Name: Antonio Jimenez

EID: aoj268

Date: 5/14/22

Class: COE 311K 14405

Dr. Karl W. Schulz

Project 02

Introduction and motivation:

I was tasked with developing and implementing a numerical method to solve a partial differential equation for a simply supported beam under a uniformly distributed load. The equation is the deflection of the bar as a function of X position on the bar. The equation being solved:

$$\text{EI}\,\frac{\delta^2 y}{\delta x^2} = -\frac{q}{2}(x^2 - xL)$$

EI is the product of Young's modulus and cross-sectional moment of inertia, q is the magnitude of the distributed load (per unit length), and L is the length of the beam. I was given the boundary conditions y (x = 0) = 0 and y (x =L) = 0. To solve the differential equation numerically, I used a second order finite-difference approximation by discretizing the beam into equidistant points.

Task given requires the use of elements learned throughout the entire course, such as plotting, object-oriented programming, verification of implementation, and many numerical methods discussed in class. By the end of this task, I would have applied what I have learned in COE311K to solve a partial differential equation and be able to analyze my previous implementation and find a way to drastically increase its runtime performance.

Prep Work:

(A) To verify that my implementation and numerical method provide accurate results, I needed to compare it to the analytical solution. Below I list the steps on how I solved the partial differential equation analytically.

After dividing both sides by EI, we are left with:

$$(1) \qquad \frac{\delta^2 y}{\delta x^2} = -\frac{q}{2EI}(x^2 - xL)$$

To solve for y, I proceeded to integrate both sides of equation (1) twice with respect to x which resulted in:

$$(2) \quad y = -\frac{q}{24EI}(x^4 - 2Lx^3) + C_1 x + C_2$$

The only step remaining was to find the values for C1 and C2. The value of C2 was attained by plugging in the first boundary condition y (x = 0) = 0 into equation (2), from which we discovered that C2 is equal to zero.

$$(3) \quad y = -\frac{q}{24EI}(x^4 - 2Lx^3) + C_1 x$$

By plugging in the second boundary condition y (x = L) = 0 into equation (3) I was able to attain the value of C1:

