Assignment 6: Evaluating bending stiffness in beams

Overview

The finite difference method is commonly used to evaluate the derivatives of a given set of data. In this assignment you will use the finite difference method to compute the bending stiffness of beams using test data and use the computed results in evaluating an analytical expression for the plastic bending stiffness.

Bending stiffness of a given beam cross section relates the applied bending moment to the curvature of the beam neutral axis (Figure 1a). Bending stiffness of different cross sections of beams are measured and are utilized in designing civil and mechanical structures. One can evaluate the bending stiffness for homogenous cross sections under elastic limit using the Young's modulus (E) and moment of inertia (I). Whereas if a portion of the cross section undergoes plastic deformation, the bending stiffness is not a constant anymore and is often evaluated using bending tests. Such tests are performed to measure the bending moment versus curvature of the beam (Figure 1b). The bending stiffness is determined by evaluating the slope of the moment-curvature curve at any given point.

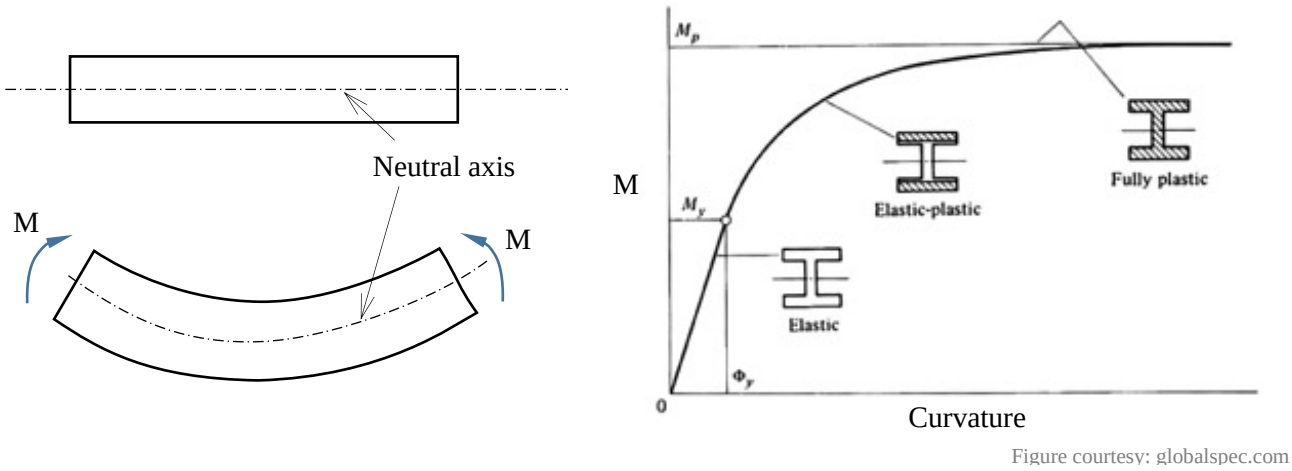


Figure courtesy: globalspec.com

Figure 1: a) Bending of beam b) The moment-curvature curve relation in elastic and plastic regions.

In this assignment, you will use the finite difference method to calculate the bending stiffness of a beam using the available moment-curvature data and evaluate fits from the data to express the bending stiffness as a function of curvature.

Part 1A: Function for approximating derivatives from data

Create a module file named **`diff.py`** in which you create a function for estimating the first derivative of some given data. The function should have the following as its definition:

```
def finite_difference(x, y):
```

where **x** and **y** are 1D NumPy arrays. The function should return another 1D NumPy array, (e.g., "dydx") which contains estimates for dy/dx, the derivative of y with respect to x.

The necessary finite-difference formulae are a bit different from what was covered in class. In these formulae, the step size may vary from point to point. In particular, you should use the following formulae:

- For the first data point, you should use a first-order accurate forward-difference approximation of the form

$$\left(\frac{dy}{dx}\right)_i \approx \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$

- For the last data point use a first-order backward difference approximation:

$$\left(\frac{dy}{dx}\right)_i \approx \frac{y_i - y_{i-1}}{x_i - x_{i-1}}$$

- For the remaining interior points, you should use a second-order accurate central difference approximation. For an arbitrary index i between the first and last index, the formula is

$$\left(\frac{dy}{dx}\right)_i \approx \left(\frac{h_{i-1}}{h_i + h_{i-1}}\right)\left(\frac{y_{i+1} - y_i}{h_i}\right) + \left(\frac{h_i}{h_i + h_{i-1}}\right)\left(\frac{y_i - y_{i-1}}{h_{i-1}}\right)$$

  where $h_i = x_{i+1} - x_i$ and $h_{i-1} = x_i - x_{i-1}$

Your **finite_difference** function and **diff** module should also meet the following requirements.

