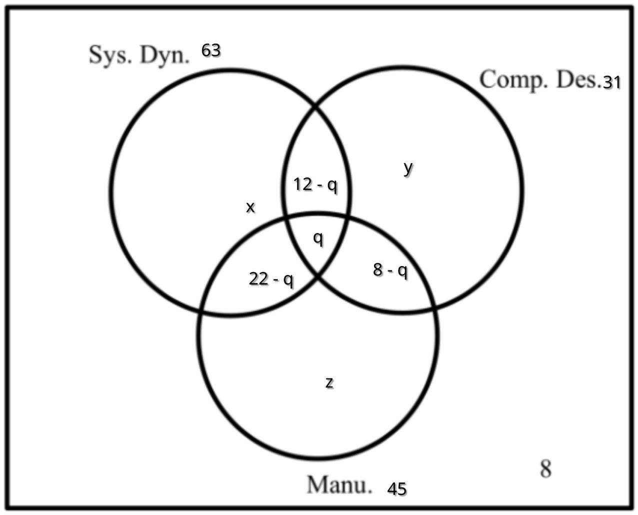
MCEN 3047 - Homework 3

Jack Reilly Goldrick

October 10, 2024

0.1 Problem 1

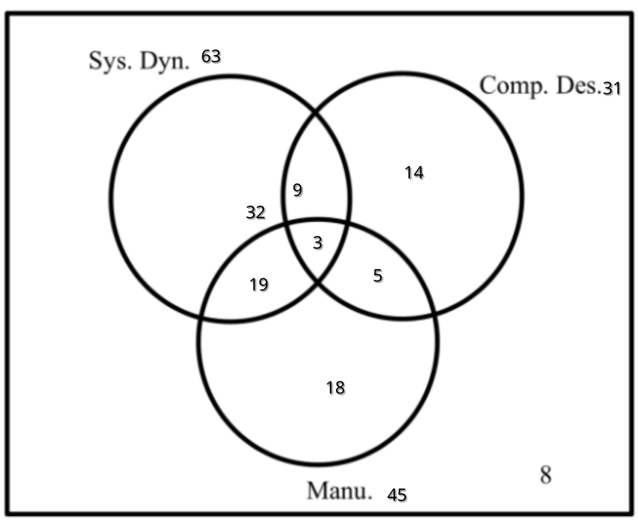


0.1.1 Part A

- Solving the following system of equations yields the following result:

$$|SysD \cap Manu \cap CompD| = 3$$

0.1.2 Part B



0.1.3 Part C

$$P = \frac{32}{108}$$

$$P = 29.6\%$$

0.1.4 Part D

$$P = \frac{9}{108}$$

$$P = 8.3\%$$

0.1.5 Part E

$$P = \frac{\frac{22}{108}}{\frac{45}{108}}$$

$$P = 48.889\%$$

0.1.6 Part F

$$P = \frac{\frac{22}{108}}{\frac{63}{108}}$$

$$P = 34.92\%$$

0.2 Problem 2

0.2.1 Part A

$$\frac{\partial \delta}{\partial I} = -\frac{PL^3}{3EI^2}$$

$$\frac{\partial \delta}{\partial L} = \frac{PL^2}{EI}$$

$$\sigma_\delta = \sqrt{(\frac{PL^3}{3EI^2})^2(\sigma_I)^2 + (\frac{PL^2}{EI})^2(\sigma_L)^2}$$

$$\sigma_\delta = .67998\text{cm}$$

0.2.2 Part B

The code returns the deflection of the beam as 2.94e-02 m with an error of 6.72046e-03 m.

0.3 Problem 3

The spring constant is 10.04 N/m with an uncertainty of 0.2064 N/m.
The Damping constant is 3.06 N/m/s with an uncertainty of 0.0561 N/m/s.

0.4 Problem 4

- Question: What happens to the least squares solution x of $b = Ax$ when A is an orthogonal matrix (A matrix with Orthonormal Basis Vectors):

  [1.] $x = A^T b$

  [2.] $x = (A^T A)^{-1} A^T b$

  [3.] $x = (A^T A)^{-1} b$

  [4.] $x = (A^T A) b$

- Answer [1.]

