System Dynamics II

```
!pip install pypoly2tri idealab_tools foldable_robotics pynamics
```

```
    Requirement already satisfied: pypoly2tri in /usr/local/lib/python
    Requirement already satisfied: idealab_tools in /usr/local/lib/pyt
    Requirement already satisfied: foldable_robotics in /usr/local/lib
    Requirement already satisfied: pynamics in /usr/local/lib/python3.
    Requirement already satisfied: imageio in /usr/local/lib/python3.7
    Requirement already satisfied: shapely in /usr/local/lib/python3.7
    Requirement already satisfied: numpy in /usr/local/lib/python3.7/d
    Requirement already satisfied: matplotlib in /usr/local/lib/python
    Requirement already satisfied: pyyaml in /usr/local/lib/python3.7/
    Requirement already satisfied: ezdxf in /usr/local/lib/python3.7/d
    Requirement already satisfied: sympy in /usr/local/lib/python3.7/d
    Requirement already satisfied: scipy in /usr/local/lib/python3.7/d
    Requirement already satisfied: pillow in /usr/local/lib/python3.7/
    Requirement already satisfied: cycler>=0.10 in /usr/local/lib/pyth
    Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib
    Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=
    Requirement already satisfied: python-dateutil>=2.1 in /usr/local/
    Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/pyth
    Requirement already satisfied: six in /usr/local/lib/python3.7/dis
```

Overview

The main objective of this "Dynamic II" code is to re-consolidate everything learned from testing our system and implement it back into our teams dynamic model. In the previous "Dynamic I" code our team was able to model our mechanism with no constraints, essentially collasing on itself, similar to a puppet when the strings are cut. In this updated dynamics code, our system will simulate real world characteristics such as: forces, torques, friction, gravity, stiffness, damping, enviroment (walls/floor), etc. We will still be simulating one sarrus mechanisms for now, but we plan to add the other two sarrus links in the next iteration of this code. This code will show a sarrus link with a compression spring in the center pushing the sarrus mechanism open. A motor will contract and relase the spring with the use of a sinusoidal input. This will simulate the sarrus mechansim

expanding (making contact with the walls) and contracting (removing conact with the walls). Frictional forces are added and will eventually allow our system to translate up vertical spaces (when we implement all three sarrus mechanisms into the code).

Proposed Mechanism

The team's mechanical design consists of three sarrus mechanisms placed on top of one another vertically as shown in Figure 1a. The sarrus mechanisms are made from cardstock material and the spring/support system is made from PLA printed parts. When stationary, the compression springs will force the sarrus mechanisms to hold an expanded position as shown in Figure 1b. On the other hand, when the servo motors are activated, the spring is compressed and the sarrus mechanisms translate in the negative y-direction and expand in the positive and negative x-directions as shown in Figure 1c. This will allow for the mechanism to anchor to nearby walls similar to the way a razor clam anchors to nearby sand with its “foot”. For testing purposes, the team used their hand force to pull on a string in place of a servo motor and pulley. The team chose three sarrus links in order to allow the mechanism to translate vertically in the following manner:

All sarrus mechanisms will begin motion in an extended position, with the spring mechanism forcing the sarrus link open. The top-most sarrus mechanism will use a servo motor to contract the spring and expand its links in both the positive and negative x-direction. This will lock the mechanism into place with the surrounding walls as shown in Figure 1d. The middle sarrus mechanism will then contract its spring in order to pull the bottom-most link upwards. The bottom-most sarrus mechanism will then contract its spring in order to lock itself against the surrounding walls. The middle and top-most sarrus mechanisms will then expand their springs in order to unlock from the surrounding walls and move upwards. Then the mechanism recycles the process starting with step two. This will allow for the system to translate in an upward or positive y-direction.

In order to achieve this motion, each sarrus mechanism is controlled by a separate motor to allow for individual control of each link. Eventually, an electrical housing

box will screw into the bottom plate of the spring mechanism seen in Figure 2a. This housing will hold the servo motor and pulley.

