Shear Force and Bending Moment Diagram Calculator

Bachelor of Science in Mechanical Engineering (Expected June 2028)

Statics and Python

Ariaa-H

June 27, 2025

Repository Link:

https://github.com/Ariaa-H/Shear-Force-and-Bending-Moment-Diagram-Calculator

Contents

1. Abstract

This Jupyter Notebook–based Python tool performs automated structural analysis of a simply supported beam, applying principles of static equilibrium to generate shear force and bending moment diagrams. It supports multiple load types—point loads, point moments, uniformly distributed loads (UDL), and linearly varying loads (LDL)—allowing users to define beam span, support positions (pin at A, roller at B), and load configurations through intuitive inputs. The program calculates reactions at supports, computes internal shear and moment distributions via the superposition principle, and visualizes results using Matplotlib plots.

Keywords: Shear force, bending moment, uniformly distributed load (UDL), linearly varying load (LDL), supports

2. Introduction

When a beam supports external loads, it develops **internal forces** that resist deformation. By conceptually “cutting” the beam at any section, three key internal forces are revealed:

- **Axial (normal) force (N)**: acts along the length of the beam.
- **Shear force (V)**: perpendicular to the beam, resisting sliding of sections.
- **Bending moment (M)**: a moment that resists bending at the cut.

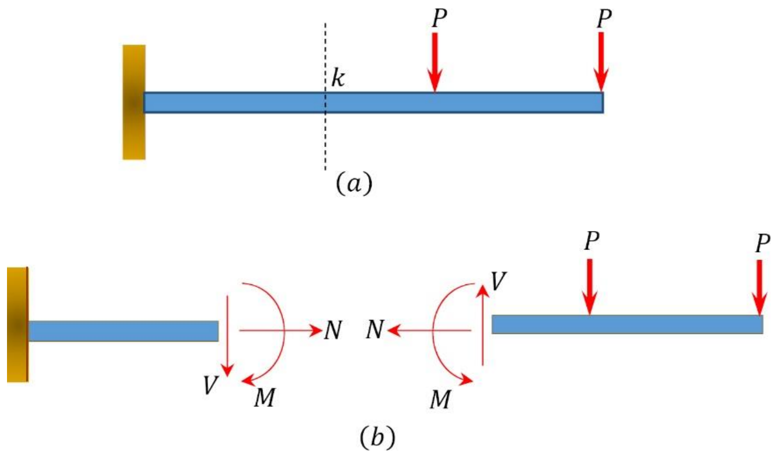


Figure 1: Internal forces in a beam. Libretexts (2023)

Plotting shear force and bending moment diagrams along the beam helps identify points of maximum stress, critical for safe design in structures such as bridges, frames, shafts, crane arms, and aerospace components.

In mechanical engineering, these diagrams inform material selection, optimize cross-sectional geometry, and validate computational models like Finite Element Analysis. However, manual calculation of these diagrams, especially under complex loading, is time-consuming and prone to error. Automating the process enhances efficiency, accuracy, and understanding.

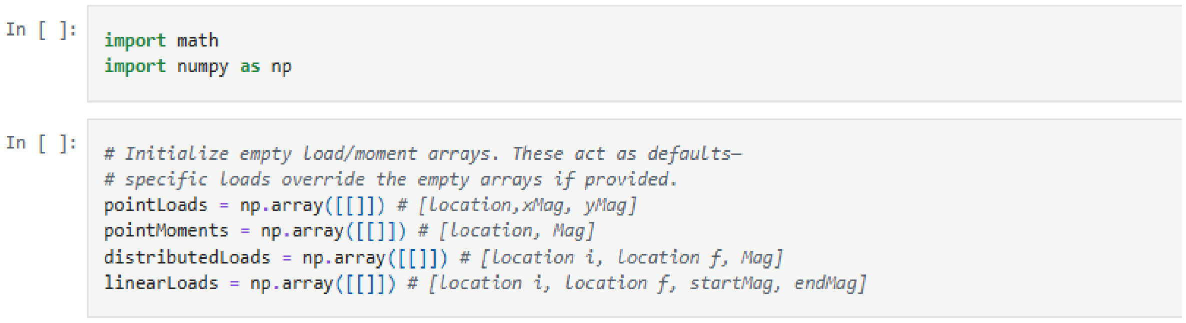
Inspired by Dr. Seán Carroll’s EngineeringSkills tutorial, this Jupyter Notebook–based Python tool allows users to define a simply supported beam (pin at A, roller at B), input various loads (point loads, point moments, UDLs, LDLs), calculate support reactions, and visualize internal force distributions using Matplotlib—all within an interactive environment (Carroll, n.d.).

3. Methodology

In this section, the step-by-step flow of the code is explained in detail:

3.1 Imports, Setup & User Input

The notebook begins by importing essential libraries: math for mathematical operations and numpy for efficient numerical array handling. Four empty arrays are initialized to later store user-defined loads:

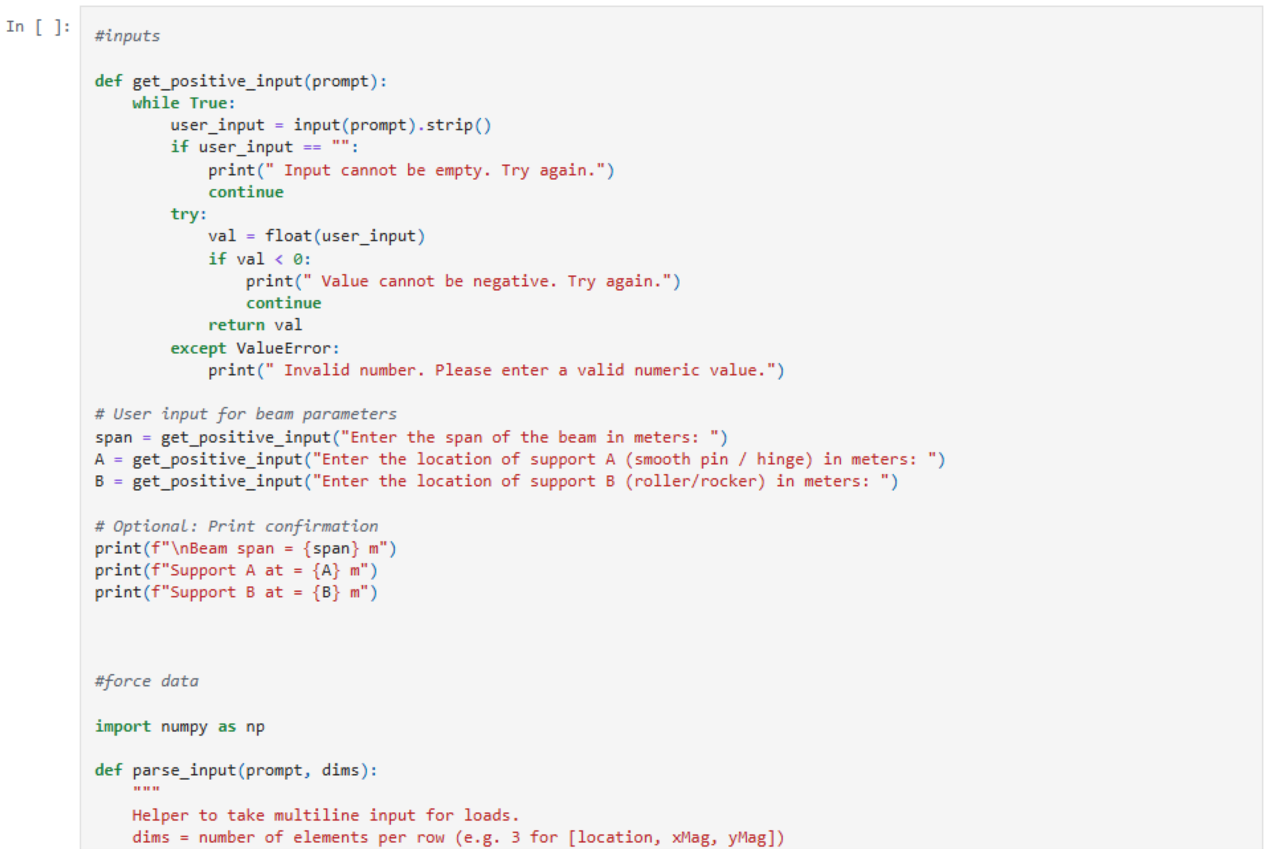


```
In [ ]:
import math
import numpy as np
```

```
In [ ]:
# Initialize empty load/moment arrays. These act as defaults—
# specific loads override the empty arrays if provided.
pointLoads = np.array([[]]) # [location,xMag, yMag]
pointMoments = np.array([[]]) # [location, Mag]
distributedLoads = np.array([[]]) # [location i, location f, Mag]
linearLoads = np.array([[]]) # [location i, location f, startMag, endMag]
```

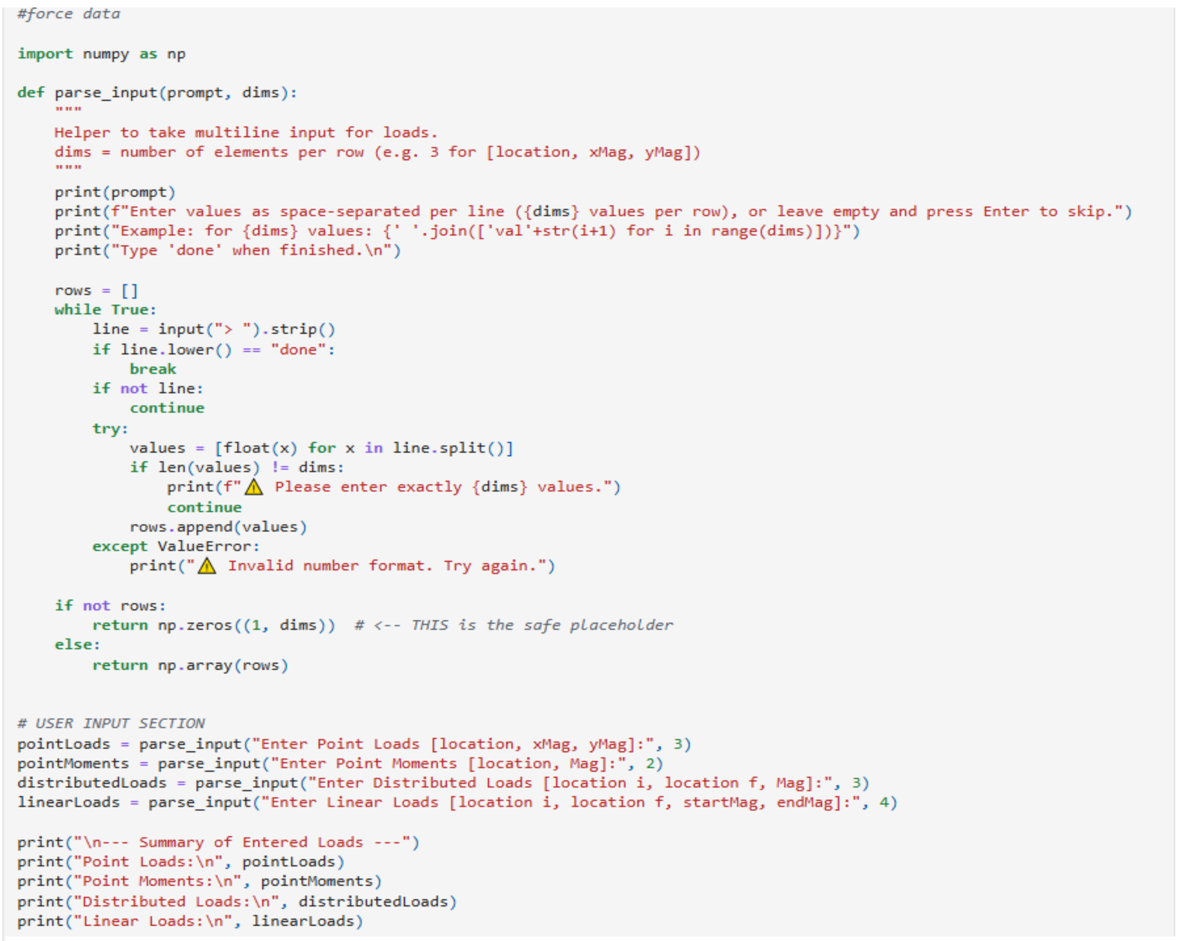
Figure 2

These serve as defaults until replaced by actual inputs. Then the user is prompted—using input validation—to define the **beam span** and support locations **A** (pin) and **B** (roller). This ensures valid, non-negative values. Next, the parse_input() routine captures multi-line entries for each load type. For example, point loads follow the format [location, xMag, yMag]. Each load type is parsed similarly into respective NumPy arrays, and a summary is printed upon completion.



```
In [ ]: #inputs

        def get_positive_input(prompt):
            while True:
                user_input = input(prompt).strip()
                if user_input == "":
                    print(" Input cannot be empty. Try again.")
                    continue
                try:
                    val = float(user_input)
                    if val < 0:
                        print(" Value cannot be negative. Try again.")
                        continue
                    return val
                except ValueError:
                    print(" Invalid number. Please enter a valid numeric value.")

        # User input for beam parameters
        span = get_positive_input("Enter the span of the beam in meters: ")
        A = get_positive_input("Enter the location of support A (smooth pin / hinge) in meters: ")
        B = get_positive_input("Enter the location of support B (roller/rocker) in meters: ")

        # Optional: Print confirmation
        print(f"\nBeam span = {span} m")
        print(f"Support A at = {A} m")
        print(f"Support B at = {B} m")



        #force data

        import numpy as np

        def parse_input(prompt, dims):
            """
            Helper to take multiline input for loads.
            dims = number of elements per row (e.g. 3 for [location, xMag, yMag])
```