$$C_1 = -\frac{qL^3}{24EI}$$

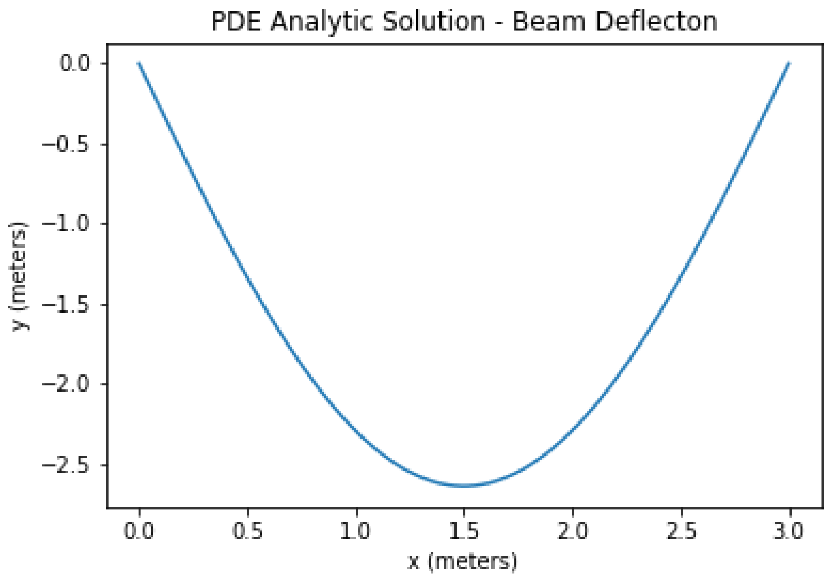
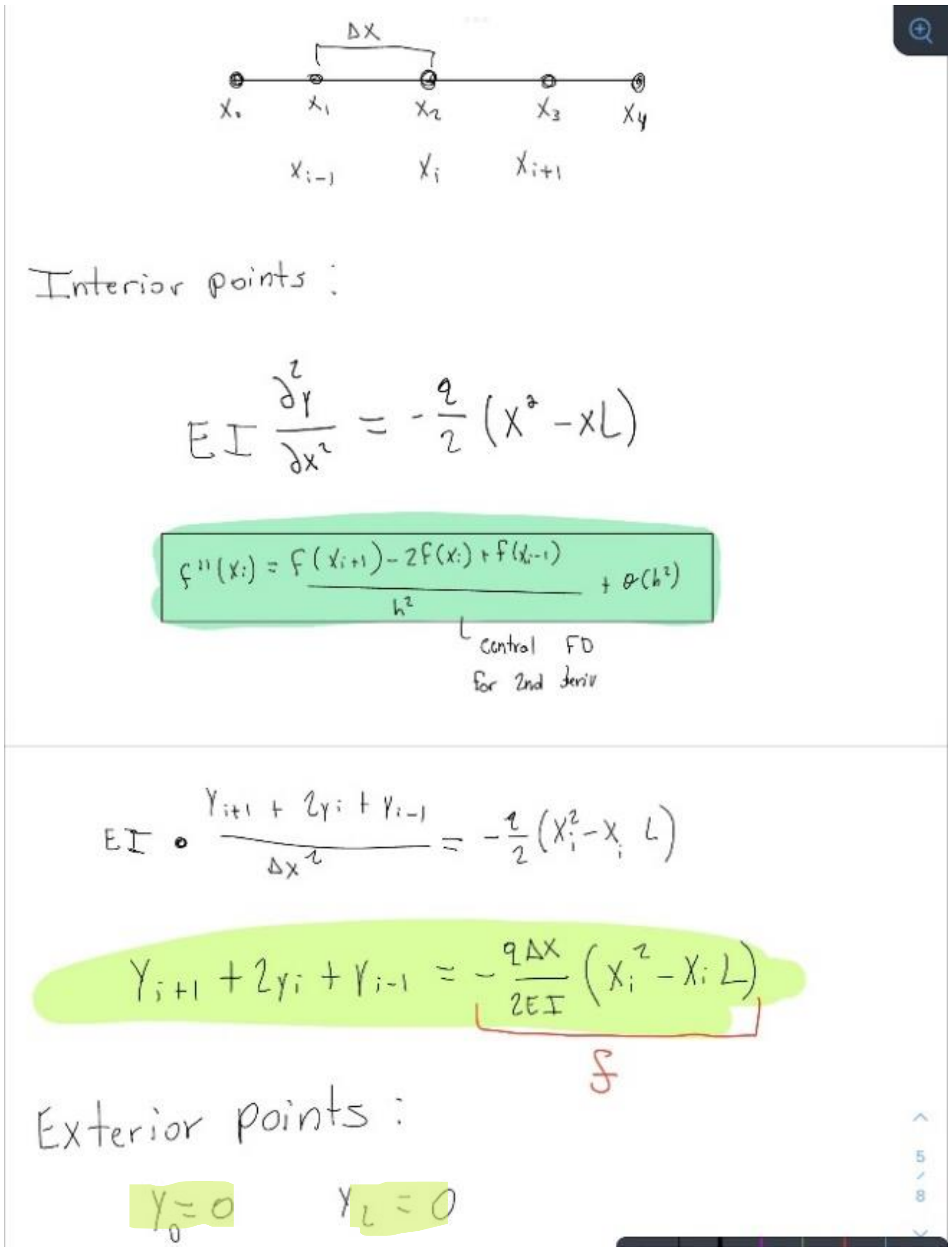


Figure 1: Analytic solution plotted on Matplotlib using n = 200 points; q = 1500 N/m; EI = 600Nm^2; L = 3m. Y values are associated with the magnitude of the beam deflection at position x along the bar.

(B) The following step was to derive our numerical method by considering coarse discretization with 5 points spanning the length of the bar. I wrote a discrete equation for calculating the deflection value at interior points and an equation for each of the deflection values at exterior points. In Figure 2, you will see that the equation for the interior points replaces $\frac{\delta^2 y}{\delta x^2}$ with its estimate attained by using central finite differencing, which has a truncation error of O(h^2). Central finite differencing requires a 3-point stencil, which is why I needed to create separate equations for the endpoints. Based on the boundary conditions we know that the deflection at the endpoints will always be zero. Using all the equations attained in the previous steps for 5 points, I created a 5x5 linear system of the form [K] {y} = {f} where [K] is the stiffness matrix, {y} is the deflection values that we are solving for and {f} is the force vector.