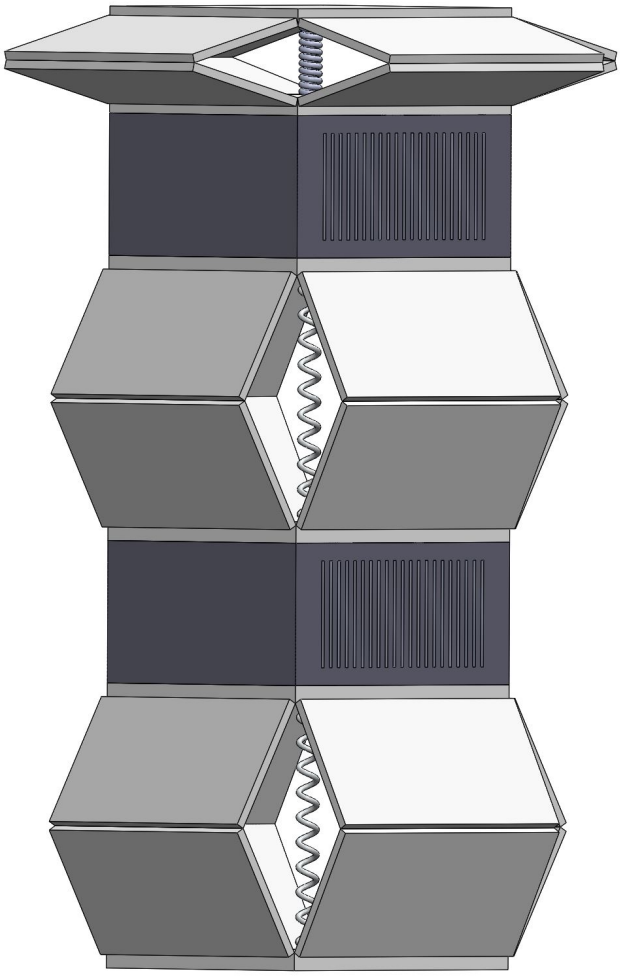


Figure 1a: Three Sarrus Mechanism CAD Model

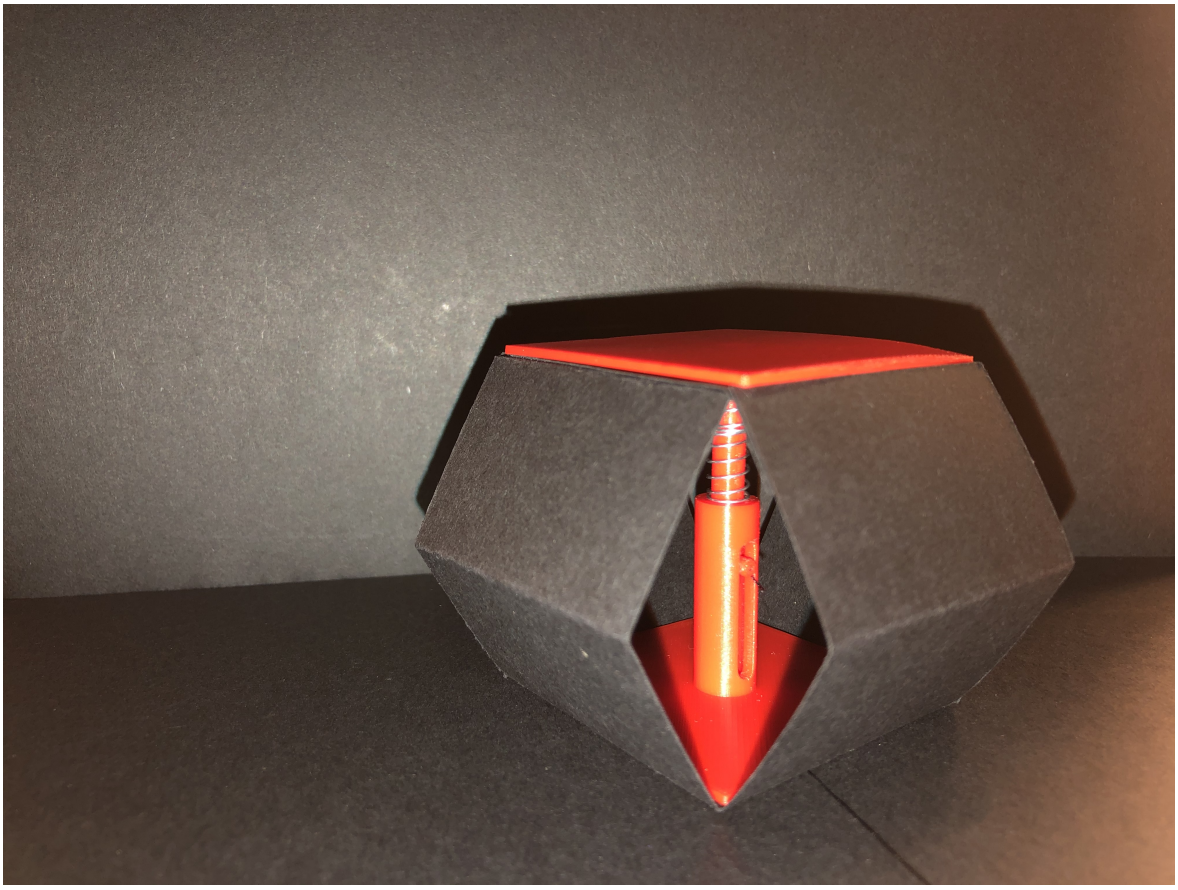


Figure 1b: Sarrus Mechanism at Rest

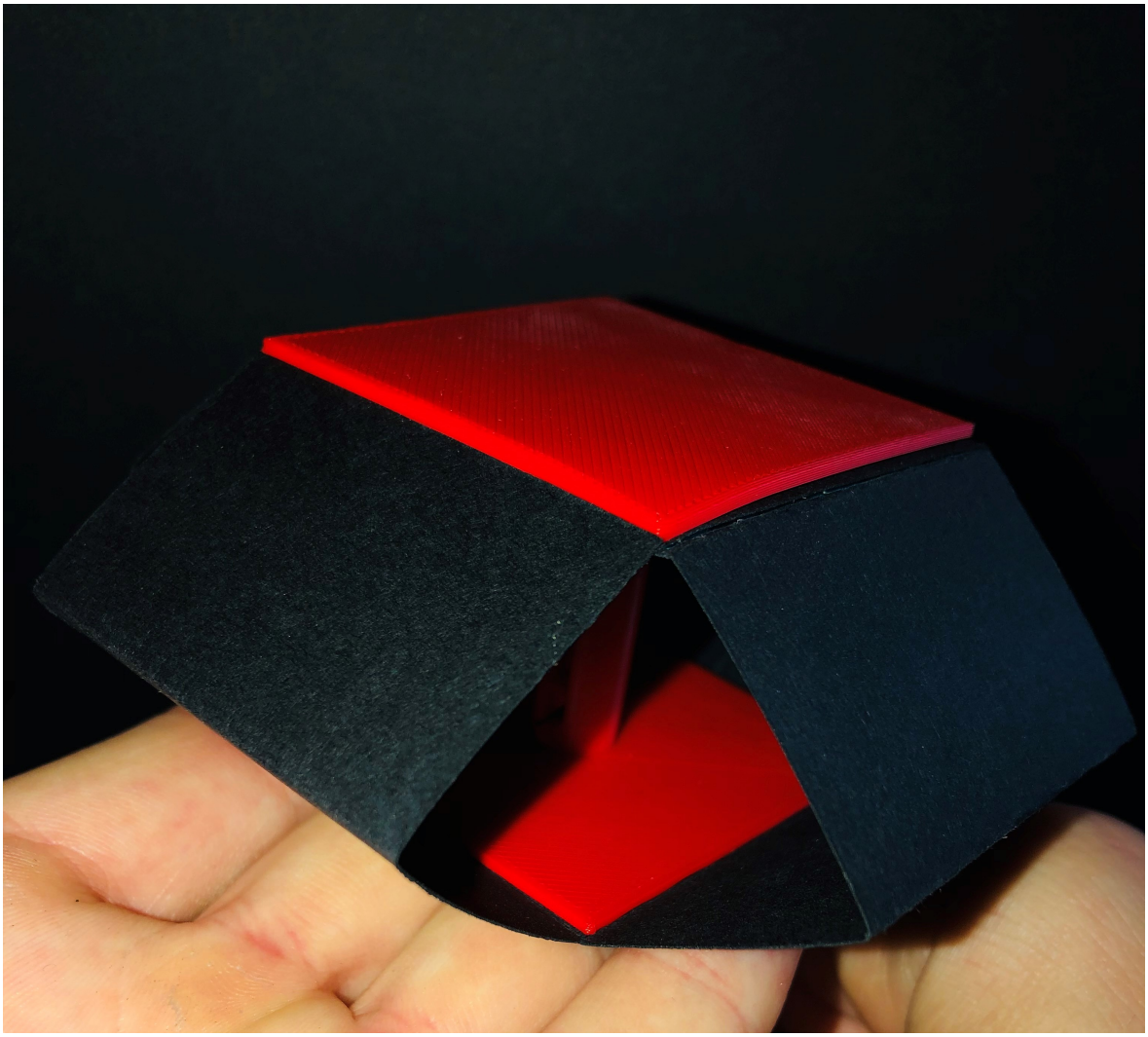


Figure 1c: Sarrus Mechanism Contracted

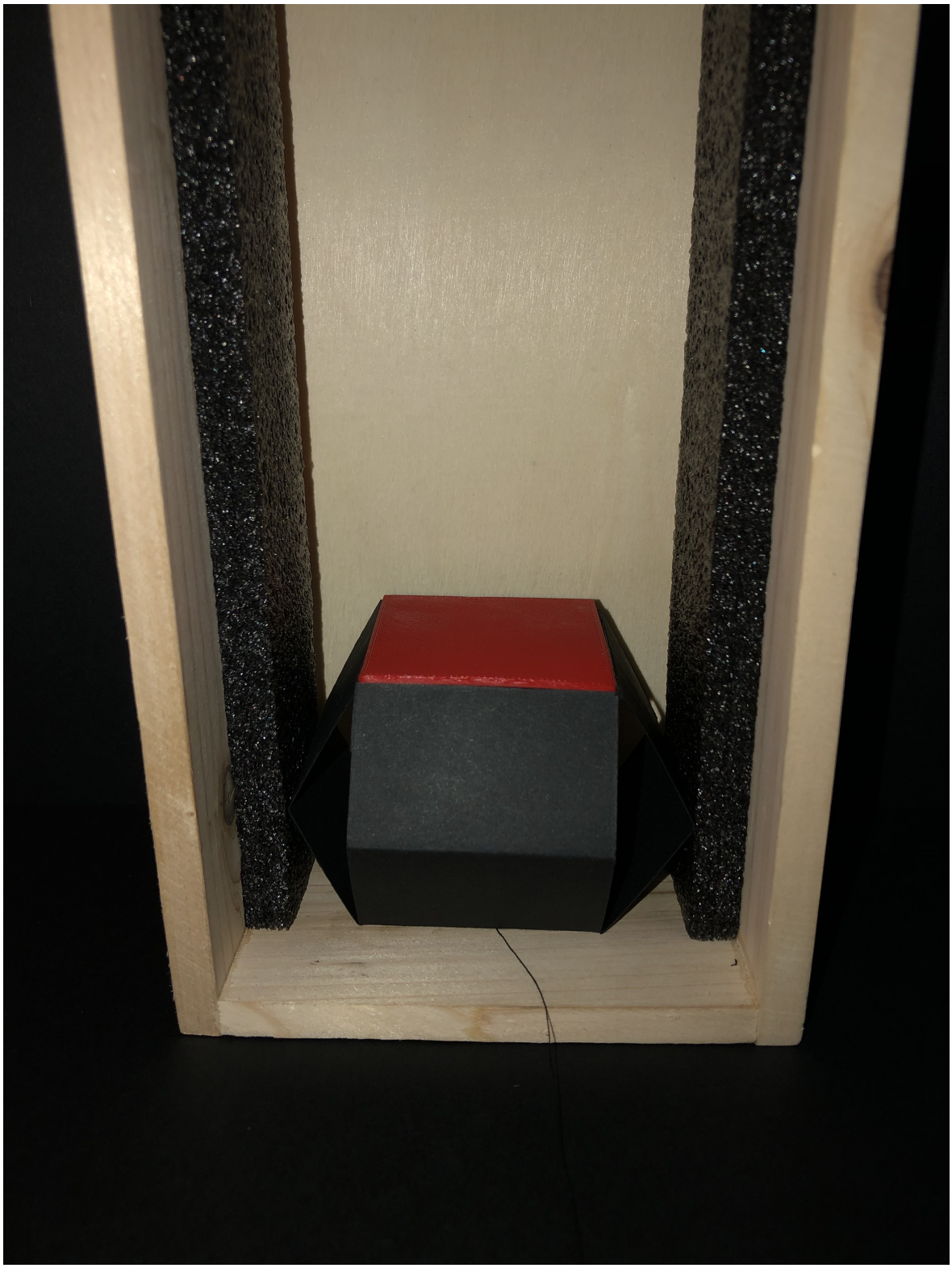


Figure 1d: Sarrus Mechanism Anchored to Walls

Forces and Torques

A compression spring mechanism created with CAD is used to force each of the three sarrus links in an extended initial position Figure 2a. When decompressed, the spring mechanism has a height of 0.05715 meters and 0.03810 meters when compressed. As a result, the compression spring mechanism shown in Figure 2a is **limited to a maximum vertical displacement of 0.01905 meters**. With the use of MATLAB and physical testing from prior assignments, the team found the **spring constant to be approximately 200.4500 newtons per meter**.

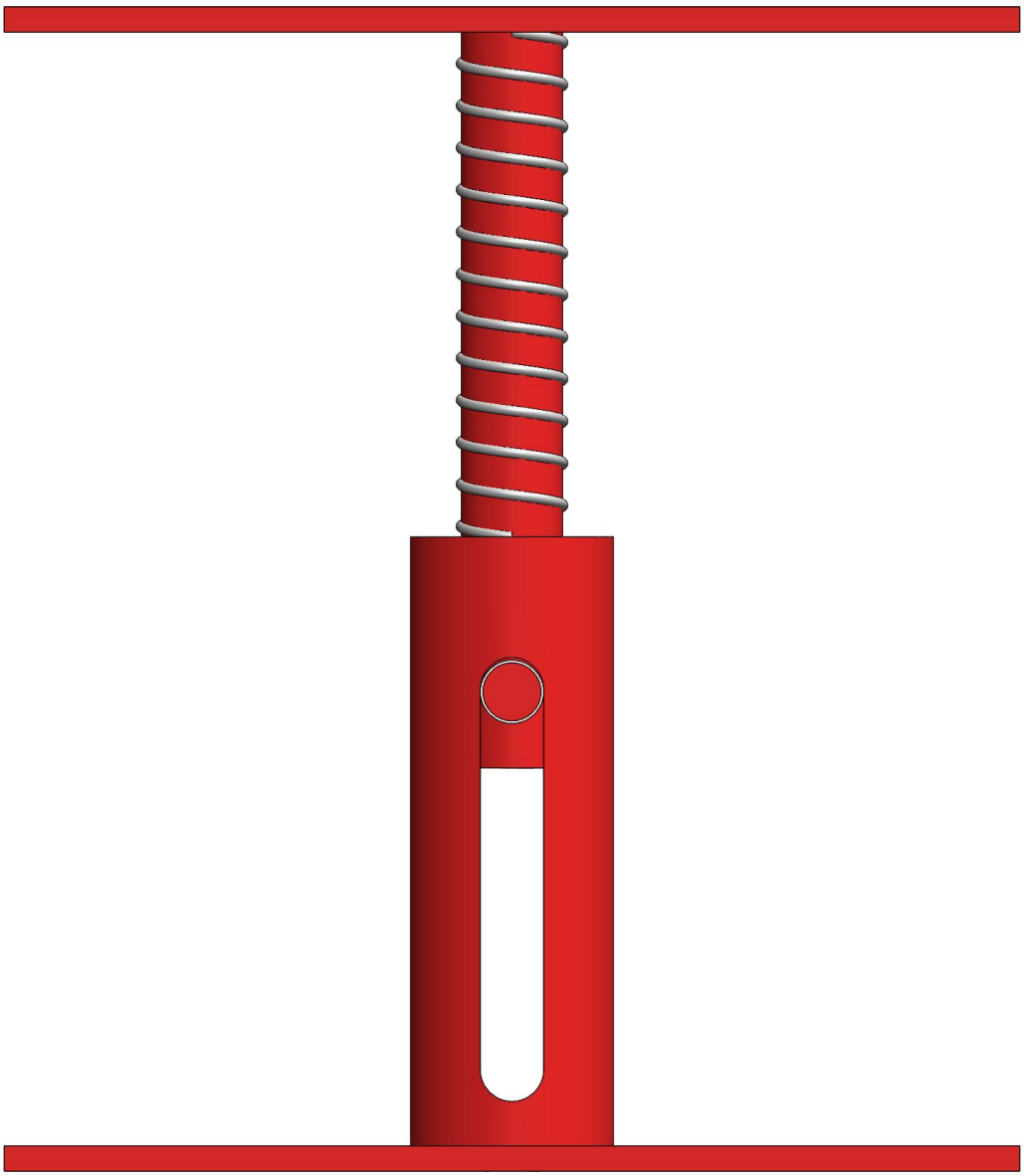


Figure 2a: CAD Rendering of Assembled Printed Parts

A servo motor with a 3D-printed pully spool and thread is used to contract each of the three sarrus links. With a spring constant of approximately 200.4500 newtons per meter and a max displacement of 0.01905 meters, we get a resultant force of 3.8186 Newtons using Hooke's Law. We know that load torque is equal to force (F) multiplied by the distance (r) between the center of rotation and the force point. Assuming the team prints a three dimensional pulley with a radius of 0.00635 meters, the **maximum torque required by the motor is 0.0242 Newton-meters**.

Dynamic Model of Razor Clam Inspired Robotic Mechanism

The Dynamics model below represents one of three sarrus mechanisms from our system. The red frame represents the Newtonian frame and the black frames represent the frame joints. Figure 3a shows all of the points and vectors needed to create our system. All of the vectors are relative to the Newtonian frame shown in red.