Figure 3

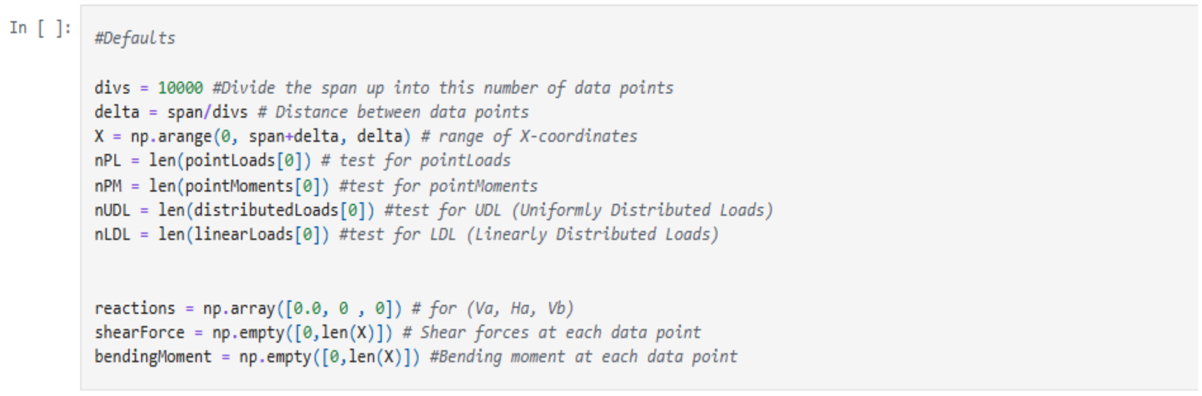


```
#force data

import numpy as np

def parse_input(prompt, dims):
    """
    Helper to take multiline input for loads.
    dims = number of elements per row (e.g. 3 for [location, xMag, yMag])
    """
    print(prompt)
    print(f"Enter values as space-separated per line ({dims} values per row), or leave empty and press Enter to skip.")
    print("Example: for {dims} values: {' '.join(['val'+str(i+1) for i in range(dims)])}")
    print("Type 'done' when finished.\n")

    rows = []
    while True:
        line = input("> ").strip()
        if line.lower() == "done":
            break
        if not line:
            continue
        try:
            values = [float(x) for x in line.split()]
            if len(values) != dims:
                print(f"⚠ Please enter exactly {dims} values.")
                continue
            rows.append(values)
        except ValueError:
            print("⚠ Invalid number format. Try again.")

    if not rows:
        return np.zeros((1, dims))  # <-- THIS is the safe placeholder
    else:
        return np.array(rows)


# USER INPUT SECTION
pointLoads = parse_input("Enter Point Loads [location, xMag, yMag]:", 3)
pointMoments = parse_input("Enter Point Moments [location, Mag]:", 2)
distributedLoads = parse_input("Enter Distributed Loads [location i, location f, Mag]:", 3)
linearLoads = parse_input("Enter Linear Loads [location i, location f, startMag, endMag]:", 4)

print("\n--- Summary of Entered Loads ---")
print("Point Loads:\n", pointLoads)
print("Point Moments:\n", pointMoments)
print("Distributed Loads:\n", distributedLoads)
print("Linear Loads:\n", linearLoads)
```

Figure 4

3.2 Beam Discretization & Data Initialization

The beam is discretized into 10,000 segments using:



```
In [ ]:
#Defaults

divs = 10000 #Divide the span up into this number of data points
delta = span/divs # Distance between data points
X = np.arange(0, span+delta, delta) # range of X-coordinates
nPL = len(pointLoads[0]) # test for pointLoads
nPM = len(pointMoments[0]) #test for pointMoments
nUDL = len(distributedLoads[0]) #test for UDL (Uniformly Distributed Loads)
nLDL = len(linearLoads[0]) #test for LDL (Linearly Distributed Loads)


reactions = np.array([0.0, 0 , 0]) # for (Va, Ha, Vb)
shearForce = np.empty([0,len(X)]) # Shear forces at each data point
bendingMoment = np.empty([0,len(X)]) #Bending moment at each data point
```

Figure 5

This generates a grid of evenly spaced coordinates (X) along the beam, enabling accurate and continuous evaluation of shear and moment functions at each point.

Count variables (nPL, nPM, nUDL, nLDL) are used to check for the presence of point loads, point moments, uniform distributed loads (UDL), and linear distributed loads (LDL). These determine whether the corresponding calculations should run.

The "reactions" array collects total support forces ([Va, Ha, Vb]) across all loads. The shearForce and bendingMoment arrays start empty and are used to store individual diagrams for each load before they are summed. The use of np.empty ensures efficient memory allocation without initializing placeholder values.

