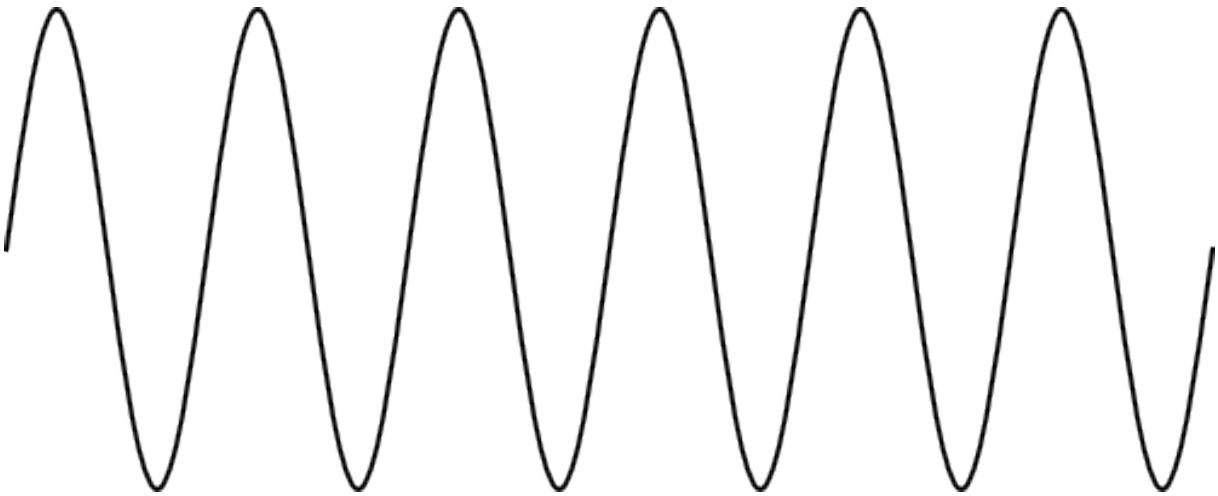
Department of Mechanical Engineering

VIBRATIONS LAB REPORT

ME20016



Frederick Moorhead

frkm20

189088561

1 Summary

The important role that spring-damper systems play in the modern world cannot be understated. In this lab, a real bike shock absorber was tested experimentally using a dynamometer, and its characteristics measured. It was found that the calibration coefficient was 0.0606 m/sV and the spring constant 69.880 kNm^{-1}. The damper was found to exhibit quasi-ideal behavior when tested with differences being found at the extremities of motion. The experiment's objective to fully investigate the characteristics of a real-world spring-damper system were deemed to be met.

Nomenclature

ω	Frequency	E	Dissipated Energy
a	Amplitude	k_v	Calibration Coefficient
C_B	Bump Coefficient		
C_R	Rebound Coefficient	v	Velocity

---

2 Introduction

Dampers undoubtedly play a key role in mechanical systems. In any system where energy is put in, it is often important to remove energy. Damping is used in many different scenarios from *tuned-mass dampers* in buildings to *shock absorbers* on mountain bikes. The aim of this lab was to investigate the latter.

As set out in the lab sheet [1], the objectives of this experiment were to determine the characteristics of a mountain bike shock absorber through experimental data-analysis. To accomplish this a suspension dynamometer (similar to Figure 1).

Figure 1: Image of a typical shock dynamometer

The data generated using the apparatus was provided for analysis by the Mechanical Engineering department.

3 Analysis Method and Results

3.1 Data Import

The first step was to import the data. Prior to this, the data files were altered to change data to SI units. They also had the header removed. The function for this can been seen in Appendix A.2. This then put the data in to data-frames, ready for analysis.

3.2 Data Calibration

Once the data was imported, the velocity was calibrated. Using the function found in Appendix A.3, a calibration coefficient of 0.0606 m/sV.

This function differentiated the displacement with respect to the the time interval (calculated using sample time and frequency from data) and plotted this against the raw velocity values. This can be seen in Figure 2. A linear regression line was plotted on the data using the function found in Appendix A.7.