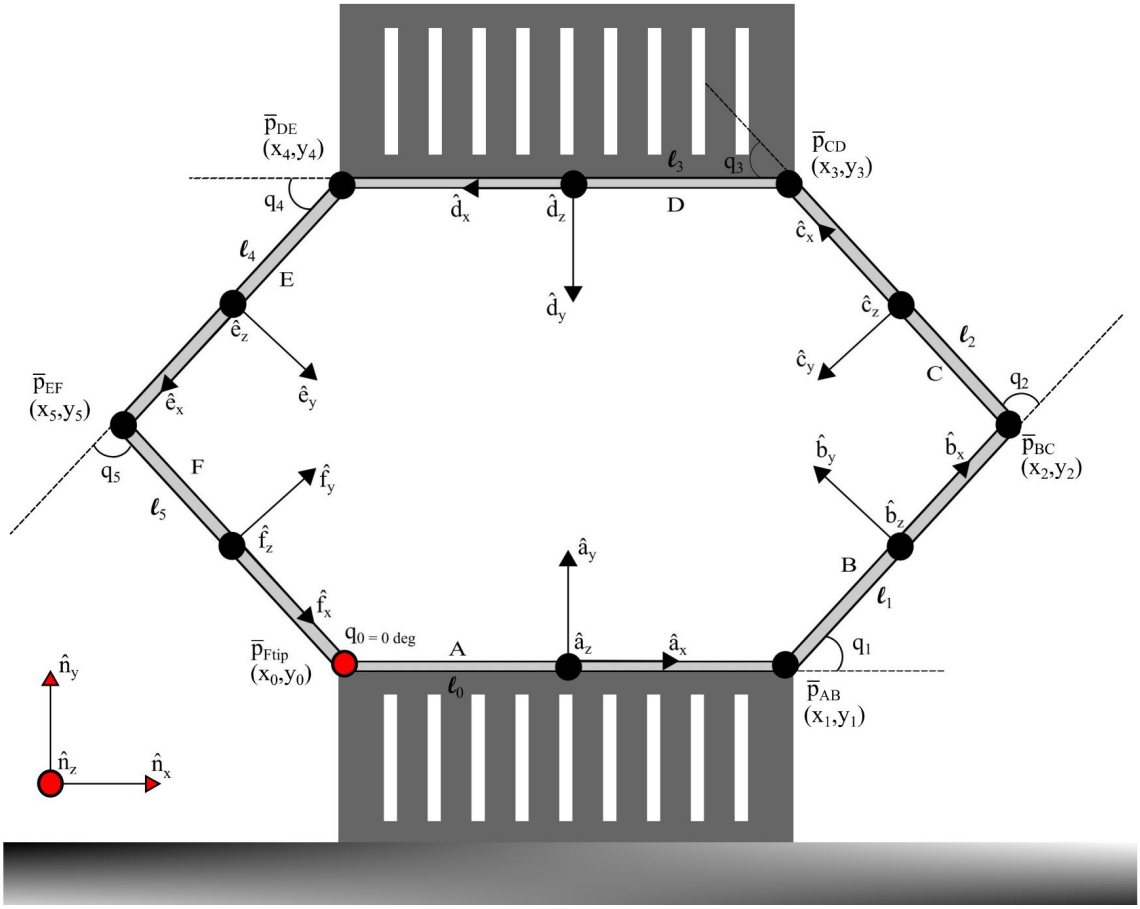


Figure 3a: 2D Dynamics Model of System

Free Body Diagram of Razor Clam Inspired Robotic Mechanism

Our team is neglecting the spring and motor forces for the following calculations. From the free body diagram shown in Figure 3b we determined that the sum of the forces in the vertical direction must be zero at equilibrium. This means that 2*us-mg=0. In the horizontal direction, the two normal forces must add up to zero. From the free body diagram we see that 2*us*Fnormal=mg, therefore, Fnormal=mg/(2us). This gives an expression for how hard the wall pushes on the sarrus mechanism in the perpendicular sense. However, the sarrus mechanisms outer joints must push with the opposite of the net force exerted on it. Lets assume the sarrus mechanism pushes against each wall with a force called Fpush. Therefore, each of the two sarrus joints must push with a magnitude of **Fpush=(mg/2)*sqrt(1+(1/us^2))**.

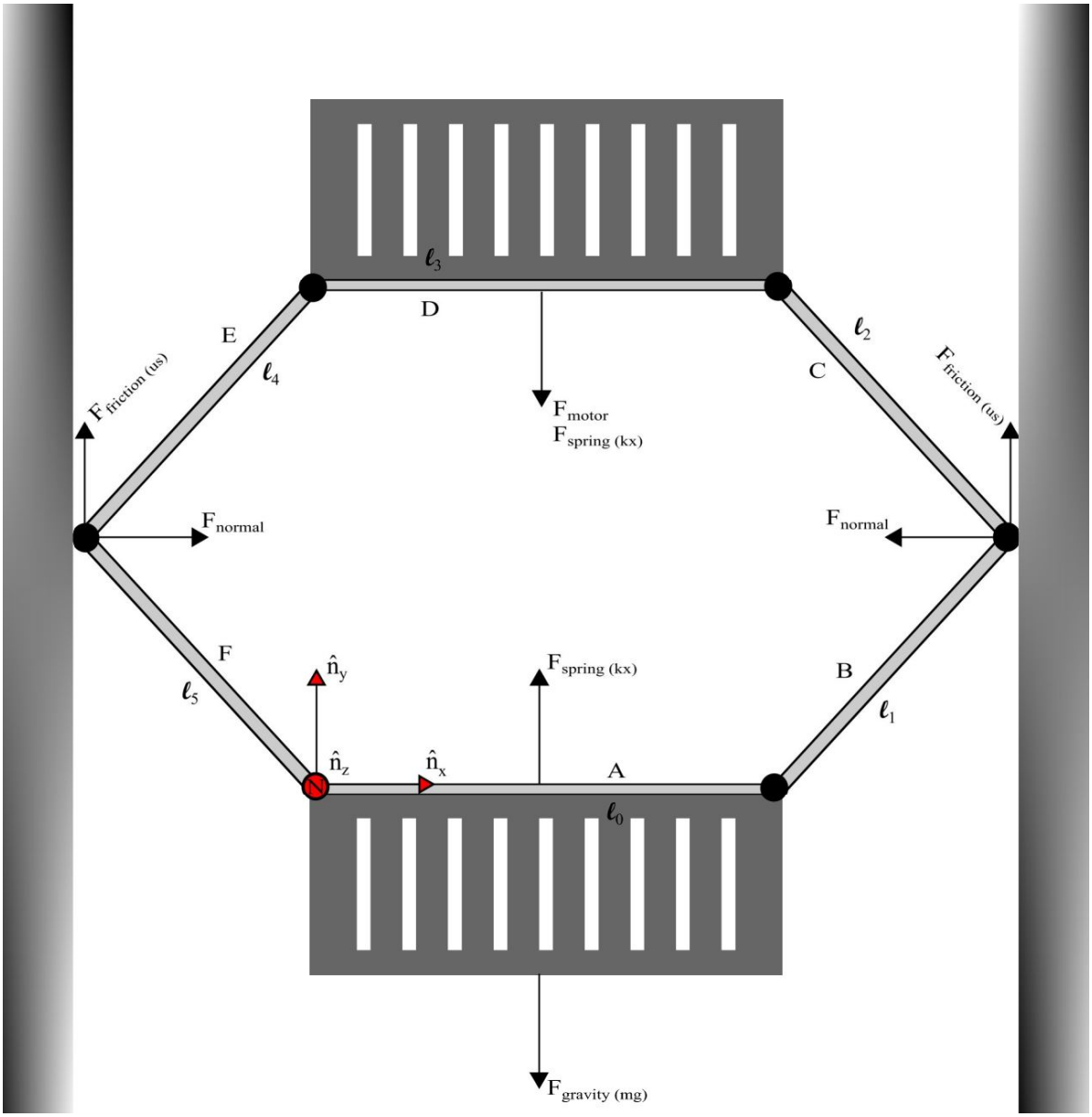


Figure 3b: 2D Free Body Diagram of System

Calculating Youngs Modulus by Carrying out Cantilever Tests

The current plan for our model is to use cardstock as the primary material. Since this is out primary material, cantilever tests were run on cardstock samples to determine obtain the stiffness of the material.

Materials:

1. Clamp
2. Four 125mm x 25mm samples of cardstock
3. At least 5 weights of varying masses
4. Callipers
5. String
6. Tape

The procedure below was provided by Daniel M. Aukes

Procedure:

1. Prepare the samples for testing

> Four near-identicle pieces of cardstock measuring 25mm x 125mm were measured as seen in Figure 4. One of these samples will not be loaded and will be referred to as the base. Once they were cut to length, they were measured and marked 25mm from each end as future reference points. The samples were labeled to keep measurement data consistent and properly organized.