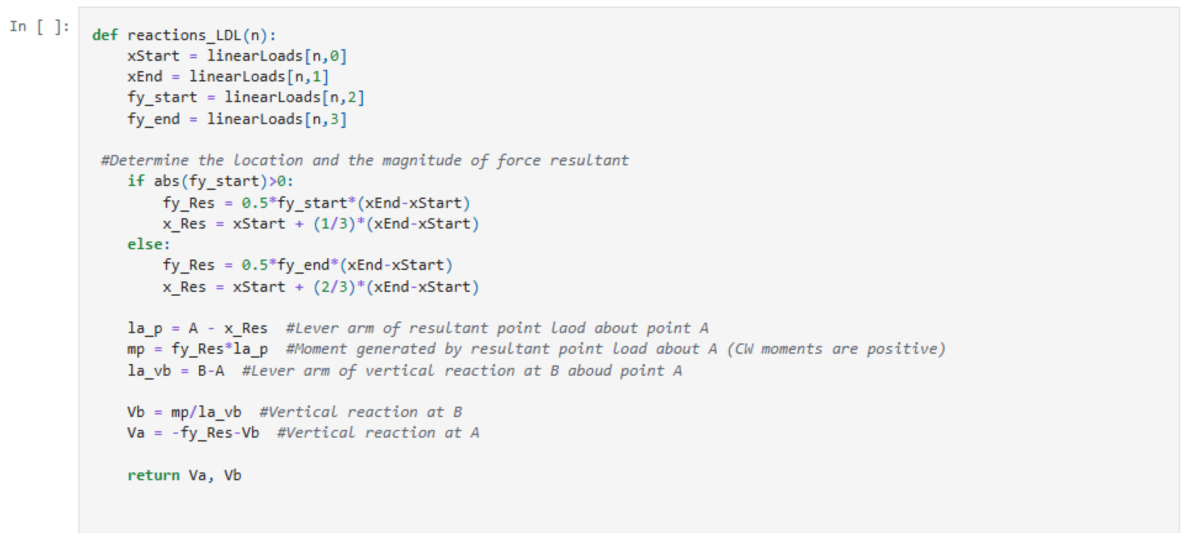
3.3 Support Reaction Calculations

For each load type, a dedicated reactions_X(n) function implements static equilibrium to compute support reactions:

- **Point loads** (reactions_PL(n)) read location and force, then use moments and force balance to calculate **Va**, **Ha**, and **Vb**.
- **Point moments** (reactions_PM(n)) solve for vertical reactions **Va** and **Vb** from a moment applied at a point.

- **UDL** (reactions_UDL(n)) converts uniform loads into an equivalent point force at the centroid, then calculates support impacts.
- **LDL** (reactions_LDL(n)) adapts to triangular loading, computes its resultant and lever arm, then solves for reactions.

Each function returns reactions, which are recorded in _record arrays and cumulatively added to reactions—a superposition that handles multiple loads seamlessly. An example for reactions_LDL(n) is shown below:



```
In [ ]: def reactions_LDL(n):
            xStart = linearLoads[n,0]
            xEnd = linearLoads[n,1]
            fy_start = linearLoads[n,2]
            fy_end = linearLoads[n,3]

         #Determine the location and the magnitude of force resultant
            if abs(fy_start)>0:
                fy_Res = 0.5*fy_start*(xEnd-xStart)
                x_Res = xStart + (1/3)*(xEnd-xStart)
            else:
                fy_Res = 0.5*fy_end*(xEnd-xStart)
                x_Res = xStart + (2/3)*(xEnd-xStart)

            la_p = A - x_Res  #Lever arm of resultant point laod about point A
            mp = fy_Res*la_p  #Moment generated by resultant point load about A (CW moments are positive)
            la_vb = B-A  #Lever arm of vertical reaction at B aboud point A

            Vb = mp/la_vb  #Vertical reaction at B
            Va = -fy_Res-Vb  #Vertical reaction at A

            return Va, Vb
```

Figure 6

3.4 Accumulating Per-Load Reactions

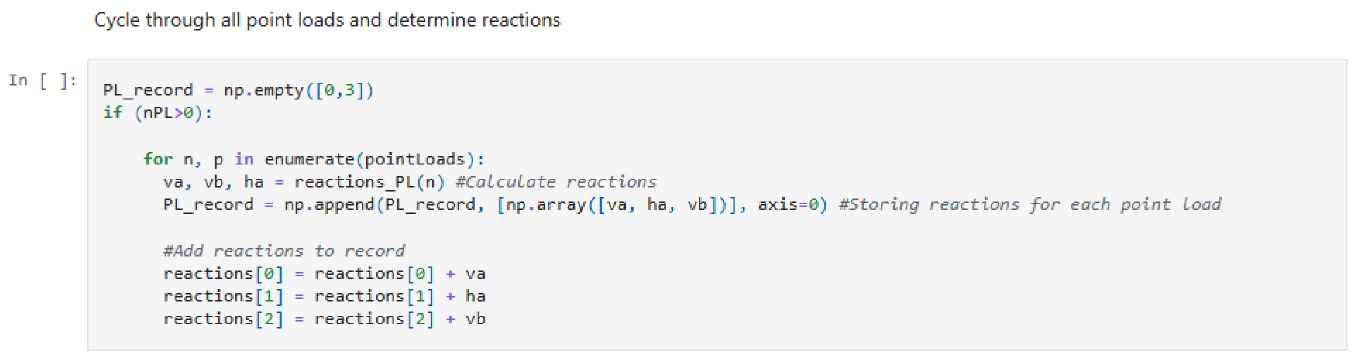
After defining reaction functions, the code prepares to apply them to actual loads. For instance, in the **point load** section, it begins by creating an empty NumPy array PL_record = np.empty([0, 3]), ready to store reactions for each point load. The if nPL > 0: check ensures that the block only runs if the user provided at least one point load.

Within the loop, each load is processed one by one:

- **Compute reactions**: reactions_PL(n) applies equilibrium equations to determine support forces (Va, Ha, Vb) for the nth point load.

- **Record individually**: The returned values are appended to PL_record, building a log of how each load contributes to overall reactions.
- **Update totals**: The same values are then added to the cumulative reactions array ([Va, Ha, Vb]), summing up contributions across all point loads.

This pattern—calculate, record, accumulate—is repeated for point moments, UDLs, and LDLs using their respective reaction functions (reactions_PM, reactions_UDL, reactions_LDL) and record arrays (PM_record, UDL_record, LDL_record).