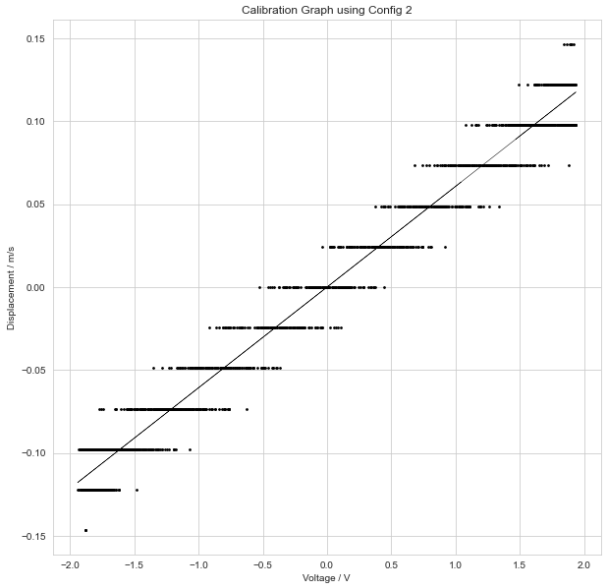


Figure 2: Showing differentiated velocity plotted against raw velocity in volts. The gradient represents the calibration coefficient k_v

$$\text{Calibrated Velocity}, v = k_v \times v_{\text{un-cal.}} \tag{1}$$

The un-calibrated velocity was then calibrated using the coefficient and Equation 1.

3.3 Calculation of spring constant

Using the function in Appendix A.4, the spring constant calculated by taking the gradient of a regression line. This was the spring coefficient k and was found to be 69.880 Nm^{-1}.

3.4 Calculation of Frequency

The frequency of the system was obtained by a function (Appendix A.5) which worked out the average time between cycles and calculates the frequency based of that average.

3.5 Calculation of Damping Coefficients C_B and C_R

Damping coefficients could be calculated in 2 methods. The first method which is more accurate, was to numerically integrate the force-displacement values over half a cycle (first for positive force values and then negative). This was implemented into the function in Appendix A.6. A snippet of the code can be seen below:

```
    area_bump = abs(integrate.trapz(df_bump['Force'],x=df_bump['Displacement']))
    area_rebound = abs(integrate.trapz(df_rebound['Force'],x=df_rebound['
                                        Displacement']))
    freq = frequency(df)
    C_b = (2 * area_bump)/(np.pi * freq * a * a)
    C_r = (2 * area_rebound)/(np.pi * freq * a * a)

    return [C_b,C_r]
```

The coefficients are calculated using the energy dissipation Equation (2) and this can also be seen in the function. A trapezoidal integration was done over the displacement as intervals.

$$E = \frac{1}{2}\pi C_R a^2 \omega \tag{2}$$

Values for the damping coefficients are shown below:

Config	C_B **/ Ns/m**	C_R **/ Ns/m**
2:	2480	2130
3:	2530	9575
4:	2980	2070
5:	1460	5250