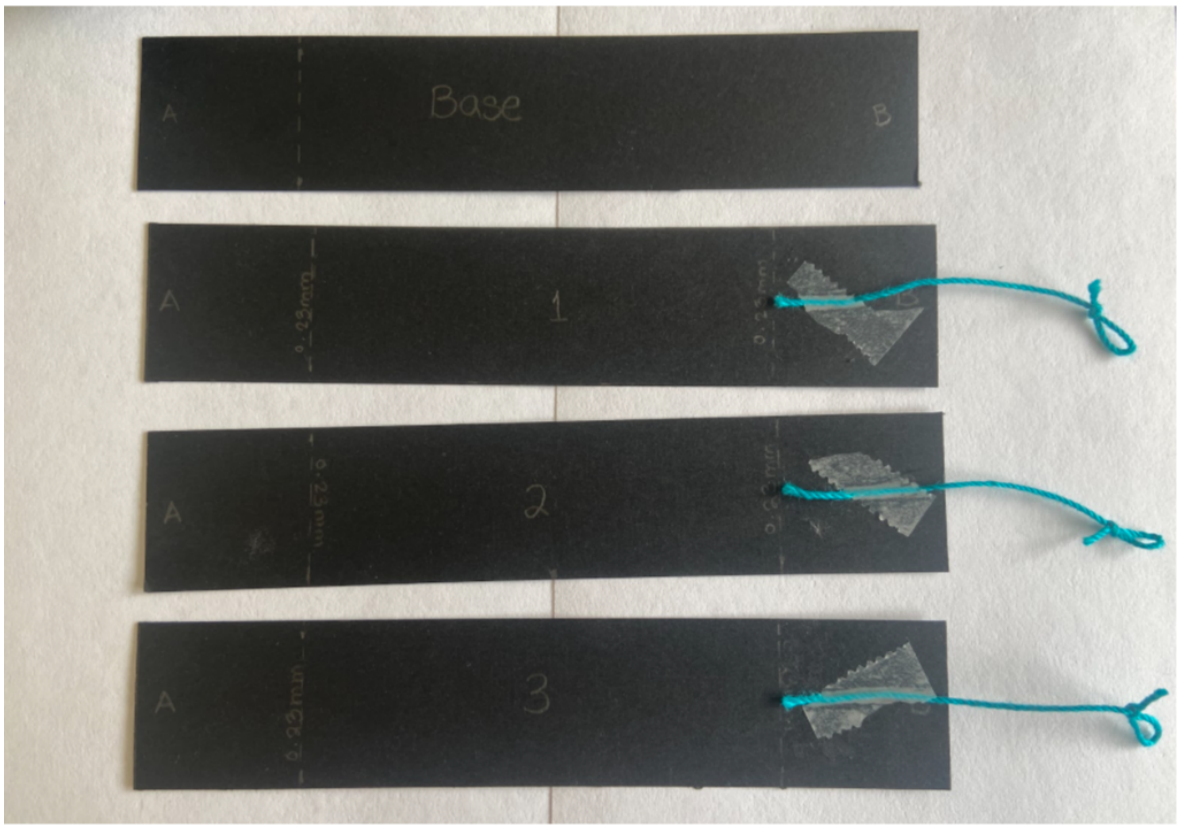


Figure 4: The three 125mm x 25mm samples of cardstock used for the cantilever tests

2. Measure the thickness of each sample in two places

> The thicknesses of each of the samples was suprisingly the same in all locations. While there was a point where it looked like one of the thicknesses was to be 0.01 mm thinner, it ultimately settled on the same value of 0.23 mm (0.00023m) as the rest of the samples. These thicknesses were marked on the samples themselves as can be found in Figure 4, as well as below in Figure 5. Once all of the measurements were completed, strings were added to each of the samples so that masses could later be attached from the loop on the end.

Cardstock Sample Measurements		
Sample	Thickness 25 mm from end A	Thickness 25 mm from end B
Base	0.23	0.23
1	0.23	0.23
2	0.23	0.23
3	0.23	0.23

Figure 5: Measurements of the samples; thickness was measured in two separate places

4. Determine the masses that will be used in the experiments

> For the cantilever experiments, it is required that one has at least five different masses that can be used to test the deflection of the sample. Initially, the masses ranged from 3 - 14g. When it was apparent that the 3g weight was pushing the edge of the mass limit, as the deflection was very high, the masses were scaled down considerably, with hte highest being 2.32g, as can be seen in Figure 6.

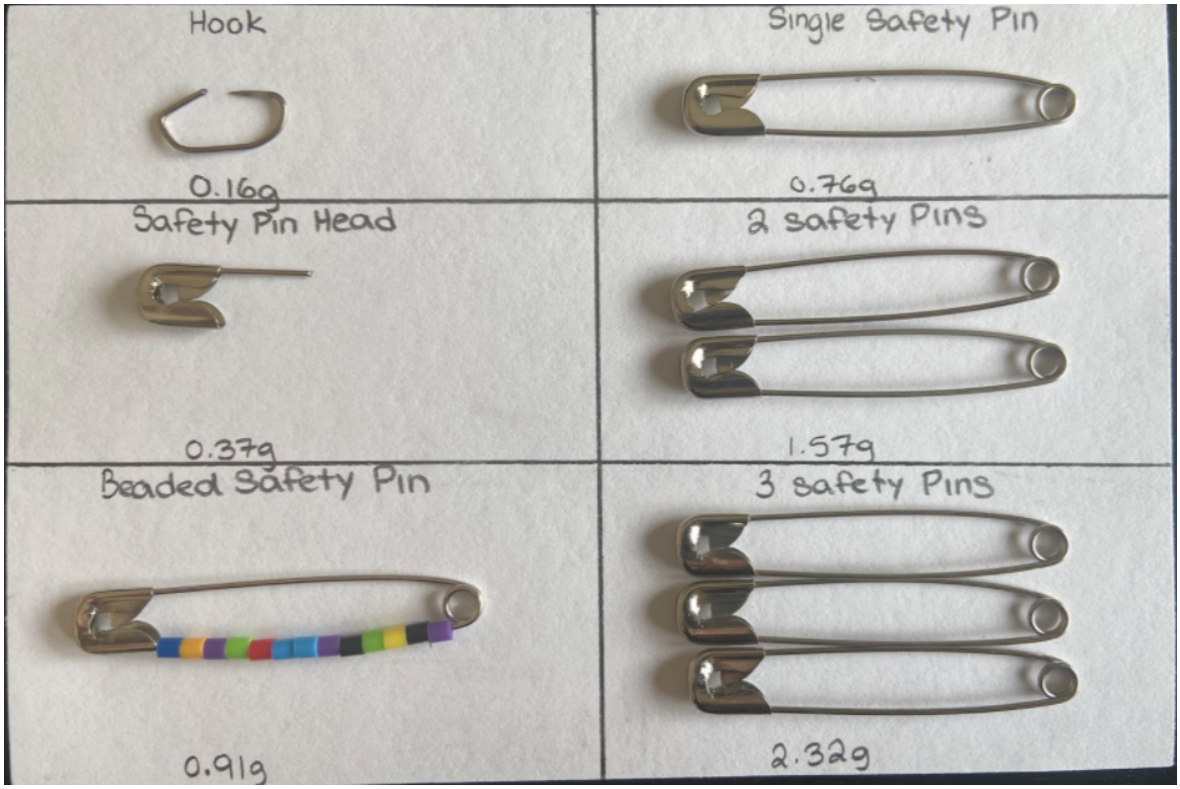


Figure 6: The six masses that were used in the cantilever tests

5. Clamp two samples together on a flat surface, the top layer being the base.

> The base being clamped atop the sample allows for consistent, stable measurements where the sample is not pulled or twisted positionally when loaded. The setup for the tests is seen in Figure 7. From here, the distance between the sample and the base was measured to get an initial defelction based on imperfections in the sample and base.