Cycle through all point loads and determine reactions

```
In [ ]:
PL_record = np.empty([0,3])
if (nPL>0):

    for n, p in enumerate(pointLoads):
      va, vb, ha = reactions_PL(n) #Calculate reactions
      PL_record = np.append(PL_record, [np.array([va, ha, vb])], axis=0) #Storing reactions for each point load

      #Add reactions to record
      reactions[0] = reactions[0] + va
      reactions[1] = reactions[1] + ha
      reactions[2] = reactions[2] + vb
```

Figure 7

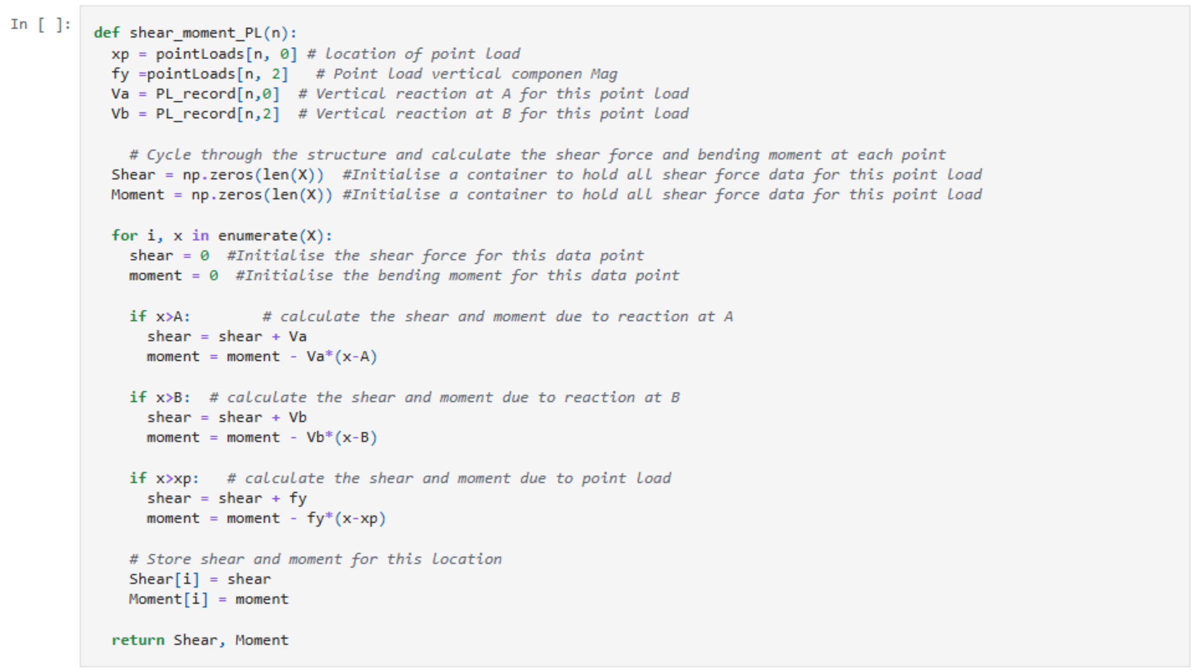
3.5 Shear and Moment Diagram Generation

Once reactions are known, the program computes internal distributions for each load through shear_moment_X(n) functions:

1. Initialize zero arrays, Shear and Moment, over the discretized positions X.
2. Loop through each position x:
    - Apply reaction contributions when x > A or x > B.
    - Add load-specific effects, such as:
        - A vertical jump for point loads.
        - A constant moment addition if the load is a point moment.

- A triangular shear variation for UDLs, or more complex shapes for LDLs.

3. Store these in the arrays, returning them per load.

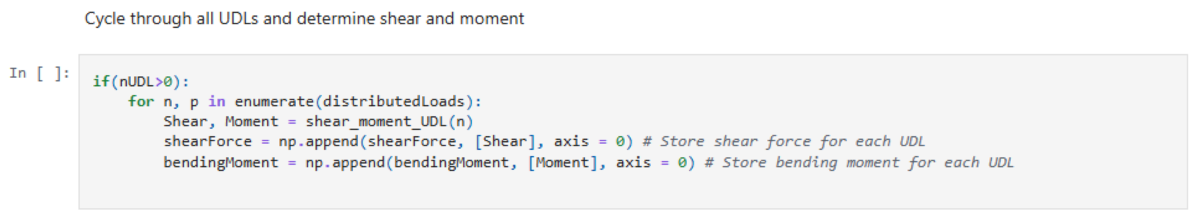


```
In [ ]: def shear_moment_PL(n):
          xp = pointLoads[n, 0] # Location of point load
          fy =pointLoads[n, 2]   # Point load vertical componen Mag
          Va = PL_record[n,0]  # Vertical reaction at A for this point load
          Vb = PL_record[n,2]  # Vertical reaction at B for this point load

            # Cycle through the structure and calculate the shear force and bending moment at each point
          Shear = np.zeros(len(X))  #Initialise a container to hold all shear force data for this point load
          Moment = np.zeros(len(X)) #Initialise a container to hold all shear force data for this point load

          for i, x in enumerate(X):
            shear = 0  #Initialise the shear force for this data point
            moment = 0  #Initialise the bending moment for this data point

            if x>A:        # calculate the shear and moment due to reaction at A
              shear = shear + Va
              moment = moment - Va*(x-A)

            if x>B:  # calculate the shear and moment due to reaction at B
              shear = shear + Vb
              moment = moment - Vb*(x-B)

            if x>xp:   # calculate the shear and moment due to point load
              shear = shear + fy
              moment = moment - fy*(x-xp)

            # Store shear and moment for this location
            Shear[i] = shear
            Moment[i] = moment

          return Shear, Moment
```

Figure 8

3.6 Superposition & Aggregation

For every load array, the corresponding shear and moment outputs are appended to global arrays:



Cycle through all UDLs and determine shear and moment

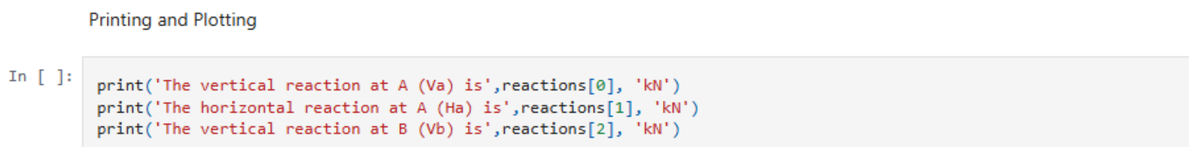
```
In [ ]: if(nUDL>0):
            for n, p in enumerate(distributedLoads):
                Shear, Moment = shear_moment_UDL(n)
                shearForce = np.append(shearForce, [Shear], axis = 0) # Store shear force for each UDL
                bendingMoment = np.append(bendingMoment, [Moment], axis = 0) # Store bending moment for each UDL
```

Figure 9

This creates row-by-row summaries where each row corresponds to one load's influence. The final diagrams are computed by summing across rows to reflect all loads simultaneously.

3.7 Output & Visualization

The cumulative reactions are printed:



Printing and Plotting

```
In [ ]: print('The vertical reaction at A (Va) is',reactions[0], 'kN')
        print('The horizontal reaction at A (Ha) is',reactions[1], 'kN')
        print('The vertical reaction at B (Vb) is',reactions[2], 'kN')
```

Figure 10

Finally, **Plotly** is used to create interactive diagrams:

- The shear force is plotted as a filled blue area.
- The bending moment is plotted as a filled green area.

