to worry about hitting multiples of 3


%Inputs:
%domainStart:Domain starting point
%domainEnd: Domain ending point
%N: Number of nodes for this domain (has to be >=2)


%Outputs:
%nodeLocs:  An array of the location of all the nodes over the start->end
   %        domain


%for now, this is easy
nodeLocs = linspace(domainStart,domainEnd,N);


end


function [ kElement ] = miniK( myMesh,currElement,A1inZ,order )
%miniK Takes in a mesh and a current element (for 1D) and
```

```matlab
%Outputs:
%None

% get 1 for el 1, 9*(currElement - 1) otherwise
if(order == 1)
    startingEdge = currElement;
elseif(order == 2)
    startingEdge = currElement * 2 - 1;
elseif(order == 3)
    startingEdge = currElement * 3 - 2;
end

% for 1D this is easy.
for i=0:order
    for j=0:order
        kGlobal(startingEdge+i,startingEdge+j) =
kGlobal(startingEdge+i,startingEdge+j) + kElement(i+1,j+1);
    end
end

function [ output_args ] = plotAnswer(myMesh,A,order,exactSol)
%PLOTANSWER Summary of this function goes here
%   Detailed explanation goes here
    xRange = linspace(0,1,400);


    figure(1)
    %clf
    %plot(xRange,exactSol(xRange),'g','LineWidth',3)
    pString = 'k';
    hold on
    if(order==1)
        %easy
        plot(myMesh,A);
    end
    if(order==2)
        %loop element by element... ugh
        length(A);
        numElements = (length(A) - 1)/2;
        for(i=1:numElements)
            currElement = i;
            x_i = myMesh(1 + (currElement-1)*order);
            x_ip1 = myMesh(2 + (currElement-1)*order);
            x_ip2 = myMesh(3 + (currElement-1)*order);

            ypoints = [A(1 + (currElement-1)*order);...
                       A(2 + (currElement-1)*order);...
                       A(3 + (currElement-1)*order)];

            coefficients = polyfit([x_i;x_ip1;x_ip2;],ypoints,2);

            miniRange = linspace(x_i,x_ip2,50);
            yvals = coefficients(1) * miniRange.^2 + coefficients(2) *...
                            miniRange + coefficients(3);
            plot(miniRange,yvals,pString,'LineWidth',3);
        end
    end
    if(order==3)
        numElements = (length(A)-1)/3;
        for(i=1:numElements)
            currElement = i;
            x_i = myMesh(1 + (currElement-1)*order);
            x_ip1 = myMesh(2 + (currElement-1)*order);
            x_ip2 = myMesh(3 + (currElement-1)*order);
            x_ip3 = myMesh(4 + (currElement-1)*order);

            ypoints = [A(1 + (currElement-1)*order);...
                       A(2 + (currElement-1)*order);...
                       A(3 + (currElement-1)*order);...
                       A(4 + (currElement-1)*order)];

            coefficients = polyfit([x_i;x_ip1;x_ip2;x_ip3;],ypoints,3);

            miniRange = linspace(x_i,x_ip3,80);
            yvals = coefficients(1) * miniRange.^3 + coefficients(2) *...
                            miniRange.^2 + coefficients(3)*miniRange + ...
                                coefficients(4);
            plot(miniRange,yvals,pString,'LineWidth',2);
        end
    end



        %pause



en
```

```matlab
%   computes the resulting kElement matrix. This might need to call
%   the loading function....?

% z-space integral is just  -1_|^^+1 of phi^1 prime * phi^2 prime * J^-1
% where J is the jacobian
% Thus, this is harder with quadratic basis functions :O
% I think we still need to hardcode some stuff though

x_i = myMesh(1 + (currElement-1)*order);
x_iPlus1 = myMesh(2 + (currElement-1)*order);

if(order > 1)
    x_iPlus2 = myMesh(3 + (currElement-1)*order);
else
    x_iPlus2 = 0;
end

if(order > 2)
    x_iPlus3 = myMesh(4 + (currElement-1)*order);
else
    x_iPlus3 = 0;
end
%pause

kElement = zeros(order+1);
if(order == 1)
    phi1prime = @(z) -1/2;
    phi2prime = @(z) 1/2;
    jacobian = @(z,xi,xip1,xip2) 0.5 * (xip1-xi);
elseif(order == 2)
    phi1prime = @(z) z - 1/2;
    phi2prime = @(z) -2*z;
    phi3prime = @(z) z + 1/2;
    jacobian = @(z,xi,xip1,xip2) z.*(xi - 2*xip1 + xip2) + 0.5 * (xip2 - xi);
elseif(order == 3)
    coefficients = getCubicCoeff(true,[0]);
    coeffs = coefficients;

    phi1prime = @(x)    coefficients(1,1) * 3 * x.^2 + ...
                        coefficients(1,2) * 2 * x    + ...
                        coefficients(1,3);

    phi2prime = @(x)    coefficients(2,1) * 3 * x.^2 + ...
                        coefficients(2,2) * 2 * x    + ...
                        coefficients(2,3);

    phi3prime = @(x)    coefficients(3,1) * 3 * x.^2 + ...
                        coefficients(3,2) * 2 * x    + ...
                        coefficients(3,3);

    phi4prime = @(x)    coefficients(4,1) * 3 * x.^2 + ...
                        coefficients(4,2) * 2 * x    + ...
                        coefficients(4,3);
    jacobian = @(z,xi,xip1,xip2,xip3) xi * phi1prime(z) + xip1 * phi2prime(z) + ...
                                      xip2 * phi3prime(z) + xip3 * phi4prime(z);
end

for row=1:order+1
    for col=1:order+1

        %ok so this is tricky. Get the left function first
        if(row == 1)
            left = phi1prime;
        elseif (row == 2)
            left = phi2prime;
        elseif (row == 3)
            left = phi3prime;
        else
            left = phi4prime;
        end

        if(col == 1)
            right = phi1prime;
        elseif (col == 2)
            right = phi2prime;
        elseif (col == 3)
            right = phi3prime;
        else
            right = phi4prime;
        end

        [gaussPoints,gaussWeights] = getGauss();
        zspan = linspace(-1,1,200);

        % multiply this all out
        %completeProduct = A1inZ(zspan) .* left(zspan) .* right(zspan) ./ jacobian(zspan,
x_i, x_iPlus1, x_iPlus2);
        %completeProduct = A1inZ(zspan) .* left(zspan) .* right(zspan) .* 2 ./ (-x_i +
x_iPlus2);
        if(order ~= 3)
            completeProduct = A1inZ(gaussPoints) .* left(gaussPoints) .*
right(gaussPoints) ./ jacobian(gaussPoints, x_i, x_iPlus1, x_iPlus2);
        else
            completeProduct = A1inZ(gaussPoints) .* left(gaussPoints) .*
right(gaussPoints) ./ jacobian(gaussPoints, x_i, x_iPlus1, x_iPlus2,x_iPlus3);
        end
        integral = sum(completeProduct .* gaussWeights);
        %this belongs here now
        kElement(row,col) = integral;


    end
end
end
```