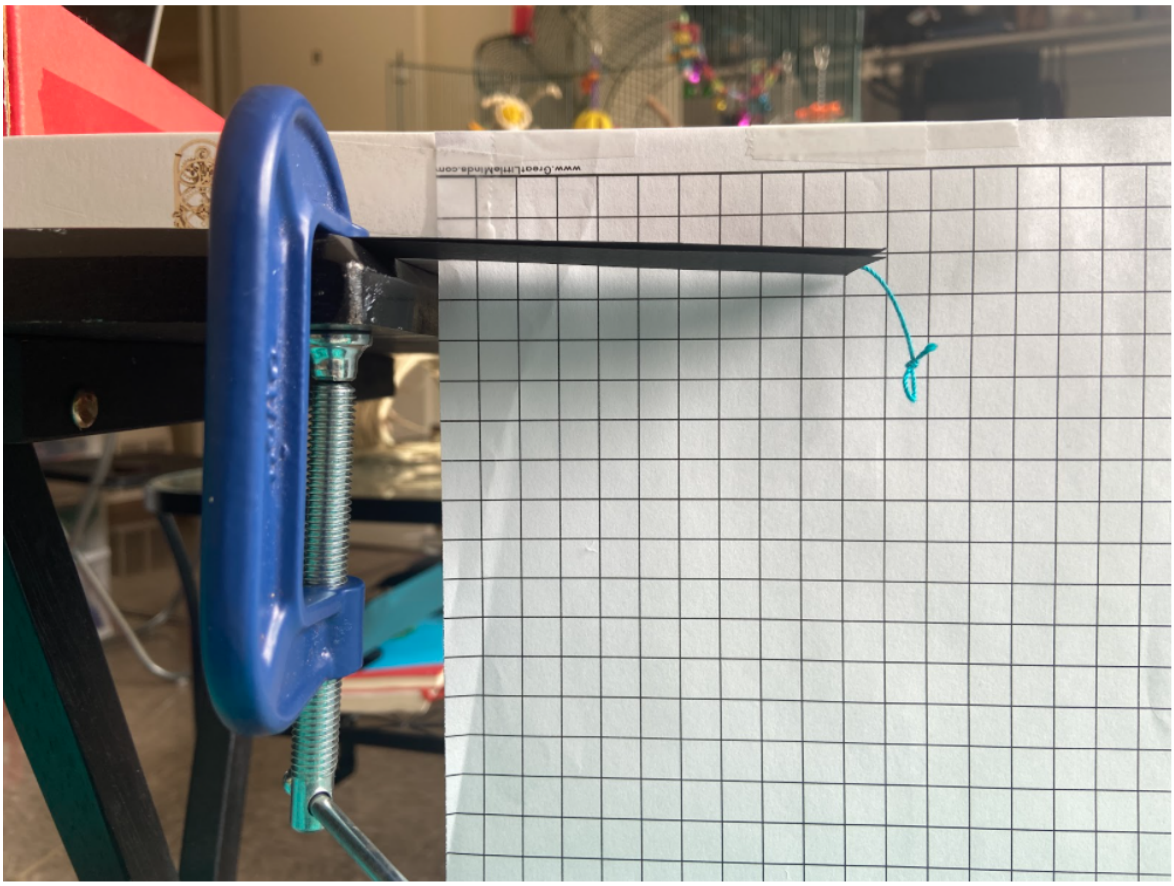


Figure 7: Setup for the cantilever test without a load

6. Set up the camera

> The camera was positioned on top of a stack of books directly in front of the sample to ensure all photos were taken from the same distance and position.

7. Measure the deflection of the sample from the base when sample is under a load

> Each of the six loads were attached to the sample beam, from which the deflection was measured. Once each mass was attached to the sample, a picture was taken, as seen in Figure 8, and the displacement was measured and recorded. All six masses were applied to each of the 3 samples, and the data is recorded as seen in Figure 9. These values remove the initial distance between the base and each sample. A

> graphical representation of the mass in relation to the deflection can be found in Figure 10.

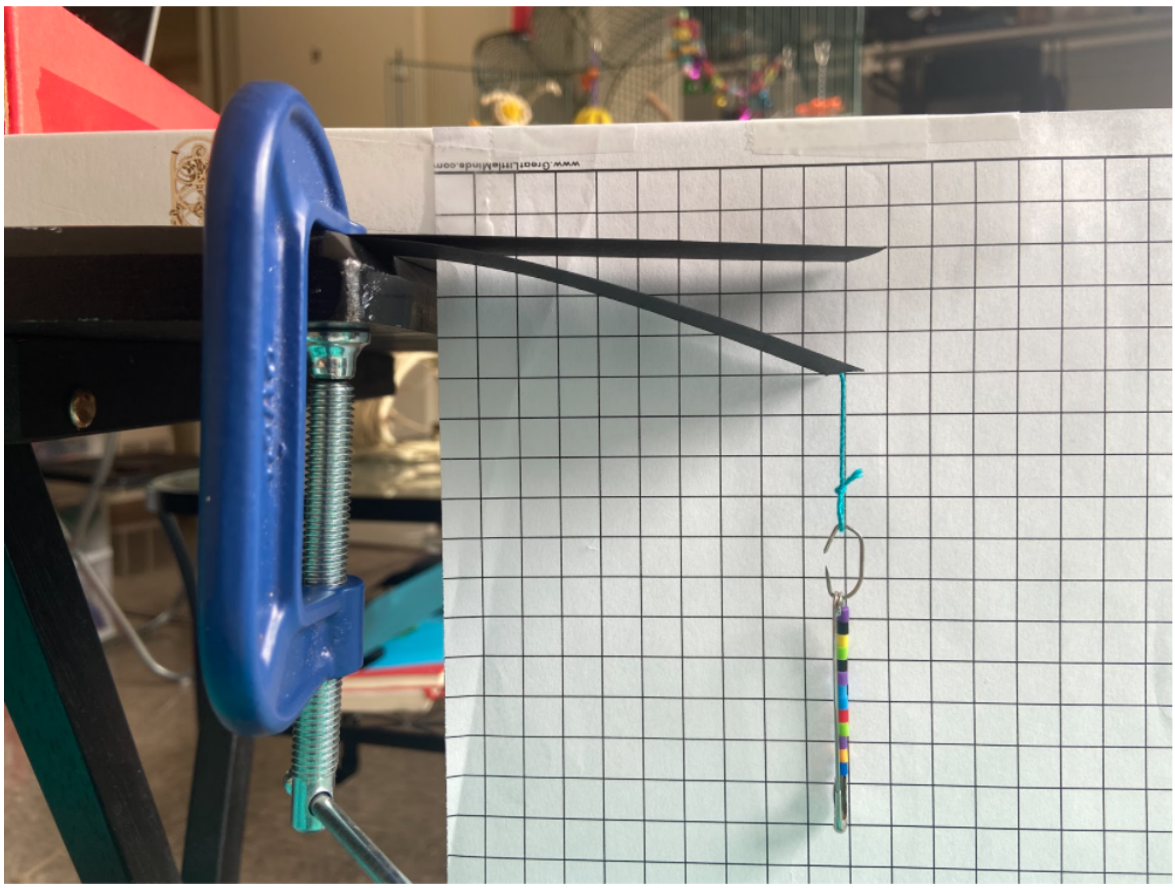


Figure 8: Image of one of the tests run with a load

Cardstock Deflection from the Base						
	Deflection (m) from base without initial deflection					
Sample	1	2	3	4	5	6
1	0.00294	0.01302	0.02371	0.02798	0.04231	0.055
2	0.00302	0.01395	0.02411	0.02903	0.04281	0.05557
3	0.00191	0.01285	0.02422	0.02743	0.04366	0.05779
Average	2.62E-03	1.33E-02	2.40E-02	2.81E-02	4.29E-02	5.61E-02

Figure 9: Deflection measurements of the beam from no-load state

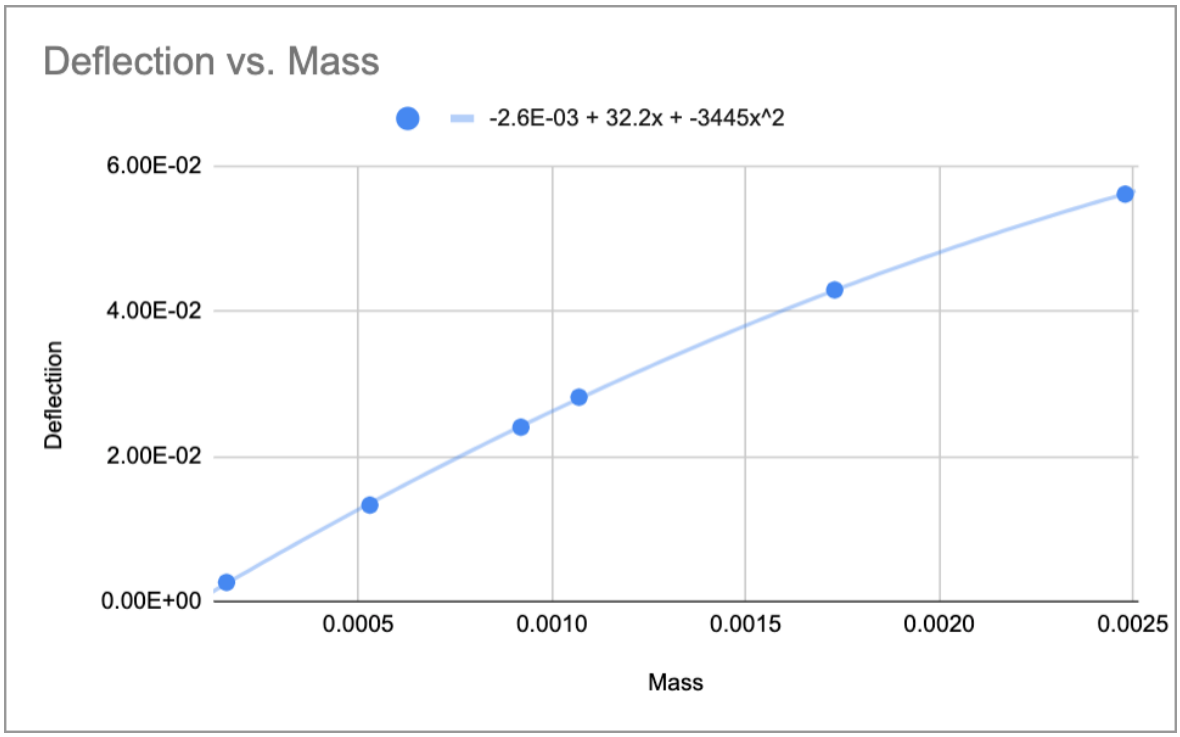


Figure 10: Graphical representation of the deflection versus mass graph

8. Calculate the Youngs Modulus value (E)

> Once the deflection of the samples under each load has been measured, it can be seen that all but one of the variables in the beam equations found in Figure 11 have been found. These can be rearranged as in Figure 12 to find Youngs Modulus (E), using the equation in Figure 13 to find I. Calculating hte Youngs modulus, the values of which can be found in Figure 14, results in the overall stiffness of the material.

