Linh Tang

CEE/MAE M20

UID: 205542275

Nov 22, 2020

# HOMEWORK 7

## 1. Euler-Bernoulli Beam Bending

### 1.1 Introduction

The goal of this problem is to calculate and show the displacement of a nitinol alloy beam when a force applies to a single point on the beam by using a discretizing governing equation. This script outputs the maximum displacement, where it occurs, and the error in the maximum displacement with different number of discretization points. Besides, the output also includes the range of x-position which always contains the maximum displacement.

### 1.2 Model and methods

In this script, the length of the bar L and the applied force P are set up as global variables. At first, a Moment function is created to calculate the moment at various x- positions when the force P applies on a specific location of the beam. The function takes x-position and d - the location the force applies on as inputs, and then returns the moment result. This function performs moment calculation based on the following equation:

$$M(x) = \begin{cases} -\frac{P(L-d)x}{L} & \text{for } 0 \le x \le d \\ -\frac{Pd(L-x)}{L} & \text{for } d < x \le L. \end{cases}$$

```
function m = Moment(x, d)
global L P;
    if (x >= 0 && x <= d)
        m = -P*(L-d)*x/L ;
    else
        m = -P*d*(L-x)/L;
    end
End
```

At the beginning of the script, constants and variables are defined and initialized based on given information.

```
L = 1; % (m) length of the bar
P = 1000; % (N) the force
d = 0.8; % (m) where the force is applied
% outer and inner radius of the cross section
ro = 0.1;
ri = 0.06;
E = 6.5e10; %(Nm^-2) modulus of the elasticity
N = 100; % Number of discretization points
```

To prepare for the governing equation, the moment of inertia I and the even distance between every two points dx are calculated as:

```
I = pi*(ro^4 - ri^4)/4;
dx = L/(N-1);
```

Two arrays are set up; one is for x-positions and one is for moment results.

```
x = linspace(0, L, N); % array for x position
M = zeros(N, 1); % moment array
```

After that, a for loop is used to get all the moment results updated to the M array by calling the Moment function.

```
for k = 1:N
    M(k) = Moment(x(k), d);
end
```

To get the y displacement, the governing equation needs to be solved.

$$EI\frac{d^2y}{dx^2} \approx EI\frac{y_{k+1} - 2y_k + y_{k-1}}{\Delta x^2} = M(x)$$

$$\Rightarrow y_{k+1} - 2y_k + y_{k-1} = \Delta x^2 M(x)/EI$$

=> A. y = b

=> y = A\b

To apply this into the problem, a system matrix A and a vector b are constructed; basically, A is the matrix containing the coefficient of the left hand side and b is a vector for the right hand side. There is no calculation at two ends of the beam x = 0 and x = L; y is set equal to 0 at those points.

```
A = zeros (N, N); % A matrix of the system
b = zeros(N, 1); % right hand side vector
A(1, 1) = 1;
A(N, N) = 1;
b(1) = 0;
b(N) = 0;
```

```
for i = 2:N-1
    A(i,i-1:i+1) = [1, -2, 1];
    b(i) = dx^2*M(i)/EI;
end
y = A\b;
```

Based on the direction, all the y values are on the positive y of the plot. The displacement is the absolute value of y. In this case, a max function is used to find the maximum y; it is also the maximum displacement. The max function also allows us to get the index where that y is obtained; this index is used to get the x-position.

```
[max_dis, index] = max(y);
fprintf('** Number of discretization points: %d points. \n', N);
fprintf('- The maximum displacement of the beam is %d m. \n',
abs(max_dis));
fprintf('- It occurs at %.4f m. \n', x(index));
```

The theoretical value for maximum y displacement is calculated using the given formula in the problem statement.

```
c = min(d, L-d);
theo_y = P*c*(L^2 - c^2)^1.5/(9*sqrt(3)*EI*L);
error = abs(max_dis - theo_y)/theo_y*100 ;
fprintf('- The error is %d .\n',error);
```

The tic toc command is used to display the elapsed time in the command window.