The complete code, including the Plotly-based plotting steps, is available in the project's GitHub repository.

4. Sample Results

For testing purposes, the following beam and load data-taken from an example in Hibbeler's *Engineering Mechanics: Statics* (14th Edition), will be input into our project:

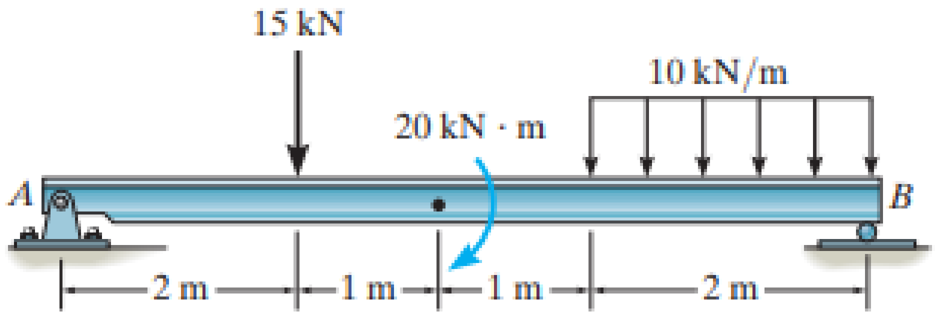


Figure 11: Problem 7-76 (Hibbeler, 2017)

As can be deduced from the picture, the span of the beam is 6 m, and the location of the smooth pin support A is 0 m, and roller support B is 6 m. Three loads are applied:

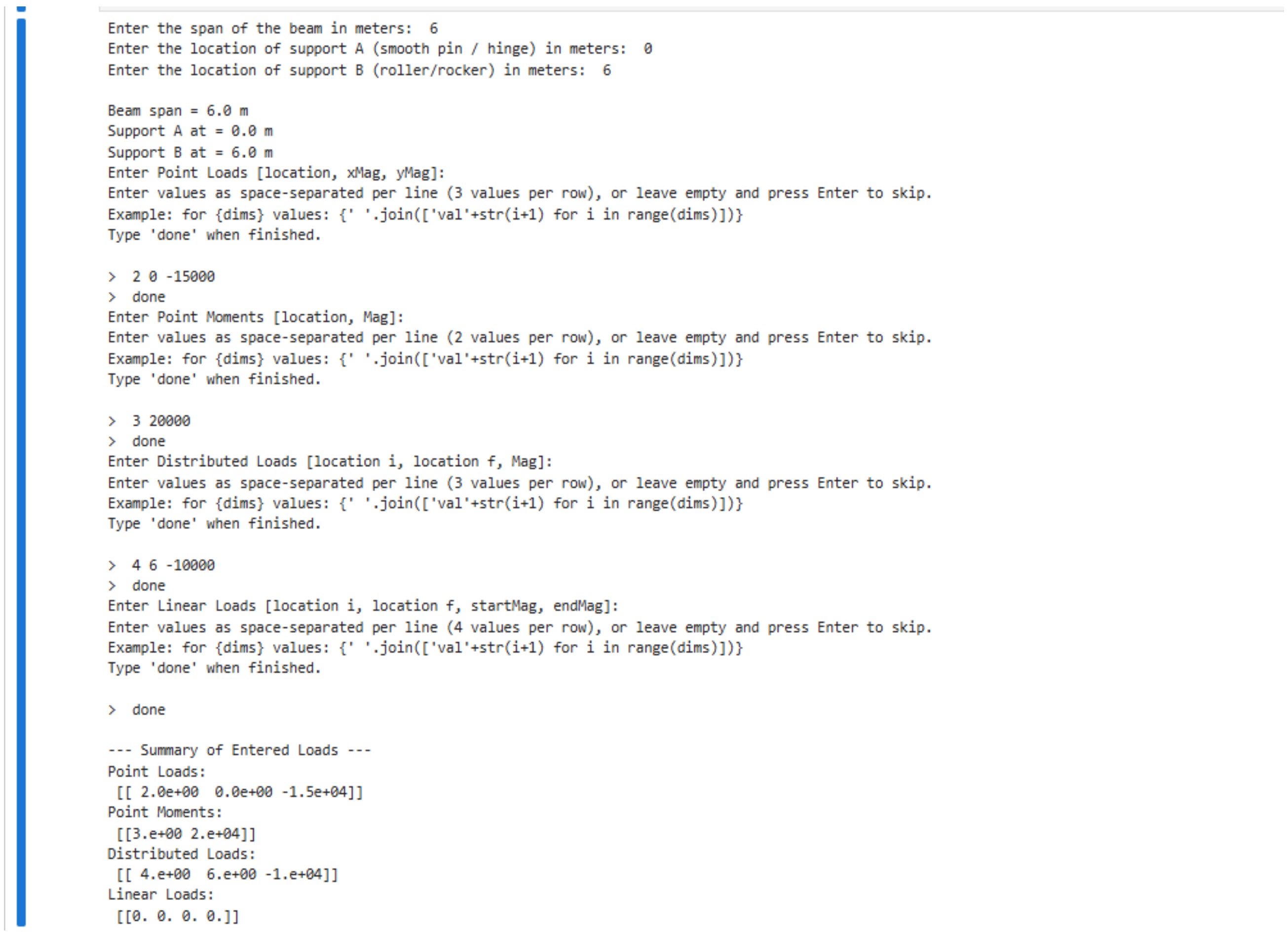
- A 15 kN downward point load at 2 m
- A 20 kN·m clockwise point moment at 3 m
- A 10 kN/m downward uniformly distributed load (UDL) from 4 m to 6 m

Hence, these are inserted in our project as arrays of:

```
pointLoads     = np.array([[2, 0, -15000]])

pointMoments   = np.array([[3, 20000]])

distributedLoads   = np.array([[4, 6, -10000]])
```

It is important to note that the user simply needs to enter the values and press space for switching to a different feature, e.g. from "location f" to "Mag". In other words, the user is not required to manually create arrays using NumPy, since the code automatically generates them, as explained in section 3.1.

