

## Introduction:

So far in the class we have only used the finite element method to solve statically-loaded one dimensional problems. Although the finite element method is widely used for stress, strain, crack, and failure analysis, the *theoretical* basis for the finite element method can really work on almost any ODE similar to the one we have examined so far. All of the steps taken to obtain the weak form (like the weakening of the differentiability requirement, the basis elements, etc) are not exclusive to the stress and strain problems we have analyzed but rather work for any problem.

In order to prove this to ourselves, in this homework we will examine the thermal diffusion equation and solve for a temperature distribution over time. We will also examine the Implicit and Explicit formulations and code the Implicit form. This exercise will help the students in the class realize that the finite element has a wide range of applications.

## Objectives:

In this homework we have two main challenges. The first is to analytically derive the weak form for both the implicit and explicit Euler formulations of the heat diffusion equation. The second is to take our existing FEM code and modify it to solve a temperature profile problem over time. Interestingly enough, the actual coding for this is rather trivial once a firm understanding of the weak form is obtained. In summary:

- Take the heat diffusion equation and perform the same steps to obtain the weak form
- With this weak form, substitute in the necessary derivative approximations for both Implicit (backward) and Explicit (forward) Euler formulations
- Expand these weak forms with test functions and basis elements in order to obtain the general matrix formulation of the problem
- Code the Implicit formulation and solve the temperature profile problem given in the homework.
- Analyze this problem with different time steps, track the solution at different points, and compare this solution with the steady-state solution.

## Procedure / Implementation:

The procedure for programming this homework was quite straightforward and brief. Consult homeworks 1-4 for more details on how to implement the basic FEM solver, and make the following modifications:

- Modify the stiffness matrix code to use the piece-wise D values instead of  $A1(x)$ . This involves making sure the Gaussian points are transformed to the z world with the appropriate mapping and the full product is divided by the Jacobian.

- Calculate an additional integral for each element in the stiffness matrix that is a function of both the piece-wise Tau value and the timestep used in the formulation. This function is multiplied by the phi primes and integrated accordingly with the appropriate mapping and multiplied by the Jacobian.
- The right-hand-side of the overall equation is now an entire matrix that is then multiplied by the last-known solution. This right-hand-side element-by-element product is simply a function of the time step and the corresponding phi's, but this process is still necessary in order to obtain the correct R vector. As you can see from the below analytical solution, this right-hand-side in the Implicit formulation represents a critical part of the time derivative approximation of c.
- Finally, perform a loop to calculate the solution at every timestep and plot this data later.

The procedure for finding the analytical solution of the implicit and explicit weak form was rather lengthy. This is detailed below:

First, start with the ODE given in the problem description:

$$\frac{d}{dx}\left(D\frac{dc}{dx}\right) - \tau c + z - \dot{c} = 0$$

We must first weaken the differentiability constraints. For the following steps, we will ignore the z term for it is a constant offset. First, multiply by the test function v:

$$\left(\frac{\partial}{\partial x}\left(D\frac{\partial c}{\partial x}\right) - \tau c - \dot{c}\right)v = 0$$

Next, integrate over the domain and set equal to 0

$$\int_0^L \left(\frac{\partial}{\partial x}\left(D\frac{\partial c}{\partial x}\right) - \tau c - \dot{c}\right)v dx = 0$$

Substitute the “sleight of hand” chain rule manipulation to bring the v test function inside the integral

$$\begin{aligned}
& - \int_0^L \frac{\partial v}{\partial x} D \frac{\partial c}{\partial x} dx + \int_0^L \frac{\partial}{\partial x} \left( D \frac{\partial c}{\partial x} v \right) dx \\
& - \int_0^L \tau c v dx - \int_0^L \dot{c} v dx = 0
\end{aligned}$$

Noticing the simple integral for one of the D terms:

$$\begin{aligned}
& \int_0^L \frac{\partial v}{\partial x} D \frac{\partial c}{\partial x} dx + \int_0^L \tau c v dx + \int_0^L \dot{c} v dx = \\
& D \frac{\partial c}{\partial x} v \Big|_0^L
\end{aligned}$$

We define the D evaluation term as g for the following steps and combine like terms for the final weak form:

$$\begin{aligned}
& D \frac{\partial c}{\partial x} = g(l) \\
& \int_0^L \frac{\partial v}{\partial x} D \frac{\partial c}{\partial x} dx + \int_0^L (\tau c + \dot{c}) v dx = g(l) v(l) \\
& \forall v
\end{aligned}$$

(Equation V)

At this point we diverge for the derivation of the Explicit versus the derivation of the Implicit. In the implicit formulation, we will take all c evaluations to be on timestep in the future (aka  $c(t + \Delta T)$ ); for the explicit, we will take these to be evaluated at their current values. This will severely affect the final outcome of the system of equations, as you will see in a moment. For now, the **IMPLICIT DERIVATION:**

Take c to be evaluated at t plus delta t and find an approximation for c dot:

$$\dot{c} = \frac{c(t + \Delta t) - c(t)}{\Delta t}$$

$$c = c(t + \Delta t)$$

Substitute these into the above expressions:

$$\int_0^L \frac{\partial v}{\partial x} D \frac{\partial c(t + \Delta t)}{\partial x} dx + \int_0^L \left[ (\tau c(t + \Delta t) + \frac{c(t + \Delta t) - c(t)}{\Delta t}) \right] v dx = g^{t+\Delta t}(l)v(l)$$