Interior points :

$$EI\,\frac{\partial^2 y}{\partial x^2} = -\frac{q}{2}\left(x^2 - xL\right)$$

$$\boxed{f''(x_i) = \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{h^2} + O(h^2)}$$

central FD for 2nd deriv

$$EI \cdot \frac{y_{i+1} + 2y_i + y_{i-1}}{\Delta x^2} = -\frac{q}{2}\left(x_i^2 - x_i L\right)$$

$$y_{i+1} + 2y_i + y_{i-1} = \underbrace{-\frac{q\Delta x}{2EI}\left(x_i^2 - x_i L\right)}_{f}$$

Exterior points :

$y_0 = 0$ $\quad$ $y_L = 0$

Figure 2: Handwritten work illustrating how equations for interior points and exterior points were attained. At the top of the image, you will see a 1-D domain corresponding to the beam with 5 points; 3 interior and two exterior points. Additionally, Δx seen in the 1-D domain is equal the mesh interval (h) seen in the central finite difference equation. Below you will see the equation for central finite difference and the equation for the interior and exterior points. Labeled f is the general right-hand side (force vector) for interior points.

5 x 5 linear system:

$$[K]\{y\} = \{f\}$$

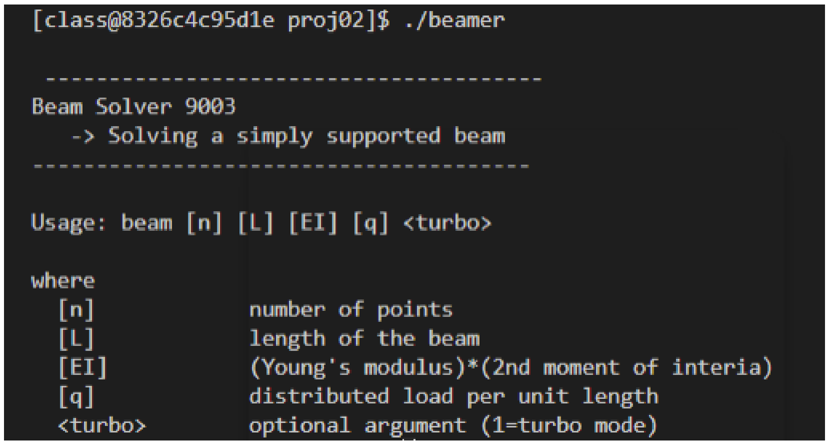
$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} 0 \\ -\frac{q\Delta x}{2EI}\left(x_1^2 - x_1 L\right) \\ -\frac{q\Delta x}{2EI}\left(x_2^2 - x_2 L\right) \\ -\frac{q\Delta x}{2EI}\left(x_3^2 - x_3 L\right) \\ 0 \end{bmatrix}$$

Figure 3: 5x5 linear system of the form [K]{y} ={f} where [K] is the stiffness matrix, {y} is the deflection values that we are solving for and {f} is the force vector. Created using equations seen in Figure 2.

Implementation:

To compile my code, you need to be in the proj02 repository and use the compilation line: “g++ -o beamer beamer.cpp beam.cpp matrix.cpp vector.cpp” in the terminal. All of the source code files required should be in the repository proj02, which include beamer.cpp, beam.cpp, beam.hpp, matrix.hpp, matrix.cpp, vector.cpp, vector.hpp, and timer.cpp. As seen through the compilation line, our tool named beamer leverages previously created vector and matrix class and contains the main() function. To run the code, you need either 5 or 6 arguments, as the turbo mode is optional. If turbo mode is not given, the code assumes you do not want to use it. If using the values n = 200 points, q = 1500 N/m, EI = 600Nm^2, L = 3m, you run the code by typing “./beamer 200 3 600 1500 1” into the terminal. In Figure 4, you see a helpful message that occurs when an inappropriate number of command-line arguments are given. In Figure 4, you also see what each of the arguments is supposed to represent and what order they are meant to be in.

Taking in these arguments, beamer.cpp creates a beam object that creates the stiff matrix, exact solution vector, force vector, and more based on the arguments inputted. When creating the stiff matrix, the solve tolerance for Gauss-Seidel is set to 1e-8. The matrix’s turbo mode is also set to whatever the user requested. Having all the components necessary such as the exact solution to compare to and calculate the l2 error, the mesh interval, and the arguments required for Gauss-Seidel, we are ready to call the beam class method getApproxSoln(), which calculates and returns the y vector which we are solving for. Having run the code with the appropriate arguments you will see the output shown in Figure 5.