The plot of y displacement as a function of x- position along the beam is obtained by using the following codes:

```
figure(1)
p = plot(x,y,'o-');
title('The displacement y over x- position')
xlabel ('x(m)');
ylabel ('y(m)');
set(gcf, 'Position', [100 40 800 500]);
set(gca, 'LineWidth', 2, 'FontSize', 12);
ax = gca;
ax.XGrid = 'off';
ax.YGrid = 'on';
saveas(p,'hw7_p1.png');
```

To find the range of x which always contains maximum y in part e, the calculation is similar with what we did above, but this calculation is done with the range of d, from 0.1 to 0.9 m (no calculation for two ends of the bar). In this part, the differences are setting up an array for d, a for loop going through that array to perform the calculation, then pasting all the obtained x - values into the array for x. Using min() and max() functions for the array x to get the range of x.

```
for n = 1:length(array_d)
    for k = 1:N
        moment(k) = Moment(x(k), array_d(n));
    end
      for i = 2:N-1
        b(i) = dx^2*moment(i)/EI;
    end
    y = A\b;
    [max_dis, index] = max(y);
```

```
    array_x(n) = x(index);% Storing the x where it has y max.
end

fprintf ('\n\n *Part e: The range of x that contains the maximum
displacement is:\n');

fprintf ('[%.4f, %.4f]. \n', min(array_x), max(array_x));
```

## 1.3 Results

- Results for 100 nodes

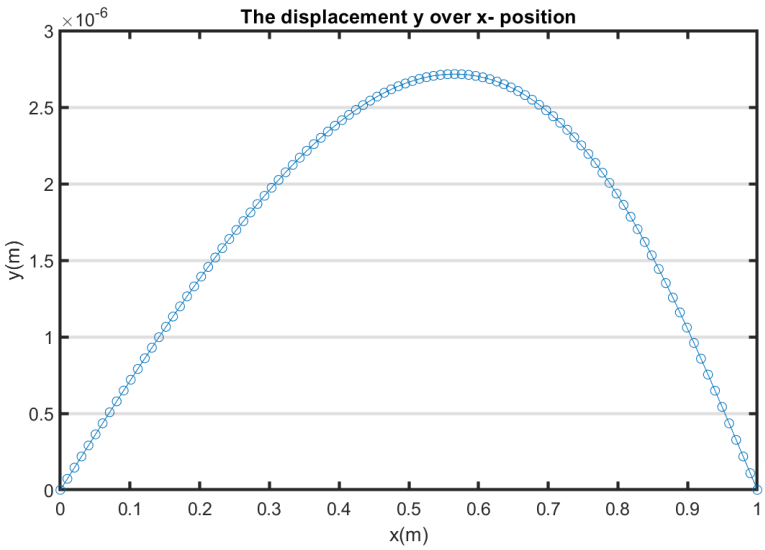


Figure 1. The plot of y displacement as a function of x- position along the beam for 100 nodes

- The range of x-position which always contains the maximum displacement is

  [0.4242m, 0.5758m].

```
Command Window
** Number of discretization points: 100 points.
- The maximum displacement of the beam is 2.7161e-06 m.
- It occurs at 0.5657 m.
- The error is 0.0084 %.
Elapsed time is 0.018835 seconds.


 *Part e: The range of x that contains the maximum displacement is:
[0.4242, 0.5758].
fx >>
```

- Plot for 10 nodes

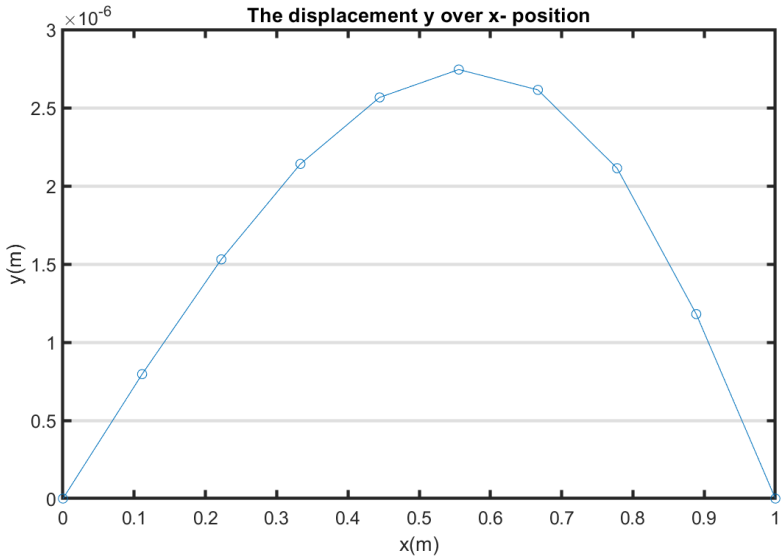


Figure 1. The plot of y displacement as a function of x- position along the beam for 10 nodes.