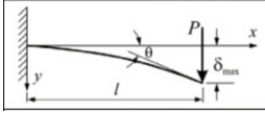
BEAM TYPE	SLOPE AT FREE END	DEFLECTION AT ANY SECTION IN TERMS OF x	MAXIMUM DEFLECTION
1. Cantilever Beam – Concentrated load P at the free end			
	$\theta = \frac{Pl^2}{2EI}$	$y = \frac{Px^2}{6EI}(3l - x)$	$\delta_{max} = \frac{Pl^3}{3EI}$

Figure 11: Equations provided by Daniel M. Aukes for calculating the deflection of the beam

$$E = \frac{PL^3}{3dI}$$

Figure 12: Rearranged equation set to find Youngs Modulus (E)

$$I = \frac{bh^3}{12}$$

Figure 13: Variable I

Load	1	2	3	4	5	6	Average
Youngs Modulus (E)	9.76E+03	6.39E+03	6.13E+03	6.08E+03	6.45E+03	7.07E+03	6.98E+03

Figure 14: The Youngs Modulus (E) values calculated from each of the masses

Approximating Compliant Beams

Using code provided by Daniel M. Aukes The code below uses the beam deflection equations found in Figures 11-13 to initally find the cross sectional moment of intertia. From there, the code allows for the determination of the value of x that results in an accurate determination of the deflection and angle of a beam with a single joint.

```
#The following code is heavily based on that created by Daniel M. Aukes
import sympy
q = sympy.Symbol('q')
d = sympy.Symbol('d')
L = sympy.Symbol('L')
P = sympy.Symbol('P')
h = sympy.Symbol('h')
b = sympy.Symbol('b')
E = sympy.Symbol('E')
x = sympy.Symbol('x')

subs = {}
```

```
subs = {}
#subs[k]=1000
subs[P]=.00016
subs[L]=.1
subs[b]=.025
subs[h]=.1
subs[E]=7.0818
subs[x]=.5
```

```
I = b*h**3/12
d1 = P*L**3/3/E/I
d1.subs(subs)
```

0.00361490016662431

```
q1 = P*L**2/2/E/I
q1.subs(subs)
```

0.0542235024993646

```
k1 = P*L*(1-x)/(sympy.asin(P*L**2/(3*E*I*(1-x))))
k1.subs(subs)
```

0.000110556584514503

```
d2 = L*(1-x)*sympy.sin(P*L*(1-x)/k1)
d2.subs(subs)
```

0.00361490016662431

```
q2 = P*L*(1-x)/k1
q2.subs(subs)
```

0.0723611355680998

```
k2 = 2*E*I*(1-x)/(L)
q3 = P*L*(1-x)/k2
q3.subs(subs)
```

0.0542235024993646

```
d2 = P*L*(1-x)*sympy.sin(P*L*(1-x)/k1)/k1
```

```
d3 = L*(1-x)*sympy.sin(P*L*(1-x)/k2)
d1.subs(subs)
```

$$0.00361490016662431$$

```
d3.subs(subs)
```

$$0.00270984675940322$$

```
del subs[x]
error = []
error.append(d1-d2)
error.append(q1-q2)
error= sympy.Matrix(error)
error = error.subs(subs)
error
```

$$\begin{bmatrix} 0 \\ 0.0542235024993646 - \operatorname{asin}\left(\frac{0.036149001666243}{1-x}\right) \end{bmatrix}$$

```
import scipy.optimize
f = sympy.lambdify((x),error)

def f2(args):
    a = f(*args)
    b = (a**2).sum()
    return b

sol = scipy.optimize.minimize(f2,[.25])
sol
```

```
      fun: 3.9928096992175975e-11
 hess_inv: array([[72.33682164]])
      jac: array([-1.02805959e-06])
  message: 'Optimization terminated successfully.'
     nfev: 24
      nit: 5
     njev: 8
   status: 0
  success: True
        x: array([0.33292887])
```

```
subs[x]=sol.x[0]
d2.subs(subs)
q2.subs(subs)
```

$$0.0542171836310331$$

CALCULATE DAMPING COEFFICIENT USING MOTION CAPTURE

The aim of the experiment is to track motion of the joint at pBC (shown in figure above) using the software called 'Tracker' and to collect data from it's motion. The plot of time vs Y-axis is observed and analyzed. The data is then used in a python program to generate plots and calculate the damping coefficient.

EXPERIMENT SETUP

Aim:

To create a 3D model of the sarrus link and capture its motion on video.

Materials used:

1. Cardstock
2. Scissors
3. Tape
4. Glue
5. Coloured paper (orange)
6. Plain paper
7. Smartphone with video capability
8. Black marker
9. Pencil
10. Keyboard mouse

Preparation:

1. Cut the cardstock into a 9.5 x 8 inch sheet, with a 2 inch width as shown in figure 2
2. Cut the orange coloured paper into three small squares.
3. Mark the sheet as shown in white lines in the figure 2.

Procedure:

1. Bend the sheet at the marked points.
2. Join the ends of the cutout that is 0.5inch, with tape, to complete a 3D model of the sarrus link.
3. Using the black marker, circle in one piece of colored square.
4. Glue one orange square on point pBC (shown in figure above)
5. Stick the remaining two squares below the prototype in a straight line like the points pNA and pNF, to create the base frame.
6. Attach the system to plain paper by taping them together.
7. Fix the apparatus on the table with tape. See figure 3.
8. Release the keyboard mouse on top of the 3D sarrus link, so as to capture the movement of the system.
9. Capture this motion by taking a video.

ASSUMPTIONS

1. The base frame is fixed and rigid.
2. The links are rigid.
3. The joints have the same stiffness and so all results obtained at one joint applies to all others.
4. Friction between keyboard mouse and cardstock is neglected.
5. Translation of the joint in the x axis is neglected.

RESULTS

1. The captured video is run through the software, 'Tracker'. The point mass is placed at the joint pBC.
2. The distance between the points in the base frame is measured to be 10cm.
3. The distance measured between the lens of the camera and the experiment setup is 30cm.