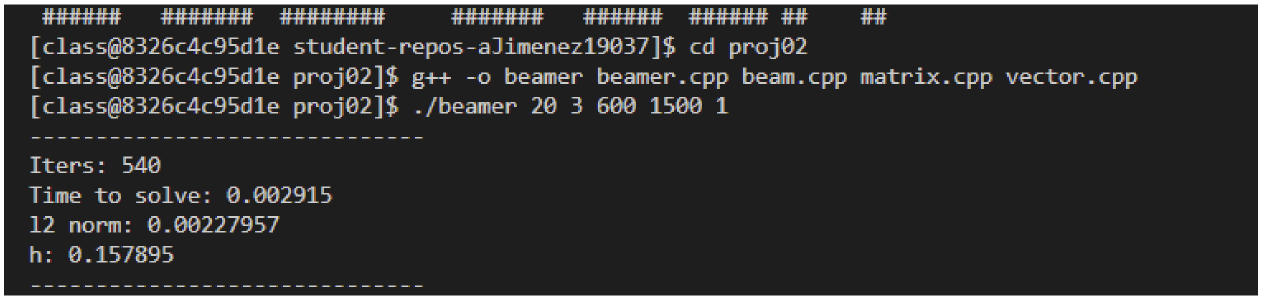
```
[class@8326c4c95d1e proj02]$ ./beamer

 --------------------------------------
Beam Solver 9003
   -> Solving a simply supported beam
--------------------------------------

Usage: beam [n] [L] [EI] [q] <turbo>

where
  [n]             number of points
  [L]             length of the beam
  [EI]            (Young's modulus)*(2nd moment of interia)
  [q]             distributed load per unit length
  <turbo>         optional argument (1=turbo mode)
```

Figure 4: Message given when code is run with inappropriate number of arguments. In this image we see in the first line that the code was attempted to be run with no arguments.



```
 ######   #######  ########     #######   ######  ###### ##    ##
[class@8326c4c95d1e student-repos-aJimenez19037]$ cd proj02
[class@8326c4c95d1e proj02]$ g++ -o beamer beamer.cpp beam.cpp matrix.cpp vector.cpp
[class@8326c4c95d1e proj02]$ ./beamer 20 3 600 1500 1
------------------------------
Iters: 540
Time to solve: 0.002915
l2 norm: 0.00227957
h: 0.157895
------------------------------
```

Figure 5: Example execution using n = 20; q = 1500 N/m; EI = 600Nm^2; L = 3m. First line shows how to access proj02 repository from within my class repository. Second line shows compilation line. Third line shows arguments given to run the code. Lastly, you see an example output that results when the appropriate arguments is given.

Analysis runs:

n	h	L2 Error	# of iterations	Wall-clock time (secs)
10	.333333	0.0101603	132	0.000654
20	.157895	0.00227957	540	0.006379
50	0.0612245	0.000340377	3143	0.158655
100	0.030303	7.40612e-05	11438	2.56387
200	0.0150754	1.9358e-5	40619	24.5928

Figure 4: Table of results from a varying number of points. As expected, an increase in the number of points (n) causes the mesh interval (h) to decrease, l2 error to decrease, and time to solve to increase. A converging l2 error also shows that our numerical method and implementation are functioning as our y vector is converging to the values of the exact solution.

Verification and Runtime Performance:

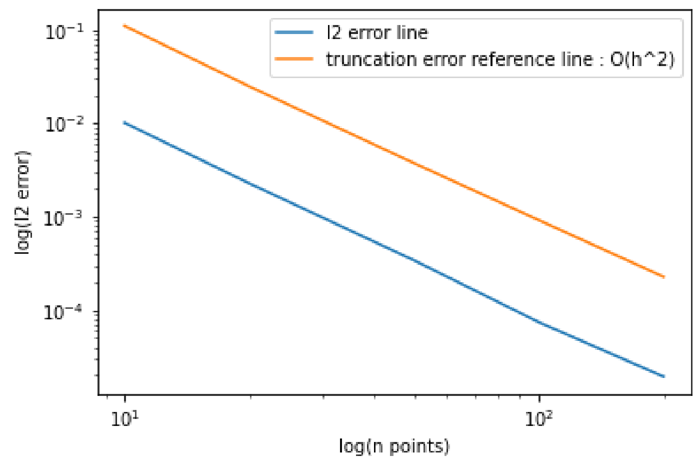


Figure 5: Matplotlib plot of l2 error as a function of n points plotted on a log/log scale with truncation error reference line demonstrating convergence rate we expect. Verifies code is working as expected, as a larger number of points will give us more a accurate, which is shows by the negative slope of the l2 error line.