Move the  $c(t)$  term to the right hand side:

$$\int_0^L \frac{\partial v}{\partial x} D \frac{\partial c(t + \Delta t)}{\partial x} dx + \int_0^L \left[ (\tau c(t + \Delta t) + \frac{c(t + \Delta t)}{\Delta t}) \right] v dx = g^{t+\Delta t}(l)v(l) + \int_0^L v \frac{c(t)}{\Delta t} dx$$

We now have a weak form that we can substitute into a series of basis functions as an approximation, similar to the weak form to matrix system of equations from the previous homeworks. Please see these homeworks for the exact details.

$$\sum_{j=1}^N a_j^{L+1} \left( \int_0^L \phi_i' D(x) \phi_j' dx + \int_0^L \phi_i \left( \tau + \frac{1}{\Delta t} \right) \phi_j dx \right) =$$

$$\sum_{j=1}^N a_j^L \left( \int_0^L \frac{\phi_j \phi_i}{\Delta t} dx \right) + \phi_i g^{L+1} \big|_{x=l}$$

This forms the basis for our FEM approximation and method of solution. The top term can be thought of as the “stiffness matrix”, and the bottom term can be thought of as the past solution ( $a^L$ ) multiplied by the right-hand-side matrix plus boundary flux condition terms. In our particular problem, there are no specific boundary flux terms (only temperatures) so that term is zero.

**For the EXPLICIT** formula, we must go back to Equation V and take a different approximation of  $\dot{c}$  and a different evaluation of  $c$ :

$$\dot{c} = \frac{c(t) - c(t - \Delta t)}{\Delta t}$$

$$c = c(t)$$

This small change will propagate to the rest of the derivation of the weak form and FEM approximation. Substitute these expressions into Equation V:

$$\int_0^L \frac{\partial v}{\partial x} D \frac{\partial c(t)}{\partial x} dx$$

$$+ \int_0^L \left[ (\tau c(t) + \frac{c(t) - c(t - \Delta t)}{\Delta t}) \right] v dx = g^t(l) v(l)$$

Separate out the  $c(t - \Delta t)$  term and re-arrange:

$$\int_0^L \frac{\partial v}{\partial x} D \frac{\partial c(t)}{\partial x} dx + \int_0^L \left[ (\tau c(t) + \frac{c(t)}{\Delta t}) v \right] dx =$$

$$g^t(l) v(l) + \int_0^L v \frac{c(t - \Delta t)}{\Delta t} dx$$

Next, perform the same FEM approximation and phi basis function substitution as shown in the earlier step:

$$\sum_{j=1}^N a_j^L \left( \int_0^L \phi_i' D(x) \phi_j' dx + \int_0^L \phi_i \left( \tau + \frac{1}{\Delta t} \right) \phi_j dx \right) =$$

$$\sum_{j=1}^N a_j^{L-1} \left( \int_0^L \frac{\phi_j \phi_i}{\Delta t} dx \right) + \phi_i g^L|_{x=l}$$

#### Noting the differences between the Explicit and Implicit Formulations:

It is important to note the difference between these two formulations when they are generalized out to systems of equations. The Implicit method generalizes to:

$$[K + M] a(t + \Delta t) = R_1(z(t + \Delta t)) + [T] a(t) + R_2(g(t + \Delta t))$$