Table 1: Showing values obtained from energy equation

3.6 Force graphs for configuration 1 to 5

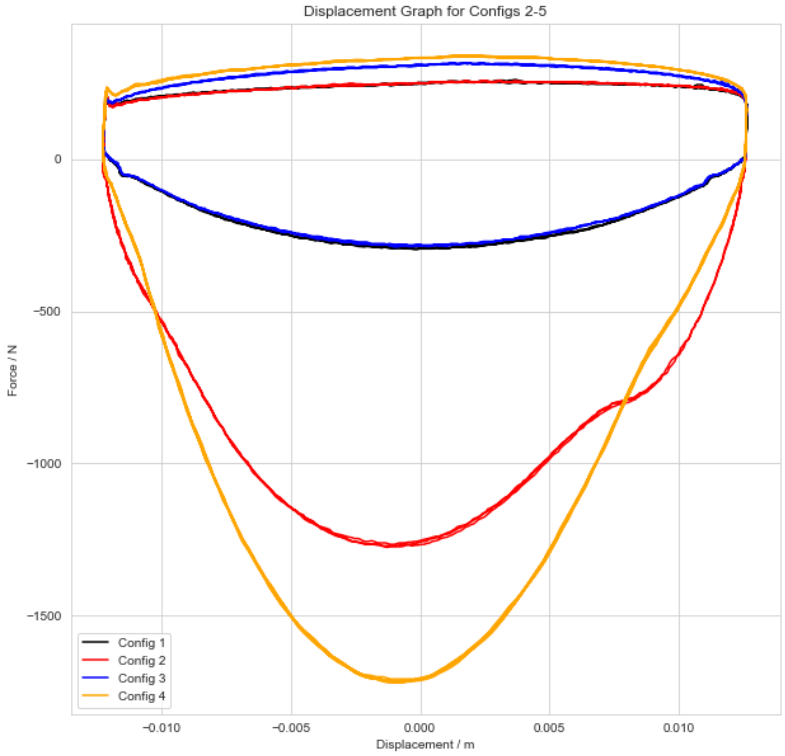


Figure 3: Graph showing force-displacement graphs for configuration 2-5

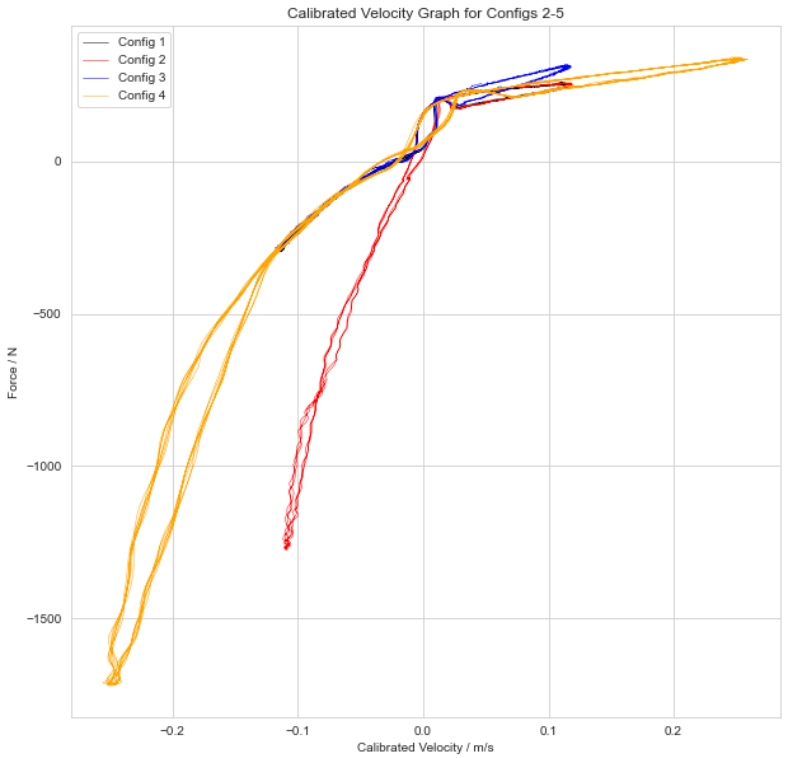


Figure 4: Graph showing force-velocity graphs for configuration 2-5

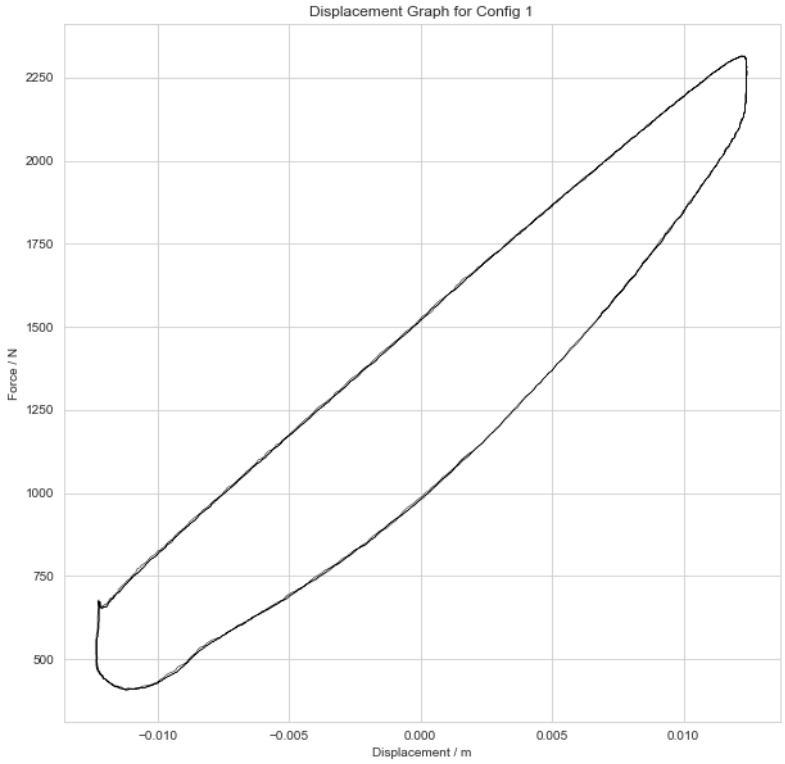


Figure 5: Graph showing force-displacement graphs for configuration 1

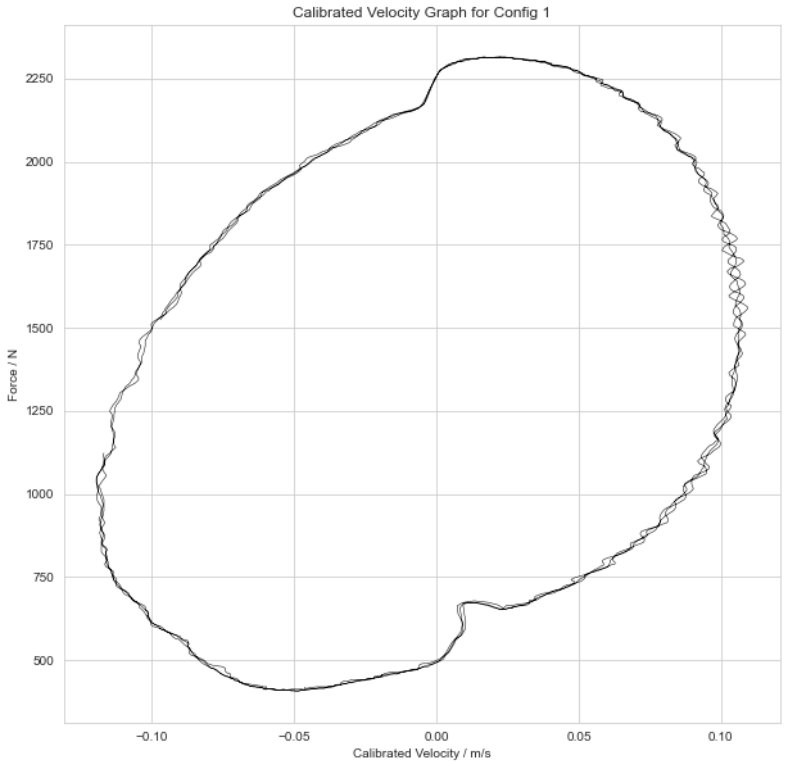


Figure 6: Graph showing force-velocity graphs for configuration 1

4 Discussion

4.1 Damper behavior (Figures 3 and 4)

From Figure 3, the force-displacement behavior of the damper under different conditions is shown. Generally this fit well with the ideal graphs, with the correct characteristic shape. However, it differed on the edges of the graph where vertical lines could be seen. This was due to the coulomb friction in the damper changing direction. It was also found that these lines were not symmetrical about zero. This was caused by pre-load in the damper due to the piggy-back reservoir.

For the force-velocity graph in Figure 4, it was expected to see straight lines. The reason there was an area between the lines, was due to the compliance of the air spring. There was also coulombic behavior around zero velocity for the same reasons described above

4.2 Spring-Damper Behavior (Figures 6 and 5)

When the spring was tested with the damper, the spring dominated the graph due to it being at a much higher force. The same characteristics described in the previous sub-section can be seen in these graphs, although they are less pronounced due to the spring superposition.