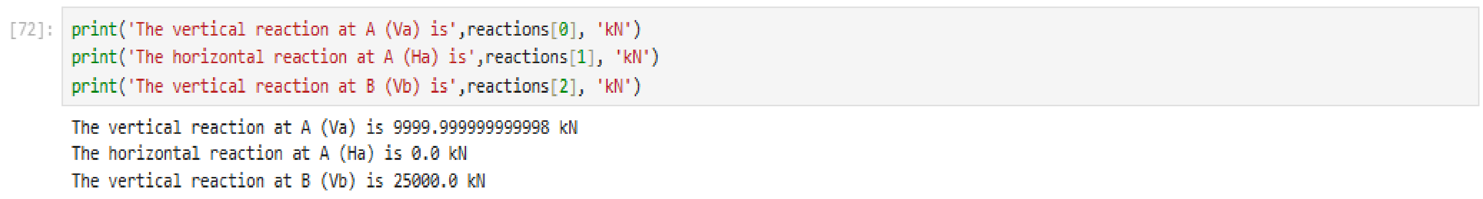
The mentioned data are inputted and printed:



```
Enter the span of the beam in meters:  6
Enter the location of support A (smooth pin / hinge) in meters:  0
Enter the location of support B (roller/rocker) in meters:  6

Beam span = 6.0 m
Support A at = 0.0 m
Support B at = 6.0 m
Enter Point Loads [location, xMag, yMag]:
Enter values as space-separated per line (3 values per row), or leave empty and press Enter to skip.
Example: for {dims} values: {' '.join(['val'+str(i+1) for i in range(dims)])}
Type 'done' when finished.

>  2 0 -15000
>  done
Enter Point Moments [location, Mag]:
Enter values as space-separated per line (2 values per row), or leave empty and press Enter to skip.
Example: for {dims} values: {' '.join(['val'+str(i+1) for i in range(dims)])}
Type 'done' when finished.

>  3 20000
>  done
Enter Distributed Loads [location i, location f, Mag]:
Enter values as space-separated per line (3 values per row), or leave empty and press Enter to skip.
Example: for {dims} values: {' '.join(['val'+str(i+1) for i in range(dims)])}
Type 'done' when finished.

>  4 6 -10000
>  done
Enter Linear Loads [location i, location f, startMag, endMag]:
Enter values as space-separated per line (4 values per row), or leave empty and press Enter to skip.
Example: for {dims} values: {' '.join(['val'+str(i+1) for i in range(dims)])}
Type 'done' when finished.

>  done

--- Summary of Entered Loads ---
Point Loads:
 [[ 2.0e+00  0.0e+00 -1.5e+04]]
Point Moments:
 [[3.e+00 2.e+04]]
Distributed Loads:
 [[ 4.e+00  6.e+00 -1.e+04]]
Linear Loads:
 [[0. 0. 0. 0.]]
```

Figure 12

The reactions Va, Ha, and Vb are printed as follows:



```
[72]: print('The vertical reaction at A (Va) is',reactions[0], 'kN')
      print('The horizontal reaction at A (Ha) is',reactions[1], 'kN')
      print('The vertical reaction at B (Vb) is',reactions[2], 'kN')

      The vertical reaction at A (Va) is 9999.999999999998 kN
      The horizontal reaction at A (Ha) is 0.0 kN
      The vertical reaction at B (Vb) is 25000.0 kN
```

Figure 13

Finally, the shear force and bending moment diagrams are generated:

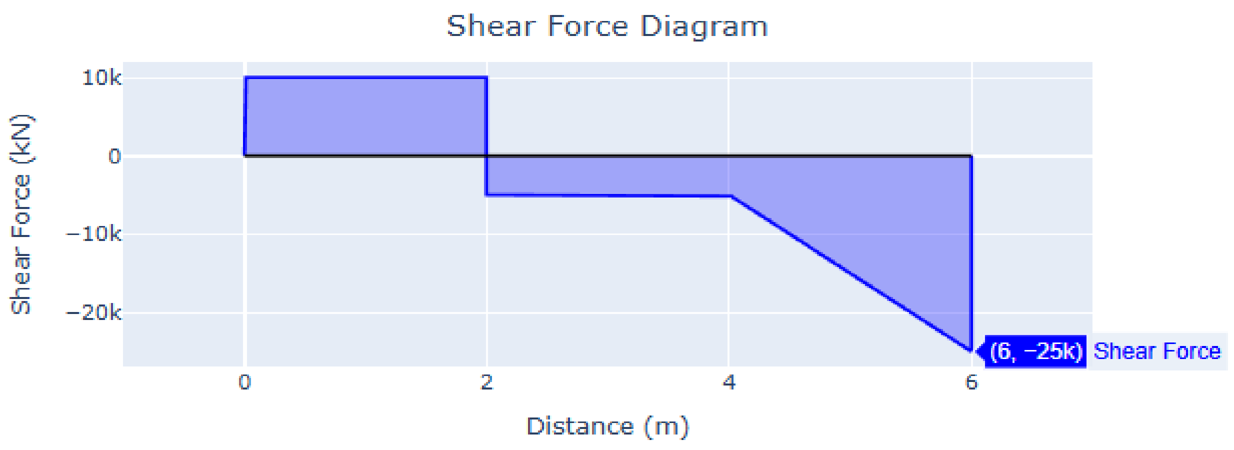


Figure 14

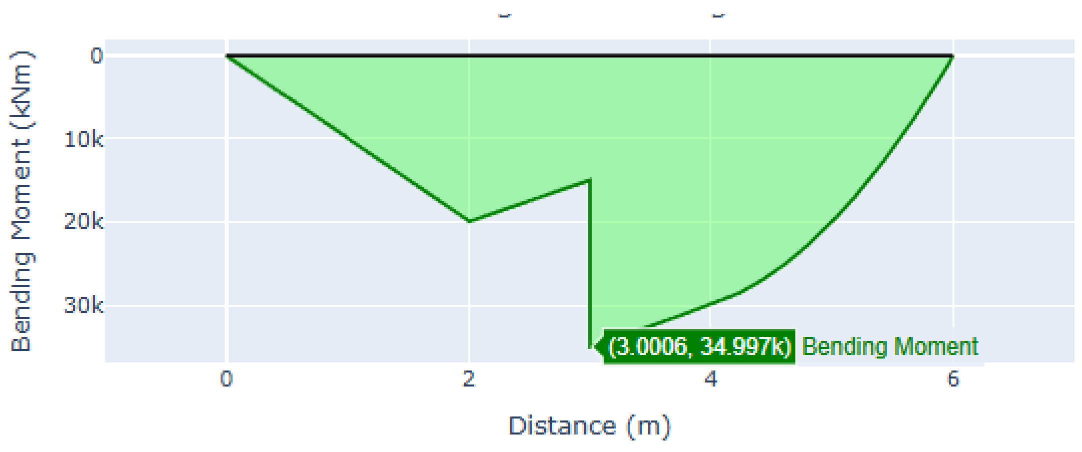


Figure 15

To confirm the validity of the code, the problem must also be solved in a traditional manner. By applying the equilibrium moment equation at point A and point B, with CCW defined as positive, and the horizontal equilibrium equation, the following results are obtained:

$$\Sigma M_A = 0; \quad Vb(6) - 15(2) - 20 - 10(2)(5) = 0 \quad Vb = 25.0\,kN$$

$$\Sigma M_B = 0; \quad 10(2)(1) + 15(4) - 20 - Va(6) = 0 \quad Va = 10.0\,kN$$

$$\Sigma Fx = 0; \quad Ax = 0$$

Furthermore, the shear force and bending moment diagrams are the following:

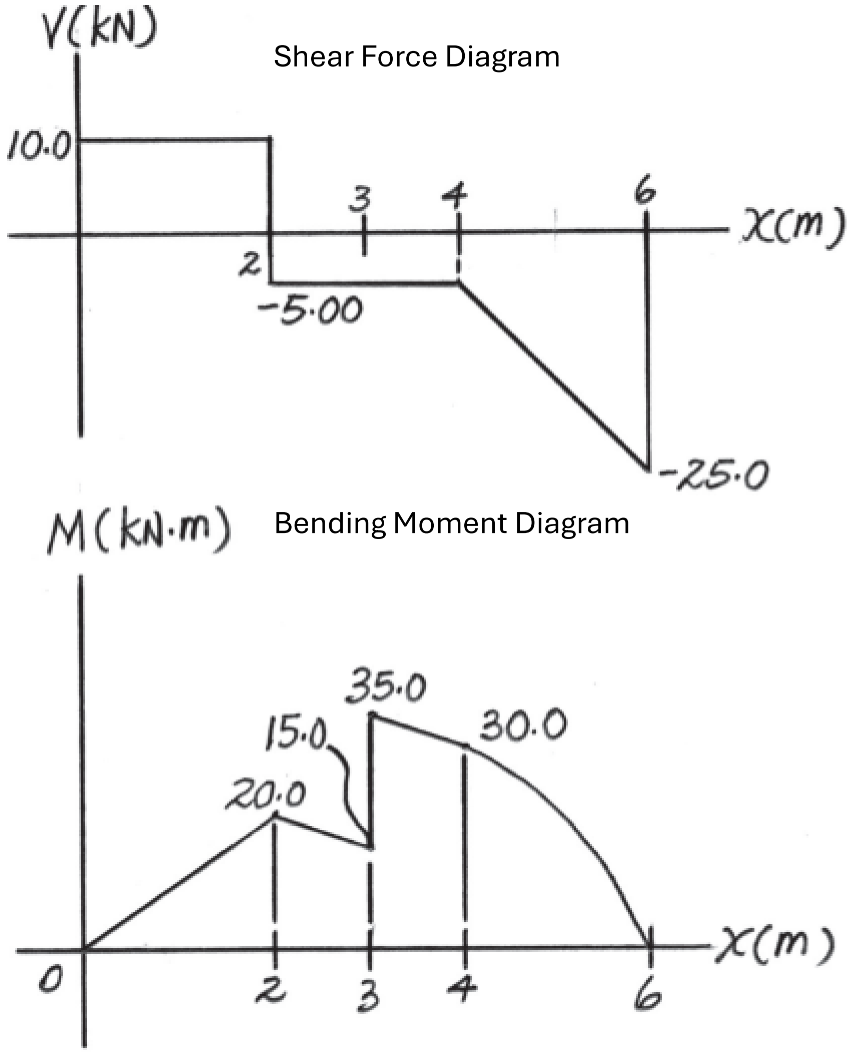


Figure 16: (Hibbeler, 2017) Shear Force and Bending Moment Diagrams

5. Evaluation

It is evident that the reaction forces computed by the code match those obtained through conventional hand calculations. The shear force diagram also aligns well with the manual solution, showing correct shape and scale. The bending moment diagram is similarly accurate; however, it might appear inverted. Despite this visual reversal, the moment values are positive and match the answer key. This inversion occurs because the textbook uses the standard convention—counterclockwise (CCW) moments as positive—while our code treats clockwise (CW) moments as positive (as noted in the comments). This difference in sign convention, not a calculation error, is why the diagram appears flipped.

6. Conclusion

In conclusion, this project successfully demonstrates the development of a Python-based tool for analyzing simply supported beams with various load types, including point loads, moments, UDLs, and LDLs. The code effectively calculates support reactions, and the shear and bending moment diagrams were verified through comparison with traditional hand calculations and reference solutions. The only notable difference was a visually inverted bending moment diagram, due to using clockwise moments as positive—even though the magnitude matches the textbook results—highlighting a clear case of sign-convention discrepancy rather than computational error. By automating these fundamental structural analysis tasks, the tool offers speed, accuracy, and educational value. Overall, this project integrates engineering theory, efficient programming, and visual output to produce a reliable and extensible statics calculator.

7. Further Improvements

Several enhancements could potentially increase the tool's versatility and analytical power in the future. Firstly, supporting multi-span beams with intermediate supports would allow modelling of more complex real-world structures like continuous bridges or multiple connected members. Expanding the range of support types, such as fixed, cantilever, and propped conditions, would further enable analysis of different boundary constraints found in frames and overhangs. Moreover, enhancing the implementation to handle 3D beam structures and varied spatial orientations would allow analysis of non-planar problems and off-axis loading cases, such as those found in trusses or spatial frames. Finally, incorporating support for complex boundary conditions such as built-in (fixed) and guided supports, along with enabling loading in various spatial orientations, would enhance the tool's capabilities, accommodating a wider range of structural configurations and advanced engineering scenarios.

8. Bibliography

Carroll, S. (n.d.). *Building a shear force and bending moment diagram calculator in Python*. EngineeringSkills.com. https://www.engineeringskills.com/project/building-a-shear-force-and-bending-moment-diagram-calculator-in-python

Hibbeler, R. C. (2017). *Engineering mechanics: Statics* (14th ed.). Pearson.

Libretexts. (2023, August 24). 1.4: Internal forces in beams and frames. Engineering LibreTexts. https://eng.libretexts.org/Bookshelves/Civil_Engineering/Structural_Analysis_%28Udoeyo%29/01%3A_Chapters/1.04%3A_Internal_Forces_in_Beams_and_Frames