- Immediately after you import NumPy, and assuming you do so using

```
import numpy as np
```

  add the following line, which is needed to prevent NumPy from treating certain exceptions as warnings.

```
np.seterr(all='raise')
```

- The function **finite_difference** should verify that the input **x** and **y** arrays have the same size. If they do not have the same size, you should raise a **ValueError** with a useful message.
- Use a **try** statement around the finite-difference approximations, and include an

```
except FloatingPointError:
```

  clause which will catch division by zero when NumPy has been imported. When this exception is caught, it means that the user provided non-unique **x** data. Furthermore, when this exception is caught you should

  - Print the string "x values must be unique" (print exactly this string!); and then

- re-raise the exception with a simple **`raise`** statement (just "`raise`"; nothing more).

Part 1B: Function for calculating the derivative of a polynomial

In a file named **`poly_deriv.py`**, write a function to compute the derivative of a polynomial fit at a given point based on an array of polynomial coefficients as input. The function should have the following as the first-line of its definition:

```
def polynomial_derivative(coef, x):
```

where **`coef`** is a 1D NumPy array of polynomial coefficients in the order c_0, c_1, c_2, …, and **`x`** is a scalar float at which to calculate the derivative of the polynomial. The function should return the scalar float value of the derivative of the polynomial evaluated at the point **`x`**.

For example, suppose for some interval you use a cubic fit of the data of the form

$$y(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3$$

You can predict the derivative at a point in that interval as the derivative of the fit:

$$\begin{aligned} y'(x) &= c_1 + 2c_2 x + 3c_3 x^2 \\ &= d_0 + d_1 x + d_2 x^2 \\ &= d_0 + x\big(d_1 + x(d_2)\big) \end{aligned}$$

The last expression shows the nested form of the polynomial, as discussed below.

- Your function should handle the possibility that the coefficient array is an empty array by raising a **`ValueError`** with a useful message.
- Your function should correctly handle the possibility that the polynomial is a constant (i.e., the coefficient array is of size 1), such that the derivative is zero across the interval. In this scenario, you need to return zero as the derivative.
- Your function is <u>required</u> to take advantage of the nested form of a polynomial to compute the derivative of the polynomial in a **`for`** or **`while`** loop. (A search for "Horner's rule" or "Horner's method" will show numerous examples; one is given below.)
- It is recommended that you first write and test a function to evaluate a polynomial at a single value using Horner's rule. Then modify that function to compute the derivative of the polynomial by observing the pattern in the coefficients.

<u>Nested Form of a Polynomial</u>

A polynomial of degree n may be computed as a simple sum:

$$P(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots$$

To evaluate the polynomial at a single value of x the Python code could look like

```
def naive_compute_polynomial(coef, x):
    p = 0.0
    for i, a in enumerate(coef):
        p += a * x**i
    return p
```

If the powers are computed as products (e.g., if x^3 is computed as $x \cdot x \cdot x$, such that a factor x^i is computed as i- 1 repeated multiplications) then this requires

$$\frac{n(n+1)}{2} \, multiplications \; \& \; n \; additions$$

The same polynomial may also be expressed in nested form:

$$P(x) = a_0 + x\left(a_1 + x\left(a_2 + \cdots + x(a_{n-1} + x\, a_n)\right)\right)$$

Computing starts from the innermost coefficient a_n and progresses to the left such that no powers are computed:

```
def horner_compute_polynomial(coef, x):
    n = coef.size
    p = coef[-1]
    for i in range(n-2, -1, -1):
        p = coef[i] + (x * p)
    return p
```

This requires only

$$n \; multiplications \; \& \; n \; additions$$

Of course, Python is often best used as a prototyping language where the naïve approach is more than sufficient for quick calculations. (Or one could use the NumPy function **`numpy.polynomial.polynomial.polyval`**, which you are not allowed to do in this assignment.) If speed was actually that important, and this was not a learning exercise, then a compiled language (like Fortran or C, which we will learn a little about at the end of this course) is likely a better choice.

Part 2: Computation of bending stiffness

The goal of Part 2 of this assignment is to estimate the bending stiffness of the moment-curvature data of a beam cross section using the contents of Part 1. The data is provided in file name **`S1.txt`**. The units of moment and curvature are **`Nm`** and $\mathbf{m}^{-1}$ respectively.

- In a file named **`hw06.py`**, use your **`finite_difference`** function from the **`diff`** module and the polynomial derivative function from the **`poly_deriv`** module to estimate the bending stiffness of the given data as a function of curvature.
- Load and plot the moment-curvature data. The data points shall be plotted as dots.
- Visualize the data and visually identify the proportionality limit. That is, visually identify the moment/curvature until which the curve stays linear and then identify from the text file the count of this data point. In practice, software available in testing machines accomplish this automatically. See (in case of a stress-strain curve), for instance:

  https://www.instron.com/en/our-company/library/glossary/p/proportional-limit

  On that note, mechanical engineering majors will use an Instron in MANE-4040 Mechanical Systems Lab, and aeronautical engineering majors will use an Instron in MANE-4920 Aerospace Structures Lab. The nuclear engineers do not get to use an Instron, but they get to irradiate materials to change their material properties such as elastic modulus and the proportionality limit.
- Compute the fit for both the elastic region and the plastic region using the fit_data module from your earlier homework assignment. It is up to you to determine the appropriate degree of each fit you use in the plastic region.
- Devote a clearly labelled block of comments in file **`hw06.py`** to explain in your own words why you chose the degree you did for each of your fits.
- Compute the derivative of the moment-curvature data using your **`finite_difference`** function.
- Compute the derivative of the fit by utilizing your **`polynomial_derivative`** function.