4.3 Experimental Error

Whilst the experimental results fit well with the ideal graphs, there was noise and some unexpected behavior from the spring-damper. One of these errors may have arose from the play in the machine. When the cycle reached points of maximum displacement, the top pin moved. This will have slightly altered the force reading measured by the load cell.

Another source of error could have been from taking the measurement of frequency from the machine, however, it was found after analysing the data that when the machine was set to a specific frequency, it was not actually operating at this frequency. This was therefore mitigated by calculating the frequency in the method described in Section 3.4.

One final source of error, may have been the oil, air, and spring increasing in temperature over the course of the experiment. However, this likely would not affect the measurements noticeably. A way to have mitigated this , could simply have been waiting sufficient time between runs, to allow the apparatus to cool back down to ambient temperature. Similarly, the piggyback reservoir looses air over time and needs to be re-compressed. This could have resulted in loss of pre-load, also changing the results over time.

5 Conclusions

Throughout this experiment, a deeper understanding of the characteristics and behavior of dampers was gained. This being the primary objective of the lab, it should be considered a success. Through analysis and applied theory of spring-damper systems, it can be concluded that they can be successfully modeled and numerical characteristics for their spring constant and damping coefficients can be calculated. The main types of non-ideal behavior were also identified.

References

[1] Jens Roesner. "Vibrations Labsheet". In: *University of Bath - Solid Mechanics 3 (ME20016)* (2020).

Appendices

A Python Code

A.1 Importing Modules

```
# For file finding:
import os
import glob

# The usual:
import numpy as np
import pandas as pd
import math
import random as rand
from scipy import integrate

# Plotting:
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

from pylab import rcParams
import seaborn as sb

# ML:
from sklearn.linear_model import LinearRegression

%matplotlib inline
rcParams['figure.figsize'] = 10, 10
sb.set_style('whitegrid')
```

A.2 Import Function

```
# This goes and finds files in the specified path of a certain extension.
# It then makes pandas data-frames out of them

# Can use suppres flag to hush output

def file_finder(path,extension,suppress=False):

    if not suppress:
        print('Looking for files in:'+path+'\n')

    os.chdir(path) # Change dir to path
    files = {}
```

```
    for file in glob.glob(extension):
        if not suppress:
            print('Found',file+'... adding')
        files["file{0}".format('_'+file)] = path+'\\'+file

    if len(files) == 0:
        print('No files found :(')
        return # Exit function if no files found

    print ('\n')

    # Creates data frames from filenames:
    dataframes = {}
    for i,file in enumerate(files):
        try:
            dataframes["df_{}".format(i)] = pd.read_csv(files[file])
        except:
            print('There was an error reading',file,'. Please check and try
                                                    again')
            return
        if not suppress:
            print( 'Writing ' + file + ' to ' + "df_{}".format(i))

    return dataframes

dataframes = file_finder(r'..\data',"*.csv")
```

A.3 Calibration

```
# Function takes a data-frame and calibrates it. Inputs are default
# for this task but can be changed if necessary. It can also be
# made to produce a graph if the quiet flag is set to False

def calibrate(df,t=2,sf=2000,diff_column='Displacement',calib_column='Velocity',
                                        quiet=True,xlab='Calibrating_Target',
                                        ylab='Differential',title='Calibration
                                        Graph'):
    yaxis = pd.DataFrame(np.gradient(df[diff_column], t/sf))
    xaxis = calib_column
    if quiet:
        calib_coef= linreg(df[xaxis],yaxis[0])[0]
        return calib_coef
    else:
        linreg(df[xaxis],yaxis[0],graph=True,title=title,eqn=True,xlab=xlab,ylab
                                                =ylab,save=True,save_name='
```

```
                                                    Calib_graph.jpg')
        return
```

A.4 Spring Coefficient

```
# Function used to calculate spring coeficient by plotting a regression
# line on the config 1 graph. The gradient of this line is the value for k.
def spring_constant(df):
    return linreg(df['Displacement'],df['Force'])[0]

print('spring_constant is',spring_constant(dataframes['df_0']))
```