- Plot for 5 nodes

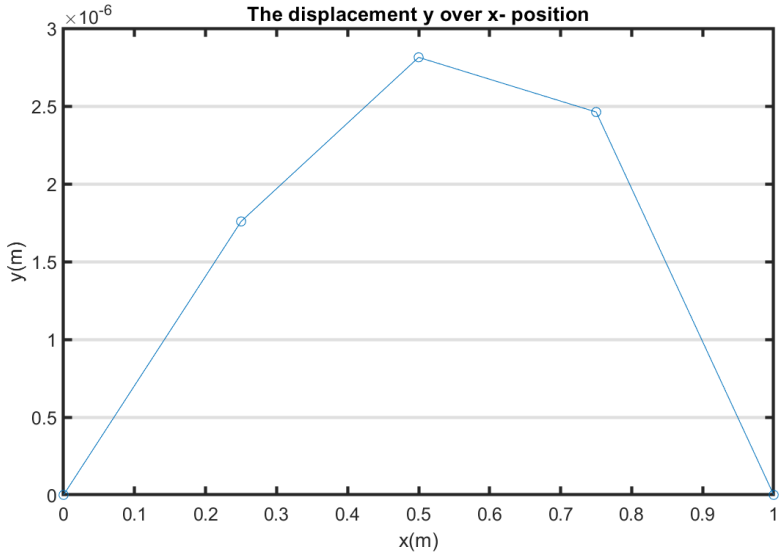


Figure 3. The plot of y displacement as a function of x- position along the beam for 5 nodes

| | 5 nodes | 10 nodes | 100 nodes |
|---|---|---|---|
| Maximum displacement (m) | 2.8131e-06 | 2.7441e-06 | 2.7161e-06 |
| x- position (m) | 0.5000 | 0.5556 | 0.5657 |
| Error (%) | 3.5801 | 1.0383 | 0.0084 |
| Elapsed time (s) | 0.016181 | 0.017526 | 0.018835 |

Table 1. The results for 5 nodes, 10 nodes and 100 nodes.

1.4 Discussion

Based on three figures corresponding to 5 nodes, 10, nodes and 100 nodes, the more discretization points, the smoother curve is obtained. According to the table 1, 100 nodes simulation gives us the most accurate maximum displacement when the error is only 0.0084%; this maximum displacement is the closest one to the theoretical value. Compared to the other two simulations, this error is much less than the error of 10 nodes simulation which is 1.0383% and the error of 5 nodes simulation which is 3.5801%. By looking at this, the greater number of discretization points,the approximate value gets closer to the theoretical value. Also from the table 1, 5 nodes simulation takes least amount of time to run the program; the elapsed time is 0.016181s which is less than the elapsed time of 10 nodes (0.017526s) and the elapsed time of 100 nodes (0.018835s). This makes sense since there are less simulations needed to be performed with less number of nodes. For 5 nodes, the program is done after going through 5 simulations instead of 100 simulations for 100 nodes. Besides, the noticeable point while writing this program is that there is no deflection at two ends of the bar and the direction that we choose for the system.

## 2. Langton's ant

### 2.1 Introduction

This problem is  to simulate a Langton's ant model; the ant's initial position and movement follows a specific rule. An animation is created to show the result in 1000 timesteps; the ant flips and changes the grid color as it moves. Besides the animation, a portion of black grids as a function of time is plotted in another figure. This problem also uses the periodic boundary condition to decide the movement of the ant when it is on the edges of the  2D grid.