Part 3: Plot the results

In this part you will present your results through plots. For full credit, please make sure you follow these instructions.

- Use a figure size of 6x8 and use the **`plt.subplot()`** function to create two subplots. For example, to create two subplots (one below another) in a figure, follow the following lines of code

```
plt.figure(figsize=[6,8])
plt.subplot(2, 1, 1)
plt.plot(xdata, ydata, label = "x-y data")
```

  To plot on the second sub-plot, you could use the following lines

```
plt.subplot(2, 1, 2)
plt.plot(xdata, stiffness, label = "bending stiffness")
```

- The first subplot must plot the data and fits of the moment-curvature values and the second subplot must plot the bending stiffness data and its fit.
- See the following link for additional details:

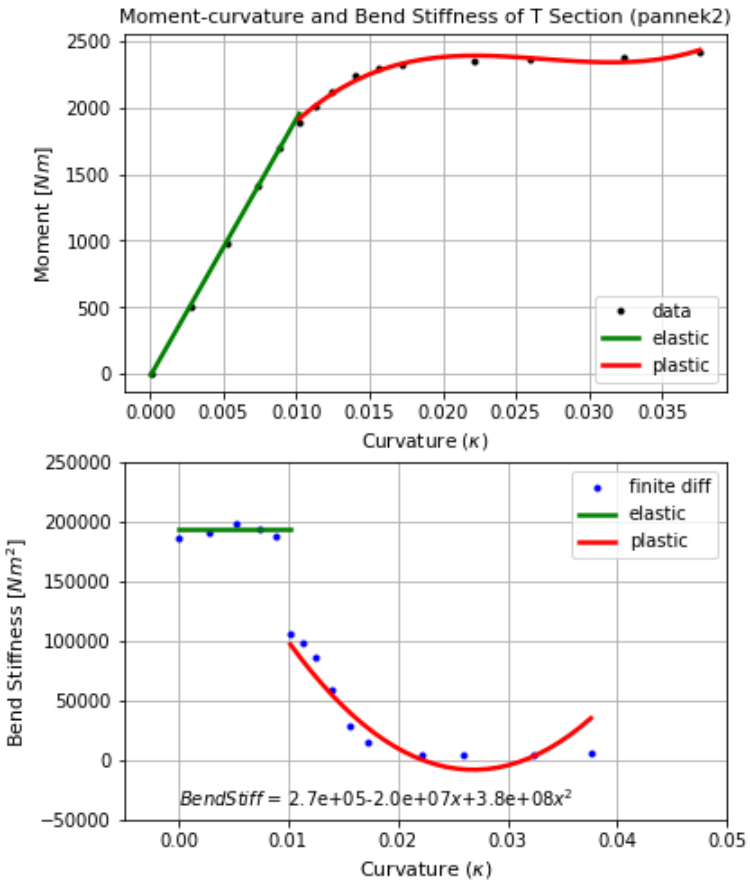
https://matplotlib.org/3.3.2/api/_as_gen/matplotlib.pyplot.subplot.html

- To include mathematical expressions (e.g. formulae) in text in the plot, you can prefix the string with an r' ' representing raw strings, with the math expression enclosed between $...$ as in the examples in the previous homework assignment. For more details on writing mathematical expressions, visit the webpage: https://matplotlib.org/tutorials/text/mathtext.html.
- Include a legend that distinguishes the data and fit curves. (You might look up the **`fig.legend()`** method.) Make sure the legend does not cover the data.
- Print the text containing the polynomial of the bending stiffness (with the curvature variable as x) at an appropriate location with proper formatting. For example,

$$BendStiff = 26.1 + 10.4x - 19x^2 + 0.1x^3$$

- Provide "**`Moment-curvature and Bending stiffness of T section(rcsid)`**" as the title for the plot that includes your RCS_ID in parenthesis, and the values for the coefficients in the legend of the fit.

A sample plot is shown below.



Save the figure as **hw06_plot.png** and submit it along with **diff.py** and **poly_deriv.py**. in the order specified in Submitty. Be sure to view your plot file for quality and accuracy before submitting it.

- Be sure to check the actual **.png** file you are submitting, not what Spyder displays as a result of running your code.

Note that you must have all your plot code inside an if block to prevent Submitty from plotting anything when running your code. For example,

```
Submitty = False
if (not Submitty):
   import matplotlib.pyplot as plt
   plt.figure(figsize=[6,8])
   ...
```