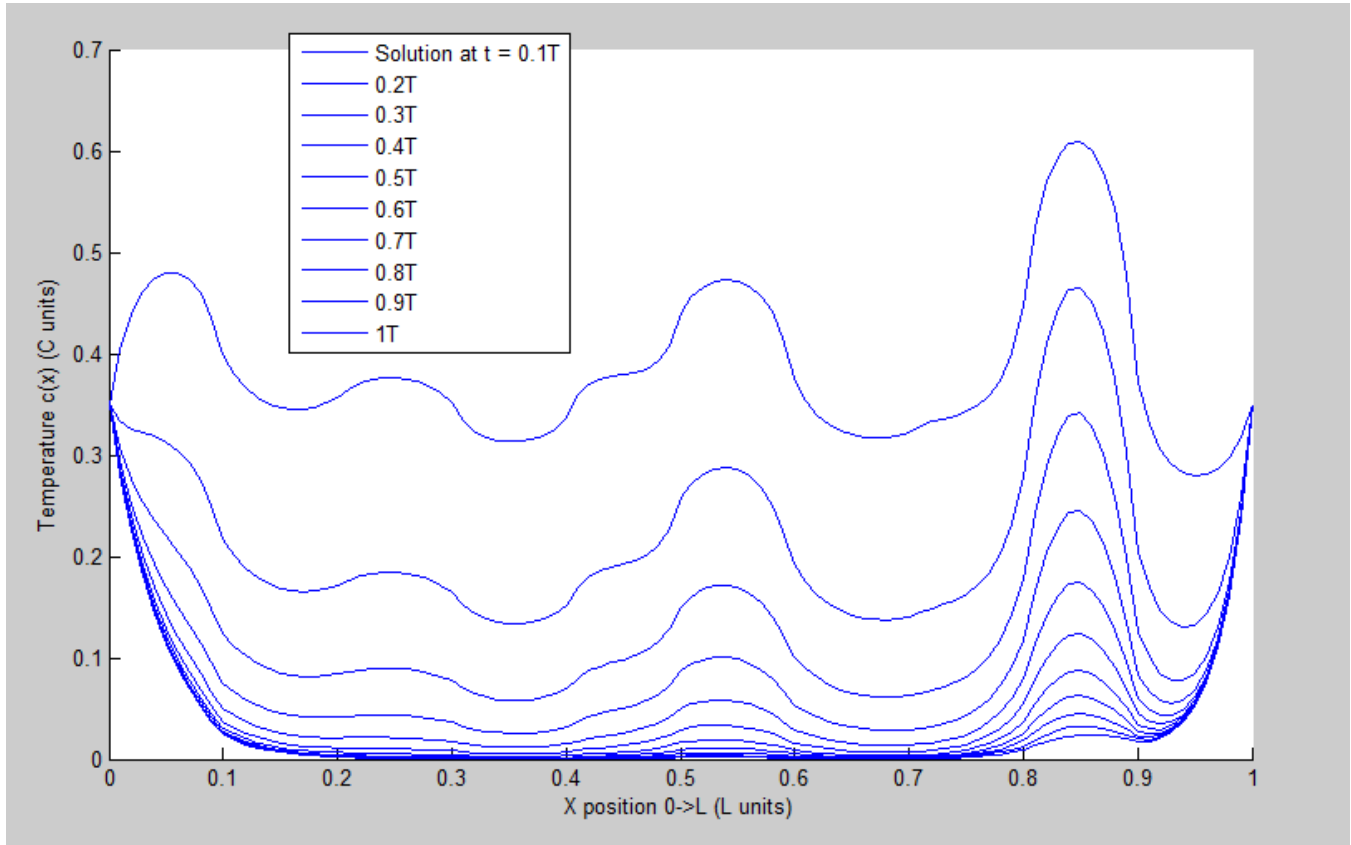
Whereas the Explicit method generalizes to:

$$[M] a(t + \Delta t) = -[K] a(t) + R_1(z(t)) - [T] a(t) + R_2(g(t))$$

The important thing to note here is that the explicit method just requires a simple update of the solution every time step by matrix vector multiplication, assuming that the M matrix is a lumped-mass matrix which is trivial to compute. The **implicit** method involves solving a series of equations **every time step** for the next configuration. Hence, it requires much more computational power, but at the added expense of a more accurate solution.

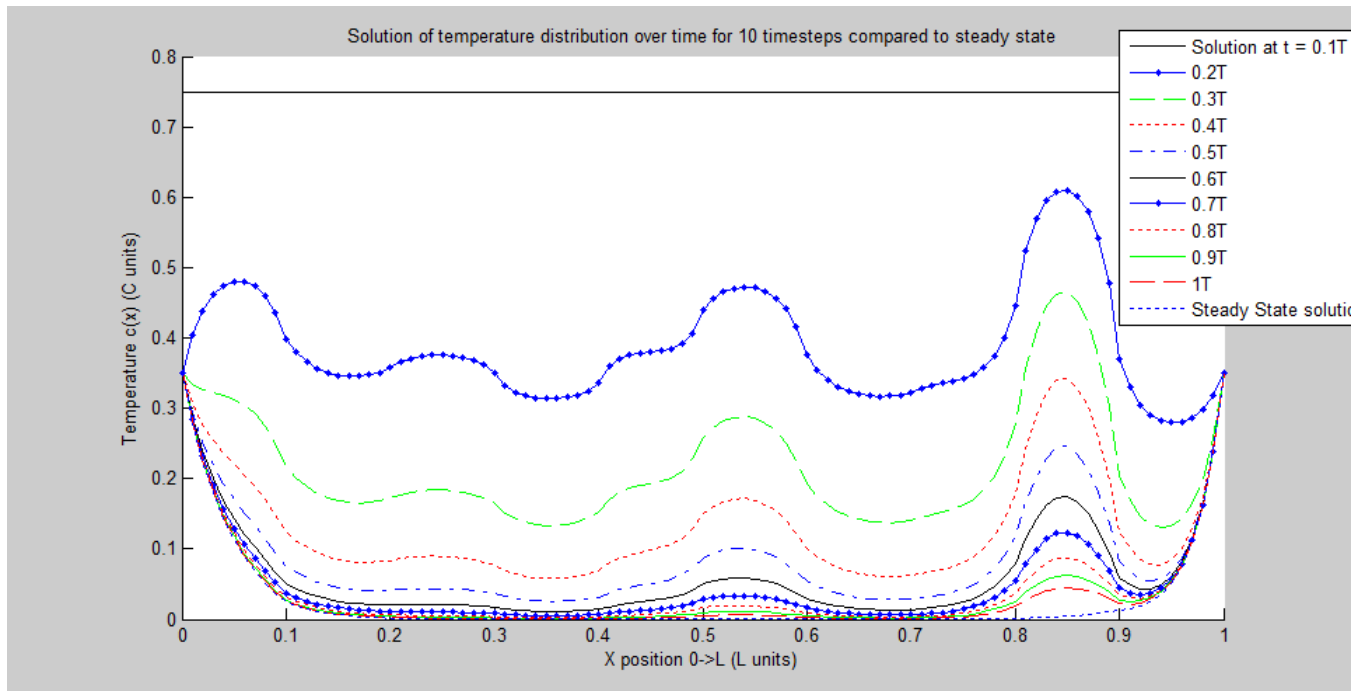
## Findings

After programming the FEM solver, I got the following plot of the solution over time:

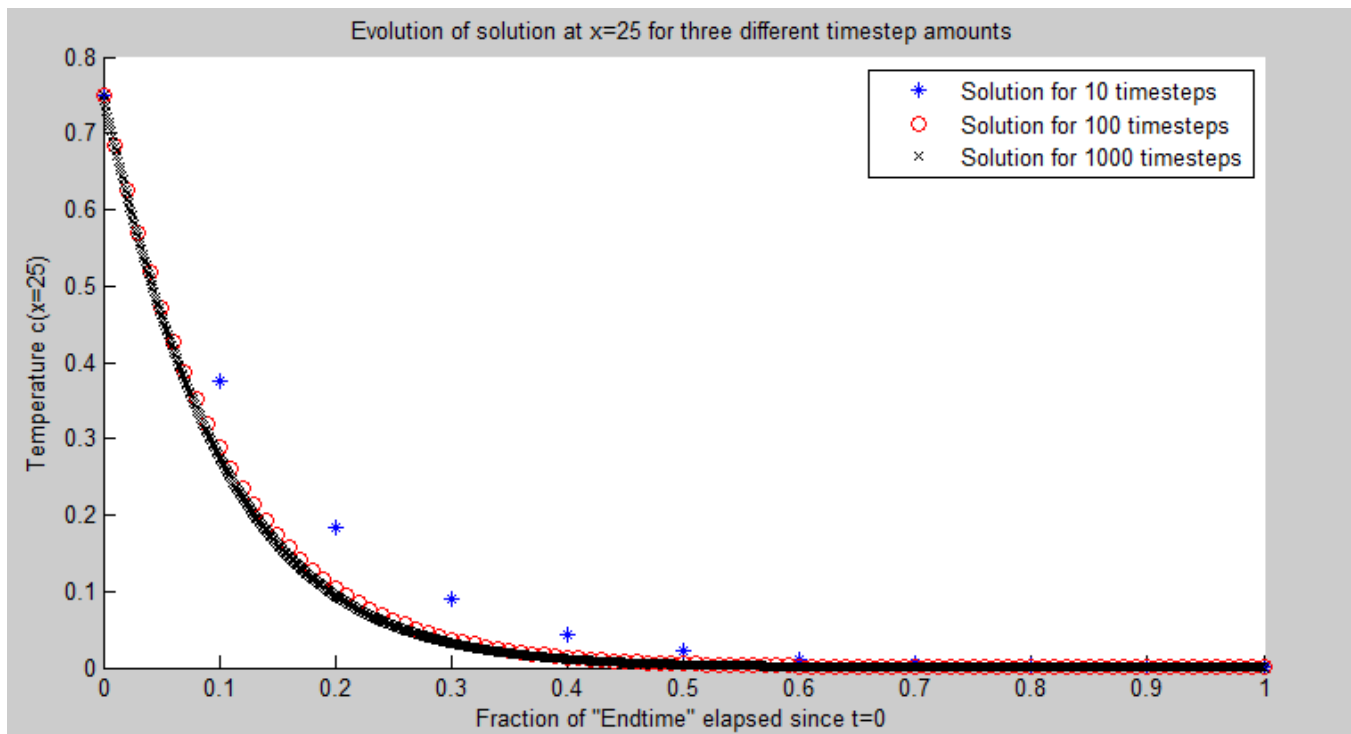


I know this plot has multiple lines of the same type, but I mainly included it just as a demonstration of what the output looks like.

Here is a better plot of the solution over time compared to the steady state solution:



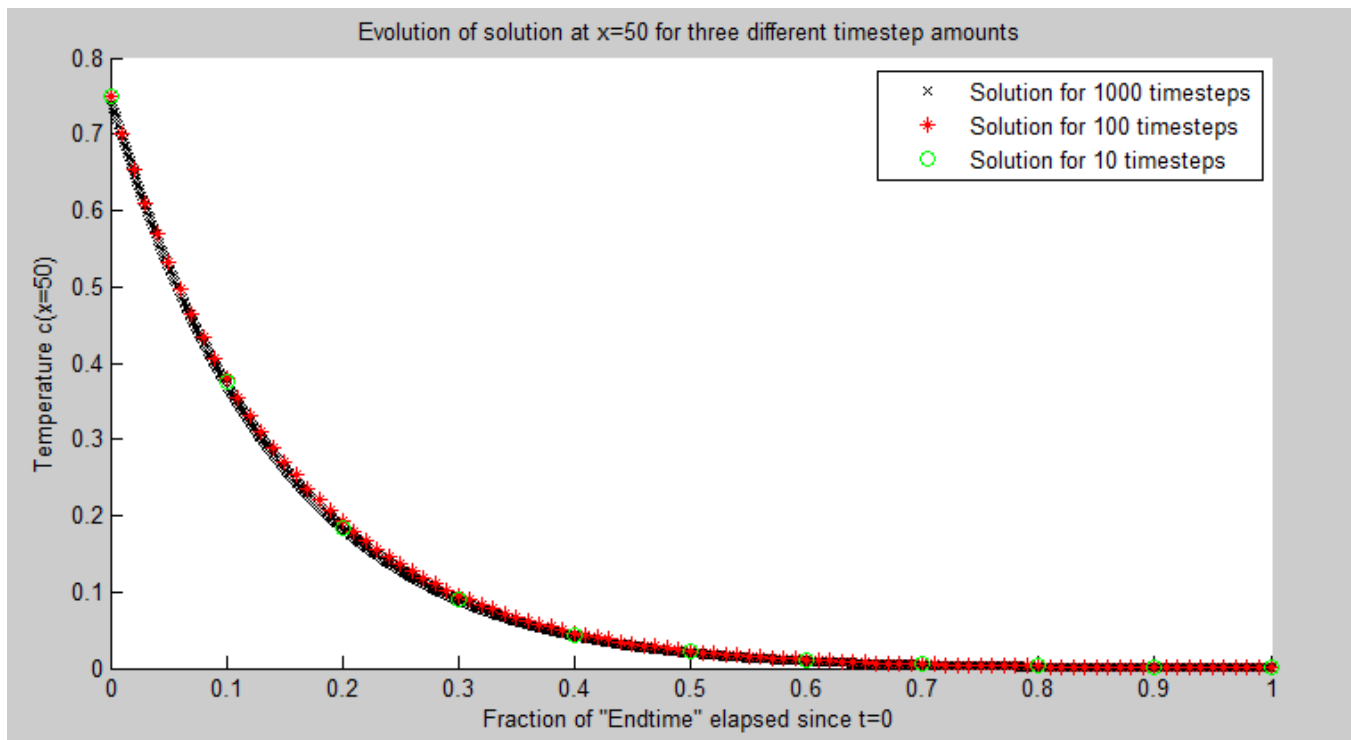
This is for 10 time steps until the end time. Tracking the solution at  $x = 0.25$ :



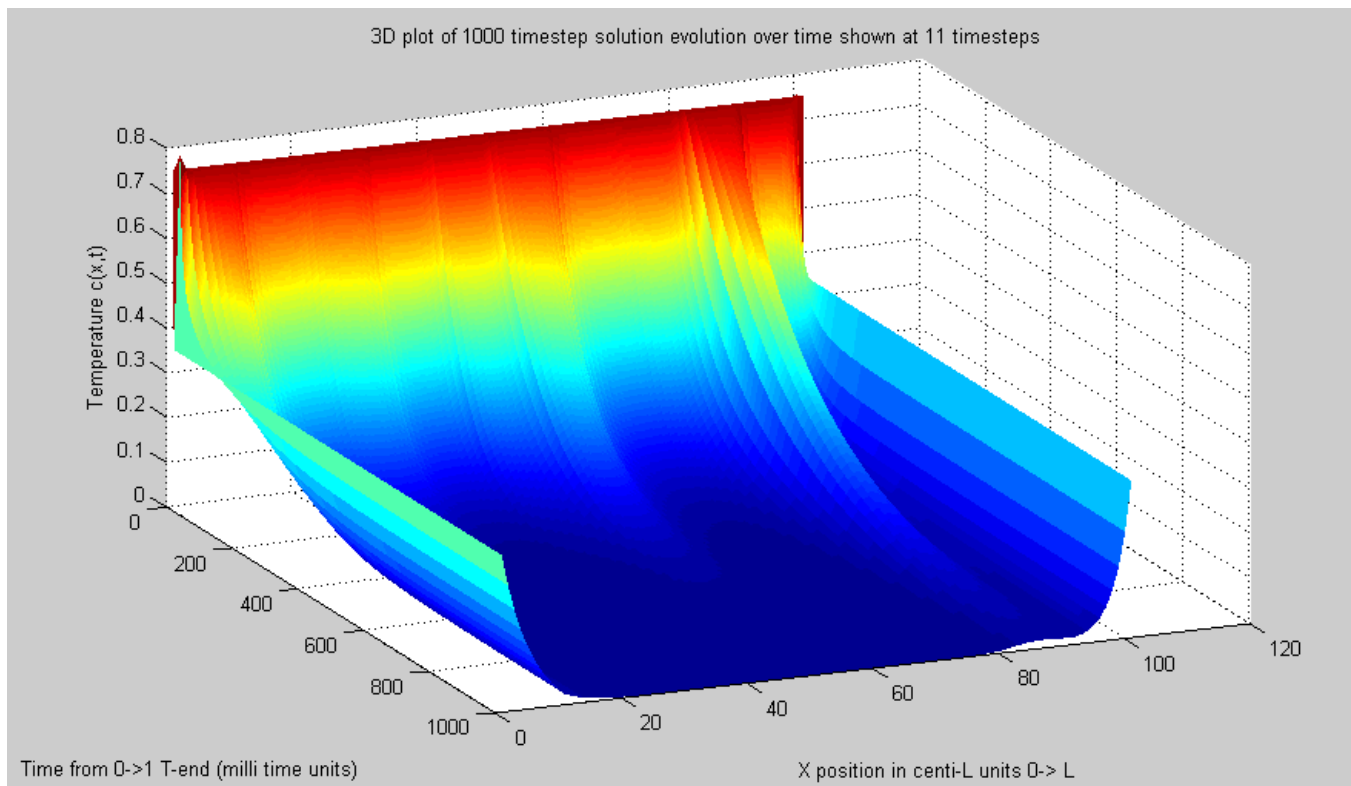
(Note: for these plots,  $x=25$  refers to the position in centi-units for  $L$ , meaning it corresponds to  $x=0.25$  in normal  $L$  units)