0.5 Code

```
import torch as tc
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy.optimize import curve_fit
import p_power as pp

class problem_1:
    pass

class problem_2:

    @staticmethod
    def deflection(P, L, E, I):
        """
        Calculate the deflection of a beam under a point load.

        Parameters
        ----------
        P :
            The point load.
        L :
            The length of the beam.
        E :
            The modulus of elasticity.
        I :
            The moment of inertia.

        Returns
        -------

            The deflection of the beam.
        """
        return P*L**3/(3*E*I)

    @staticmethod
    def central_derivative_deflection(P, L, E, I, h, targ=None):
        if h==0:
            return 0

        """
        Calculate the deflection of a beam under a point load
            ↪ using central difference.
```

```
        Parameters
        ----------
        P :
            The point load.
        L :
            The length of the beam.
        E :
            The modulus of elasticity.
        I :
            The moment of inertia.
        h :
            The step size.

        Returns
        -------

            The deflection of the beam.
        """

        match targ:
            case 'P':
                return (problem_2.deflection(P+h, L, E, I) -
                    problem_2.deflection(P-h, L, E, I))/(2*h)
            case 'L':
                return (problem_2.deflection(P, L+h, E, I) -
                    problem_2.deflection(P, L-h, E, I))/(2*h)
            case 'E':
                return (problem_2.deflection(P, L, E+h, I) -
                    problem_2.deflection(P, L, E-h, I))/(2*h)
            case 'I':
                return (problem_2.deflection(P, L, E, I+h) -
                    problem_2.deflection(P, L, E, I-h))/(2*h)
            case None:
                return None
            case _:
                raise ValueError('Invalid target.')


    def error_prop(self, err_P, err_dL, err_dE, err_dI):
        """
        Calculate the error in the deflection of a beam under a
            point load.

        Parameters
        ----------
```

```
        sig_P :
            The error in the point load.
        sig_dL :
            The error in the length of the beam.
        sig_dE :
            The error in the modulus of elasticity.
        sig_dI :
            The error in the moment of inertia.

        Returns
        -------

            The error in the deflection of the beam.
        """
        return np.sqrt((err_P)**2 + (err_dL)**2 + (err_dE)**2 + (
            ↪ err_dI)**2)


    def run(self, P=500, L=.1, I=8.1e-11, E=70e9, sig_P=0, sig_L
        ↪ =.7e-2, sig_I=.72e-11, sig_E=0):
        """
        Run the deflection calculation.

        Parameters
        ----------
        P :
            The point load.
        L :
            The length of the beam.
        E :
            The modulus of elasticity.
        I :
            The moment of inertia.
        sig_P :
            The error in the point load.
        sig_dL :
            The error in the length of the beam.
        sig_dE :
            The error in the modulus of elasticity.
        sig_dI :
            The error in the moment of inertia.
        h :
            The step size.

        Returns
        -------
```

```
            The deflection of the beam.
        """



        dP = self.central_derivative_deflection(P=P, L=L, E=E, I=I
            ↪ , h=sig_P, targ='P')
        dL = self.central_derivative_deflection(P=P, L=L, E=E, I=I
            ↪ , h=sig_L, targ='L')
        dE = self.central_derivative_deflection(P=P, L=L, E=E, I=I
            ↪ , h=sig_E, targ='E')
        dI = self.central_derivative_deflection(P=P, L=L, E=E, I=I
            ↪ , h=sig_I, targ='I')

        err_d = self.error_prop(sig_P * dP, sig_L * dL, sig_E * dE
            ↪ , sig_I * dI)
        print(f'The deflection of the beam is {self.deflection(P,
            ↪ L, E, I):.2e} m with an error of {err_d:.5e} m.')
        return err_d


class problem_3:
    @staticmethod
    def get_df(file_path):
        df = pd.read_csv(file_path)
        return df

    @staticmethod
    def tensorize_df(df):
        # print(data.head())
        # Convert the data to a numpy array
        data = df.to_numpy()
        x1 = data[:, 1]
        x0 = data[:, 0]

        return tc.tensor(x1, dtype=tc.float), tc.tensor(x0, dtype=
            ↪ tc.float)
    @staticmethod
    def model(t, A, phi, b, omega_d):


        return (A * np.exp(-b *.5 * t) * np.cos(omega_d * t + phi)
            ↪ )
```

```
def estimate_spring(self, x, t, initial_guess=None):
    if initial_guess is None:
        initial_guess = [1, 1, 1, 1]
    # import pdb; pdb.set_trace()
    params, covariance = curve_fit(self.model, xdata=t.numpy()
        ↪ , ydata=x.numpy(), p0=initial_guess)

    # Extract estimated parameters
    A_est, phi_est, beta_est, omega_est = params
    # print(params)

    return covariance, beta_est, omega_est

def comp_k(self, omega_d, b):
    return omega_d**2 + .25 * b**2

def central_derivative_k(self, omega_d, b, h, targ=None):
    if h==0:
        return 0
    match targ:
        case 'omega_d':
            return (self.comp_k(omega_d+h, b) - self.comp_k(
                ↪ omega_d-h, b))/(2*h)
        case 'b':
            return (self.comp_k(omega_d, b+h) - self.comp_k(
                ↪ omega_d, b-h))/(2*h)
        case None:
            return None
        case _:
            raise ValueError('Invalid target.')


def run(self):
    df = self.get_df("../data/p3.csv")

    x, t = self.tensorize_df(df)

    # print(t)
    # import pdb; pdb.set_trace()

    param_covar, b_est, omega_d_est = self.estimate_spring(x=x
        ↪ , t=t)

    dk_db = self.central_derivative_k(omega_d=omega_d_est, b=
        ↪ b_est, h=np.sqrt(param_covar[2,2]), targ='b')
```

```
        dk_dw = self.central_derivative_k(omega_d=omega_d_est, b=b_est, h=np.sqrt(param_covar[3,3]), targ='omega_d')


        uncert = np.sqrt(dk_db**2 * param_covar[2,2] + dk_dw**2 * param_covar[3,3])

        print(f'The spring constant is {self.comp_k(omega_d_est, b_est):.2f} N/m with an uncertainty of {uncert:.4f} N/m.')

        print(f'The Damping constant is {b_est:.2f} N/m/s with an uncertainty of {np.sqrt(param_covar[2,2]):.4f} N/m/s.')




def main():



    p2 = problem_2()
    p2.run()

    p3 = problem_3()
    p3.run()


main()
```