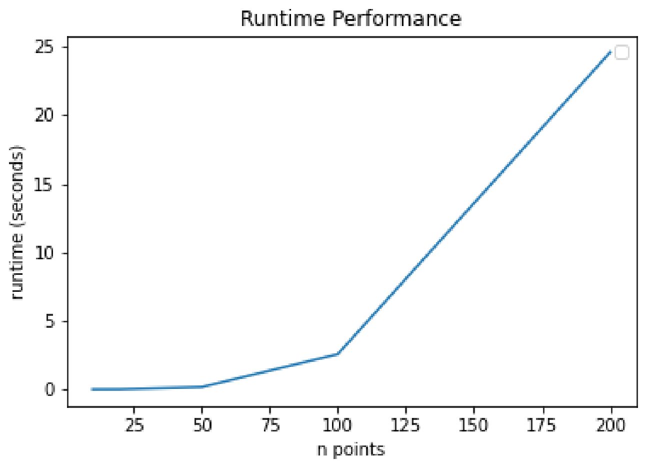


Figure 6: Matplotlib plot of runtime performance as a function of n points. Aligns with my expectations, as solving a linear system with gauss-seidel takes longer as the size of the matrix increases. Although the solving time for Gauss-Seidel can vary largely based on the composition of the matrix.

Performance modifications:

Intending to reduce the time of the n=200 case by 5x, I expanded upon my Gauss-Seidel implementation to cater to a matrix with a bandwidth of 3. The adjustments make the Gauss-Seidel method only loop through values that are necessary and are not zero. This drastically reduced the runtime performance, as a lot of time was being spent looping through and multiplying values of 0 within the matrix. Since we simply skipped the values equal to zero, our h, l2 error, and the number of iterations remained the same. Similar to what we saw before, time drastically increases as n increases. I also made a few strength reduction modifications which decreased the time for both the turbo and regular solver. The time reduction seen through strength reduction was minimal when compared to that caused by adjusting the Gauss-Seidel implementation to cater to a sparse matrix with a bandwidth of 3.

n	h	L2 Error	# of iterations	Wall-clock time (secs)
10	.333333	0.0101603	132	0.000453
20	.157895	0.00227957	540	0.002394
50	0.0612245	0.000340377	3143	0.026201
100	0.030303	7.40612e-05	11438	0.167685
200	0.0150754	1.9358e-5	40619	0.947584

Figure 7: Table illustrating new wall-clock time values using turbo mode. When comparing the table from Figure 4, we see that the wall-clock time of the n=200 case decreased by a factor of around 24x.

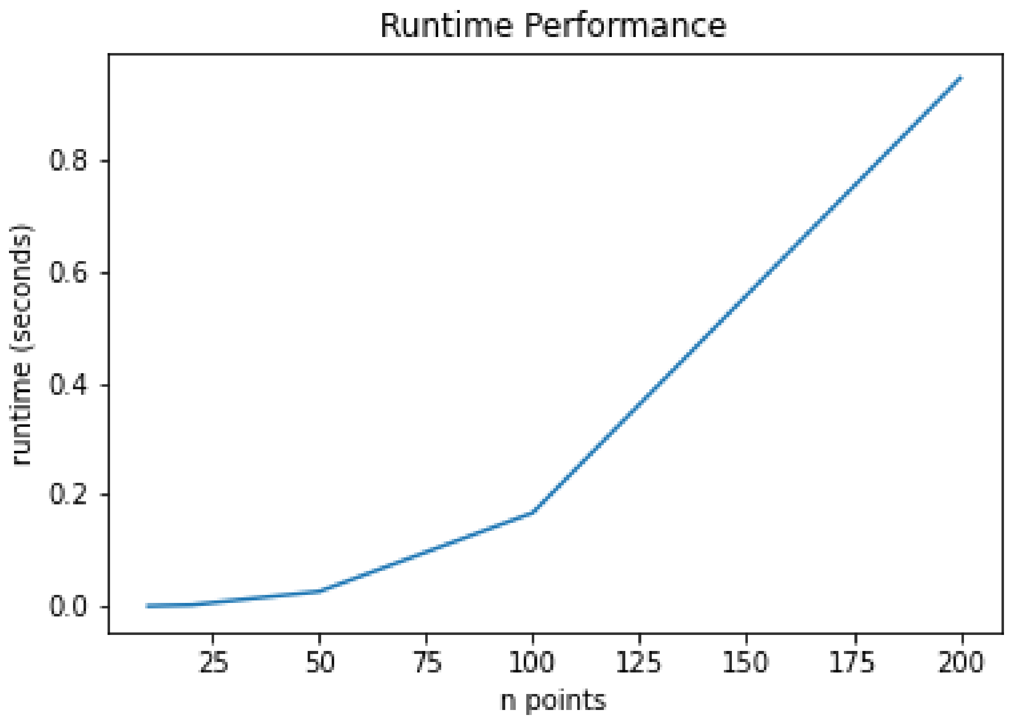


Figure 8: Matplotlib plot of runtime performance as a function of n points. The plot resembles the plot of the runtime performance without the use of turbo mode.