Tracking the solution at  $x = 0.5$ :

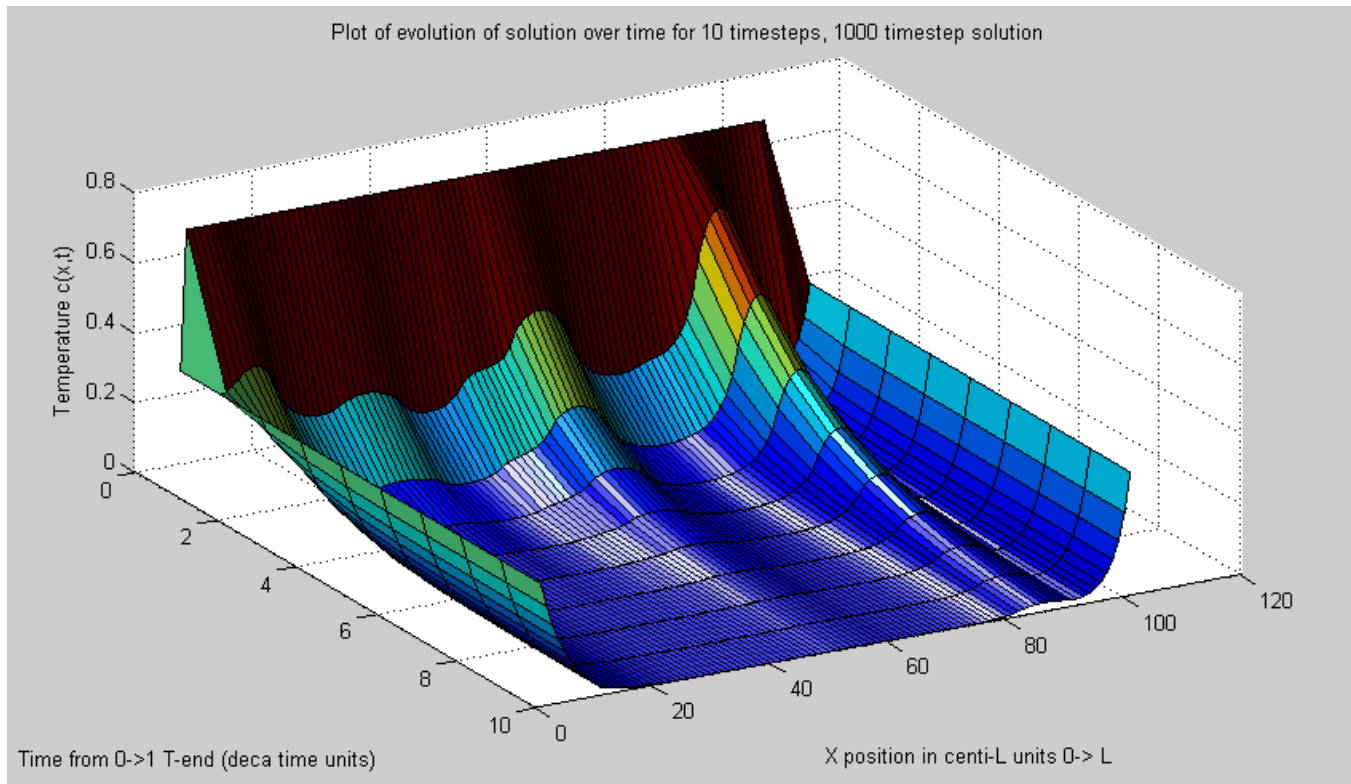




Plotting the \_entire\_ dataset for the 1000 timestep solution over time:



Plotting just the  $T = 0, 0.1T_{end}, 0.2T_{end}, \dots, 1T_{end}$  solutions for the temperature diffusion problem:



The xlabel correlates each line to the particular timestep.

## Discussion

There are some interesting points to note from these plots. Firstly, the solution tracking at  $x=0.25$  shows how larger timesteps can lead to a considerably different solution paths from the initial configuration to the final steady-state solution. For our simulation, all the steady-state solutions were the same for different timesteps, but if we solved a particularly turbulent problem with multiple equilibrium states instead, that may not always be the case. Hence, it pays to have at least have a fair ( $\geq 100$ ) number of timesteps in the simulation. If Runge-Kutta, midpoint, or some other type of iterative scheme was used instead, larger timesteps might be acceptable.