### 2.2 Model and methods

In this script, the size of 2D grids includes number of rows and number of columns are set as global variables; a 2D array for the grid is created and initialized based on the given size. Another constant is defined here is the total number of steps for the simulation. After that, the initial position of the ant is defined by generating a random position in the grid; the randi() function is used to generate a random column and random row from 1 to 50. There are four directions that the head points to; it is helpful if each direction is represented by a value; north is one, east is two, south is three and west is fourth.  Since the ant's head initially points to North, this variable is initialized as 1.

```
global num_rows num_cols
num_rows = 50;
num_cols = 50;
square_color = zeros(num_rows, num_cols);
% number of simulations
```

```
max_steps = 1000;
% Define the initial position of t
ant_row = randi(50);
ant_col = randi(50);
% Ant's head initially points to North
head = 1;
```

After that, two functions are created to manage the ant's behaviors; one is used to flip the ant, and the other one is moving the ant forward. Both of them are put at the bottom of the main script. The ant_flip function takes the 2D grid, current positions of the ant (curr_row, curr_col) and the head direction (head) as input; then it returns the new direction (new_head) after flipping the ant. By setting number 1-4 corresponding to north, east, south, west respectively, the value of direction increases by 1 when flipping clockwise 90 degrees. The value of direction decreases by 1 when flipping counter clockwise 90 degrees.

When the ant is in white square, the ant flips clockwise; this means the value of direction will be added by 1. Similarly, the value of direction will be subtracted by after flipping counter clockwise when the ant is in black square. To keep the value of direction always in the range 1-4, the value is reset when it is out of range by adding or subtracting by 4.

```
function [new_head] = ant_flip (grid, curr_row, curr_col, head)
  if(grid(curr_row, curr_col) == 0)
        new_head = head + 1; %white square (0), flip clockwise 90
degree
  else
        new_head = head - 1; %black square(1), flip counter-clockwise
90 degree
  end
```

```
  % keep the head direction in the range 1-4
    if new_head > 4
        new_head = new_head - 4;
    elseif new_head < 1
        new_head = new_head + 4;
    end
end
```

The second function is the ant_move() function; this function moves the ant forward by 1 square based on the head direction. Therefore, the function takes the current position for the ant (curr_row, curr_col) and the head direction (head) as inputs, then returns the next position of the ant (ant_row, ant_col). When moving toward North or South, only the row changes by subtracting or adding by 1. Similarly, only the column is added or subtracted by 1 when the ant moves toward East or West.

On the other hand, the periodic boundary is used to move the ant when it is on the edges (first row, last row, first column, last column). When the ant is at the first row and wants to move North, it moves to the last row. When the ant is at the last row and wants to move South, it moves to the first row. When the ant is at the first column and wants to move West, it moves to the last column. When the ant is at the last column and wants to move East, it moves to the first column.

```
function [ant_row, ant_col] = ant_move (curr_row, curr_col, head)
global num_rows num_cols
    % Using periodic boundary
    if head == 1
        % Move North
```

```
    ant_row = curr_row - 1;
    ant_col = curr_col;
    % The ant is at the 1st row, wants to move North
    if ant_row < 1
        ant_row = num_rows; % Move to the last row
    end
elseif head == 2
    % Move East
    ant_col = curr_col + 1;
    ant_row = curr_row;
    % The ant is at the last col, wants to move East
    if ant_col > num_cols
        ant_col = 1; % Move to the 1st col
    end
elseif head ==  3
    % Move South
    ant_row = curr_row + 1;
    ant_col = curr_col;

    % The ant is at the last row, wants to move South
    if ant_row > num_rows
       ant_row = 1; % Move to the 1st col
    end
elseif head == 4
    %Move West
    ant_col = curr_col - 1;
    ant_row = curr_row;
    % The ant is at the 1st col, wants to move West
```

```
            if ant_col < 1
                ant_col = num_cols; % Move to the last col
            end
        end
    end
```

Back to the first part of the main script after finishing two functions, a for loop is used to simulate the model in 1000 time steps. In this for loop, the order of commands is flipping the ant, changing the color of the current square, moving the ant to the next position, and calculating the portion of black squares.