A.5 Frequency calculation

```
def frequency(df,col='Displacement',t=2,sf=2000):
    ## Returns frequency in radians per second

    #First find points where displacement goes from negative to positive:
    minmax=[]
    df.sort_index(axis=1)
    for i in range(1,len(df)-1):
            if df[col][i]<0 and df[col][i-1]>0 or df[col][i]>0 and df[col][i-1]<
                                                        0 :
                minmax.append(i)

    # Find average distance between samples:
    avr_sample_int=sum([minmax[i+1]-minmax[i] for i in range(0,len(minmax)-1)])/
                                            (len(minmax)-1)
    avr_t_int=(2/2000) * (avr_sample_int*2)

    return  (1/avr_t_int)*( 2 * np.pi)
```

A.6 Damping Coefficients C_B and C_R

```
def damping_coefs(df,velocity='Calibrated Velocity',displacement='Displacement',
                                        force='Force'):
    ## Returns and array [C_b , C_r] for given dataframe

    df.sort_values(by=[displacement],inplace=True)

    a=df['Displacement'].max()

    # Separate possitive values and negative
    df_bump = df[(df[[force]] > 0).all(1)]
    df_rebound = df[(df[[force]] < 0).all(1)]
```

```
    area_bump = abs(integrate.trapz(df_bump['Force'],x=df_bump['Displacement']))

    area_rebound = abs(integrate.trapz(df_rebound['Force'],x=df_rebound['
                                            Displacement']))

    freq = frequency(df)

    C_b = (2 * area_bump)/(np.pi * freq * a * a)
    C_r = (2 * area_rebound)/(np.pi * freq * a * a)

    return [C_b,C_r]
```

A.7 Graph Plotting and Linear Regression Function

```
'''
     ------------------------------------------------------------------------
    | Function can either generate a selected graph of data with a Linear |
    | Regression line plotted on it and the equation of the line if graph |
    | is True                                                             |
    |                                                                     |
    |                                  or                                 |
    |                                                                     |
    |       By default it will return an array with the following:        |
    |                                                                     |
    |         [x1 coef,x2 coef.,std err of x1, std error of x2]           |
     ------------------------------------------------------------------------

'''
def linreg(x_csv,y_csv,graph=False,title='',eqn=False,xlab='',ylab='',save=False
                                        ,save_name='figure.jpeg'):

    x = np.array(x_csv) # Parses in data
    y = np.array(y_csv)

    if graph:

        linreg = LinearRegression()
        x_rs = x.reshape(-1,1)          # Transposes array
        linreg.fit(x_rs,y)              # Makes linear regression line
        y_pred = linreg.predict(x_rs) # Generates predicted y values for each x
                                                value


        fig = plt.figure()
        ax = fig.add_subplot(111)
```

```
        ax.scatter(x_rs,y,3,color='black')                                  # Plots
                                                clock data on a scatter. 6
                                                Series total
        ax.plot(x_rs, y_pred, color='black', linewidth=0.5)  # Plots x values
                                                against predicted y values

        #scale_y = 1e-3

                                                # Set Scale
        #ticks_y = ticker.FuncFormatter(lambda y, pos: '{0:g}'.format(y/scale_y)
                                                )
        #ax.yaxis.set_major_formatter(ticks_y)

                                                # Applies scale to y axis

        plt.xlabel(xlab)    # Naming x axes
        plt.ylabel(ylab)    # Naming y axes
        plt.title(title)

        if eqn:
            print ("Equation for regression line of",
                   title,": >>> y =",linreg.coef_[0],"x +",
                   linreg.intercept_, "<<<")

        if save:
            plt.savefig(save_name)
        return

    if not graph:

        # Polyfit gives linear regression coefs.
        # cov=True produces covarience from which std.error can be generated
        # by squarooting the diagonals of the matrix.

        coef, covar = np.polyfit(x,y,1,cov=True)

        std_err_m = math.sqrt(covar[0,0])
        std_err_intcpt = math.sqrt(covar[1,1])

        return [coef[0],coef[1],std_err_m,std_err_intcpt]
```