The solution tracking at  $x=0.5$  then shows us how large timesteps do not actually matter in certain areas of the solution. This is most likely because  $x=0.5$  is a location that experiences little rapid change between timesteps, but this communicates the importance of  $D$  and  $\tau$  in running these simulations. Higher values of source or sinks and higher values of thermal diffusion can lead to rapidly-changing results which will affect what timestep is necessary.

The 3D plot gives us an idea of the evolution of the entire solution over time. We can see how the initial high temperature diffuses out through the environment to the steady state solution. We can also see how areas of high thermal diffusivity (coefficient  $D$ ) shed their heat faster, and areas of low  $\tau$  coefficients

(the sink coefficient) retain their heat for longer. Because of the 0.35 boundary conditions, the edges of the graph are always held at 0.35. At the final steady state solution, the middle of the theoretical “bar” is at 0 temperature, but the edges of the bar are held at 0.35 so this forms a temperature gradient along the profile of the bar. We also can see that this temperature gradient is not linear but rather exponential, as we would expect from temperature diffusion.

In conclusion, this has been a beneficial exercise in looking at another application of the finite element method for solving ordinary differential equations. This also gives us some insight into the differences between the explicit and implicit Euler formations and the importance of choosing a correct timestep. These will all be useful tools going forward in research or in industry.

## 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.

```
% Peter Cottle ME280a hw5!!
%clear all
clc
%close all

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

L = domainEnd - domainStart;

BC_left = 0.35;
BC_right = 0.35;

initial_temp = 0.75;

z2xDomain = @(z,xi,xipl) 1/2*((xipl-xi).*(z+xipl+xi));

numSteps = 1000;
N = 101;
deltaX = 0.01;
Davg = 1.585 * 10^-6;
endTime = 100 * ((deltaX^2)/Davg);
totalT = 0;
deltaT = endTime / numSteps;

%initial thing
myMesh = mesh(domainStart,domainEnd,N);
lastA = myMesh.*0 + initial_temp;
allTheA = zeros(N,numSteps + 1);
allTheA(:,1) = lastA;

whichStep = 1;

% loop while error is below tolerance
while totalT <= endTime
    allTheA(:,whichStep) = lastA;
    whichStep = whichStep + 1;
    % 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,3*N);

    % R is now a matrix...
    rGlobal = zeros(N,N);

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

    for currElement=1:numElements

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

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

        % Compute the loading factor...

function [ kGlobal ] = assembleIntoGlobal(kElement,currElement,kGlobal)
%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

%Outputs:
%None

% get 1 for el 1, 9*(currElement - 1) otherwise
startingEdge = currElement;
order = 1;
% 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 [ dVector ] = DinX( xVector )
%DINX Summary of this function goes here
% Detailed explanation goes here

result = zeros(size(xVector));

for(i = 1:length(result))
    xPos = xVector(i);

    if(xPos <= 0.1)
        Dval = 1.5;
    elseif(xPos <= 0.2)
        Dval = 2.9;
    elseif(xPos <= 0.3)
        Dval = 2.25;
    elseif(xPos <= 0.4)
        Dval = 1.15;
    elseif(xPos <= 0.5)
        Dval = 0.55;
    elseif(xPos <= 0.6)
        Dval = 1.25;
    elseif(xPos <= 0.7)
        Dval = 2.25;
    elseif(xPos <= 0.8)
        Dval = 1.25;
    elseif(xPos <= 0.9)
        Dval = 0.75;
    else
        Dval = 2.0;
    end

    result(i) = Dval * 10^-6;
end

dVector = result;

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

```

        rElement = miniR(myMesh,currElement,deltaT);
        rGlobal =
assembleIntoGlobal(rElement,currElement,rGlobal);
    end

    R = lastA * rGlobal;
    R = R';
    % 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) = [];

    % 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 = [BC_left; A; BC_right];

    % Calculate error, aka integral of abs value of difference

    % display our N
    %figure(14);
    %hold on;
    %plot(myMesh,A);
    lastA = A';
    totalT = totalT + deltaT;
    %pause(0.005)
    whichStep
    %pause

end

function [ rElement ] = miniR(myMesh,currElement,deltaT)
%miniR Takes in a mesh and a current element (for 1D) and
% 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
order = 1;
x_i = myMesh(1 + (currElement-1)*order);
x_iPlus1 = myMesh(2 + (currElement-1)*order);

rElement = zeros(2);

phi1 = @(z) (-z+1)/2;
phi2 = @(z) (z+1)/2;

philprime = @(z) -1/2;
phi2prime = @(z) 1/2;

jacobian = @(z,xi,xipl) 0.5 * (xipl-xi);

[gaussPoints,gaussWeights] = getGauss();

%check our mapping
for row=1:order+1
    for col=1:order+1

        %ok so this is tricky. Get the left function first
        if(row == 1)
            left = philprime;
            leftNorm = phil;
        elseif (row == 2)
            left = phi2prime;
            leftNorm = phi2;
        end

        if(col == 1)
            right = philprime;
            rightNorm = phil;
        elseif (col == 2)
            right = phi2prime;
            rightNorm = phi2;
        end