For flipping and moving the ant, two corresponding functions are called in the for loop as following:

```
    %% Flip the ant
        head = ant_flip(square_color, ant_row, ant_col, head);
    %% Move and check boundary
        [ant_row, ant_col] = ant_move(ant_row, ant_col, head);
```

For changing the color of the current square, black is changed to white, and vice versa.

```
      if(square_color(ant_row, ant_col) == 0)
            % White to black
            square_color(ant_row, ant_col) = 1;
        else
            % Black to white
            square_color(ant_row, ant_col) = 0;
        end
```

To calculate the portion of black squares in the grid, an array holding the results is initialized outside the for loop.

```
portion = zeros(max_steps, 1);
```

In the for loop, a counter variable is set up at each time step. After that, a double for loop is used to go through all squares in the grid, and count the black squares.

```
counter = 0;
for i = 1: num_rows
        for j = 1: num_cols
            if (square_color(i,j) == 1)
                counter = counter + 1;
            end

        end
    end
    % Calculate the portion
    portion(k) = counter/ (num_rows * num_cols);
```

To create an animation of the model, imagesc() is used, but the default colormap is parula. That is why we need to create our own black- white colormap for this system. The following code also includes command lines for creating a figure for animation and getting a video.

```
    figure(1)
    imagesc(square_color); % Create animation
    % Create a black and white colormap: 1 1 1 is white, 0 0 0 is black
    mymap = [1 1 1
    0 0 0];
    colormap(mymap);
    box on
    title(sprintf('Timesteps %d.', k));
    set(gcf, 'Position', [300 50 500 500]);
```

```
    set(gca, 'LineWidth', 2, 'FontSize', 10);

    drawnow();

    frame = getframe(gcf); % get the content of the current frame of
the fig

    writeVideo(v, frame); % save this frame into the videoWrite.
```

For the portion plot, this plot is designed to appear after the first animation figure, thus this is performed after the for loop is done.

```
figure(2)

p = plot(1:max_steps, portion, 'r');

title('The portion so black grids')

xlabel ('Time steps');

ylabel ('Black portion');

set(gcf, 'Position', [300 50 500 500]);

set(gca, 'LineWidth', 2, 'FontSize', 10);

ax = gca;

ax.XGrid = 'off';

ax.YGrid = 'on';

saveas(p, 'hw7_p2.png');
```

2.3 Results

- The Youtube link for the animation (this shows the model works well at the boundaries)

  https://youtu.be/Yd6MvMBaPcM

- The plot of black square portion:

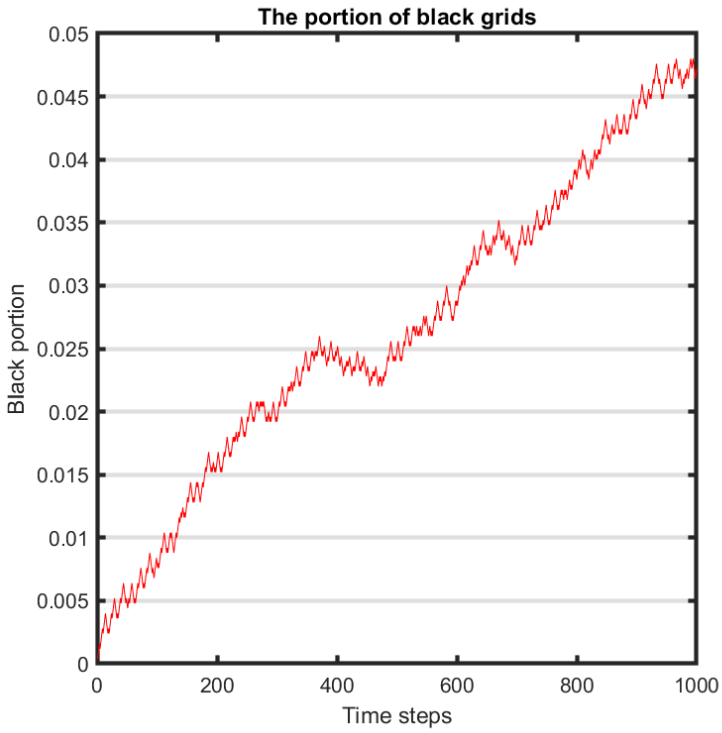


Figure 1. The plot of black portion as a function of time

2.4 Discussion

In this problem, there were few things that I found challenging when working on this model. The first one was figuring out how to choose and assign a value for each direction that makes the algorithm easier to work on. The second thing is making a black- white colormap for this system; I ended up finding a good example on Mathworks. The last thing is figuring out the order of the codes; I might not get the correct result if the order messed up. In the future, this problem can be improved by live plotting the portion graph as running the simulation; we might use animatedline and addpoints() function in Matlab.