        % multiply this all out
        completeProductD = DinX(z2xDomain(gaussPoints,x_i,x_iPlus1)) ...
            .* left(gaussPoints) .* right(gaussPoints) ...
            ./ jacobian(gaussPoints, x_i, x_iPlus1);

        integralD = completeProductD' * gaussWeights;

        completeProductT = ((1/deltaT) + ...
            TauInX(z2xDomain(gaussPoints,x_i,x_iPlus1))) ...
            .* leftNorm(gaussPoints) ...
            .* rightNorm(gaussPoints) .* ...
            jacobian(gaussPoints,x_i,x_iPlus1);

        integralT = completeProductT' * gaussWeights;

        %
        % integralD
        % integralT
        % integralT + integralD
        % pause

        %this belongs here now
        kElement(row,col) = integralD + integralT;

    end
end
% kElement
% pause

end

function [ tauVector ] = TauInX( xVector )
%DINX Summary of this function goes here
% Detailed explanation goes here

result = zeros(size(xVector));

for(i = 1:length(result))
    xPos = xVector(i);

    if(xPos <= 0.1)

```

```

% 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

z2xDomain = @(z,xi,xipl) 1/2*(xipl-xi).*(z+xipl+xi);
order = 1;
x_i = myMesh(1 + (currElement-1)*order);
x_iPlus1 = myMesh(2 + (currElement-1)*order);

kElement = zeros(2);

phi1 = @(z) (-z+1)/2;
phi2 = @(z) (z+1)/2;

philprime = @(z) -1/2;
phi2prime = @(z) 1/2;

jacobian = @(z,xi,xipl) 0.5 * (xipl-xi);

[gaussPoints,gaussWeights] = getGauss();

%check our mapping
% x_i
% x_iPlus1
% z2xDomain(gaussPoints,x_i,x_iPlus1)
% max(z2xDomain(gaussPoints,x_i,x_iPlus1))
% min(z2xDomain(gaussPoints,x_i,x_iPlus1))
% pause

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

        %ok so this is tricky. Get the left function first
        if(row == 1)
            left = philprime;
            leftNorm = phil;
        elseif (row == 2)
            left = phi2prime;
            leftNorm = phi2;
        end

        if(col == 1)
            right = philprime;
            rightNorm = phil;
        elseif (col == 2)
            right = phi2prime;
            rightNorm = phi2;
        end

        % multiply this all out
        completeProductD = DinX(z2xDomain(gaussPoints,x_i,x_iPlus1)) ...
            .* left(gaussPoints) .* right(gaussPoints) ...
            ./ jacobian(gaussPoints, x_i, x_iPlus1);

        integralD = completeProductD' * gaussWeights;

        completeProductT = ((1/deltaT) + ...
            TauInX(z2xDomain(gaussPoints,x_i,x_iPlus1))) ...
            .* leftNorm(gaussPoints) ...
            .* rightNorm(gaussPoints) .* ...
            jacobian(gaussPoints,x_i,x_iPlus1);

        integralT = completeProductT' * gaussWeights;

        %
        % integralD
        % integralT
        % integralT + integralD
        % pause

        %this belongs here now
        kElement(row,col) = integralD + integralT;

    end
end
% kElement
% pause

end

function [ tauVector ] = TauInX( xVector )
%DINX Summary of this function goes here
% Detailed explanation goes here

result = zeros(size(xVector));

for(i = 1:length(result))
    xPos = xVector(i);

    if(xPos <= 0.1)

```

```

        rightNorm = phi1;
    elseif (col == 2)
        rightNorm = phi2;
    end

    % multiply this all out
    completeProductR = (1/deltaT) ...
        .* leftNorm(gaussPoints) ...
        .* rightNorm(gaussPoints) .* ...
        jacobian(gaussPoints,x_i,x_iPlus1);
    integralR = completeProductR' * gaussWeights;

    %this belongs here now
    rElement(row,col) = integralR;

end
end

% rElement
% pause

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;
% 0.000000000000000000000000000000;
% 0.5384693101056830910363144;
% 0.9061798459386639927976269];
%
% gaussWeights = [0.2369268850561890875142640;
% 0.4786286704993664680412915;
% 0.568888888888888888888888888889;
% 0.4786286704993664680412915;
% 0.2369268850561890875142640];

end

```

```

        TauVal = 0.7;
    elseif(xPos <= 0.2)
        TauVal = 2.0;
    elseif(xPos <= 0.3)
        TauVal = 1.5;
    elseif(xPos <= 0.4)
        TauVal = 2.25;
    elseif(xPos <= 0.5)
        TauVal = 1.55;
    elseif(xPos <= 0.6)
        TauVal = 0.85;
    elseif(xPos <= 0.7)
        TauVal = 2.25;
    elseif(xPos <= 0.8)
        TauVal = 1.95;
    elseif(xPos <= 0.9)
        TauVal = 0.25;
    else
        TauVal = 2.95;
    end

    result(i) = TauVal * 10^-3;
    % result(i) = 0;
end

tauVector = result;

end

```