4. Since the joint was moved with an object falling on top of it, there is damping observed in the data plotted.
5. The damping is shown by the plot time vs y-axis.
6. The plot shown in Fig 1.b is based on the x and y values obtained by the tracker

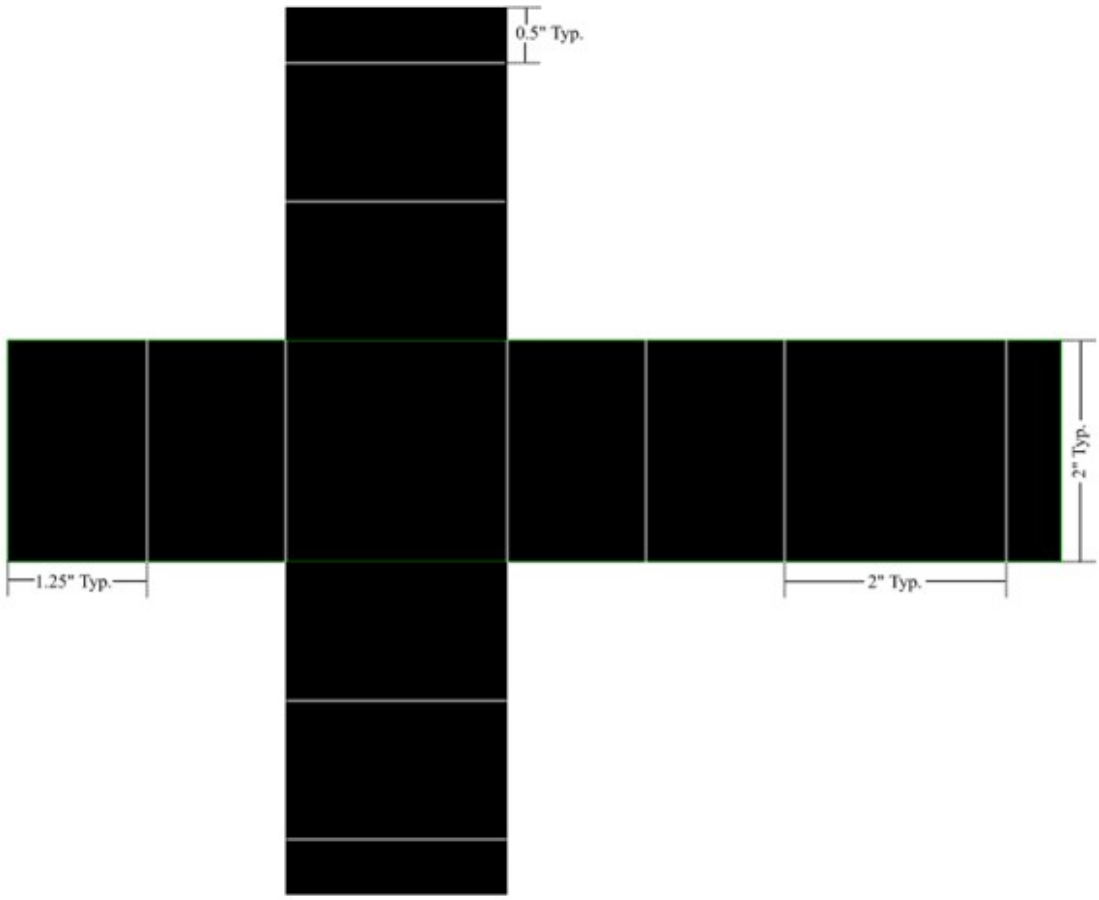


Figure 2: 2D schematic of 3D Sarrus system

Figure 3: The setup of the experiment

Figure 4: The plot of the point mass placed at pBC, which is the position of the joint that was tracked. The plot shows time vs y-axis.

Note: You must upload the file named "3.csv" for the code to continue running.

```
import pandas as pd
import numpy
import matplotlib.pyplot as plt
import scipy.interpolate as si
from math import pi, sqrt
```

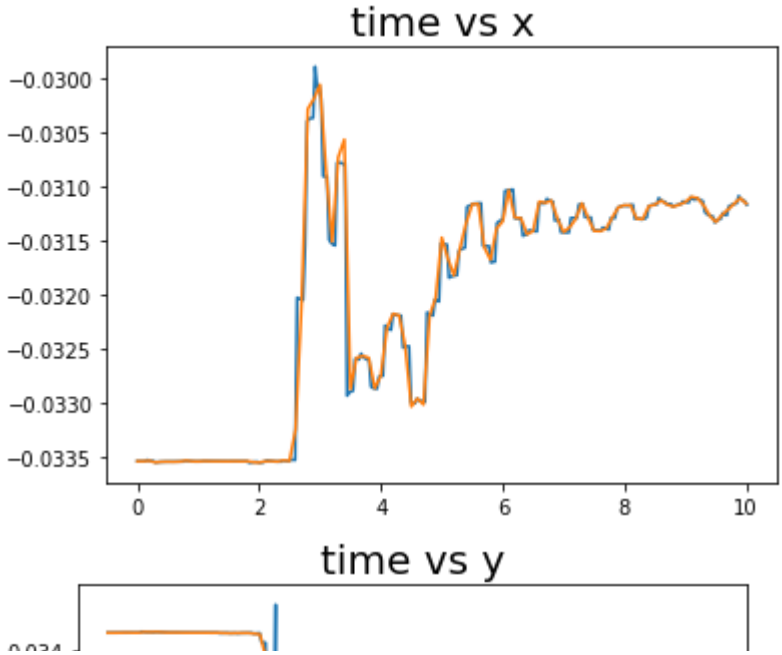
```
from math import pi,sqrt
import math


from google.colab import files
uploaded = files.upload()
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving 3.csv to 3.csv
```

```
import io
df2 = pd.read_csv(io.BytesIO(uploaded['3.csv']))

x = df2.x.to_numpy()
y = df2.y.to_numpy()
t = df2.t.to_numpy()


xy = numpy.array([x,y]).T
f = si.interp1d(t,xy.T,fill_value='extrapolate',kind='quadratic')
new_t = numpy.r_[0:t[-1]:.1]

plt.figure()
plt.title('time vs x',fontsize = 20)
plt.plot(t,x)
plt.plot(new_t,f(new_t)[0])

plt.figure()
plt.title('time vs y',fontsize = 20)
plt.plot(t,y)
plt.plot(new_t,f(new_t)[1])
```

```
[<matplotlib.lines.Line2D at 0x7f1f8a5c5590>]
```



After getting the experimental results, the data from the plot above is analyzed and used for calculating the damping coefficient as shown below.

```
Td = 2.133/4
Wd = (2*pi)*Td
x0 = 0.03238
xn = 0.03102
n = 3
delta = (1/n)*(numpy.log(x0/xn))
seta = delta/(sqrt((4*pi**2)+(delta**2)))
Wn = Wd/sqrt(1 - seta**2)
k = 200.45
m = k/Wn**2
b = 2*seta*sqrt(k*m)

print('Damped period of motion is - ',Td)
print('\nDamped natural frequency is - ',Wd)
print('\nDamping ratio - ',seta)
print('\nNatural frequency - ',Wn)
print('\nSystem mass - ',m)
print('\nDamping coefficient - ',b)
```

```
Damped period of motion is -  0.53325

Damped natural frequency is -  3.3505085650535142
```

```
Damping ratio -  0.0022763761387949535

Natural frequency -  3.350517246067861

System mass -  17.855924673382383

Damping coefficient -  0.2723756145753064
```

REFERENCE

"Basic Tracker Tutorial", Daniel Aukes, Foldable Robotics [Online]. Available: https://egr557.github.io/modules/validation/Tracker%20tutorial.html [Accessed: 07- Mar-2021]

Pynamics Code

The following lines of code will initialize the code and import all of the necessary packages needed to run pynamics:

```
%matplotlib inline
```

```
#This block of code imports the necessary modules created by Daniel M.
import pynamics
from pynamics.frame import Frame
from pynamics.variable_types import Differentiable,Constant
from pynamics.system import System
from pynamics.body import Body
from pynamics.dyadic import Dyadic
from pynamics.output import Output,PointsOutput
from pynamics.particle import Particle
import pynamics.integration
import numpy
import matplotlib.pyplot as plt
plt.ion()
from math import pi
```

```
#This block of code create a new system object and set that system as t
```

```
system = System()
pynamics.set_system(__name__,system)
```

The following lines of code will create and store the following constants and variables of the system: frame lengths, frame mass, coefficient of gravity, damping coefficient, spring coefficient, spring preloads (defines initial spring position), and moment of inertia of each frame. All of these constants and variables are in SI units. The constants derived from the methods described above are implemented in this section of the code.

```
#This block of code declares and store constants in SI units
l0 = Constant(0.0508,'l0',system) #defines the lengths (in meters) of e
l1 = Constant(0.0254,'l1',system) #1in ~ 0.0254m
l2 = Constant(0.0254,'l2',system)
l3 = Constant(0.0508,'l3',system)
l4 = Constant(0.0254,'l4',system)
l5 = Constant(0.0254,'l5',system)

mA = Constant(0.02,'mA',system) #defines the mass (in kg) of each frame
mB = Constant(0.01,'mB',system) #1g ~ 0.001kg
mC = Constant(0.01,'mC',system)
mD = Constant(0.02,'mD',system)
mE = Constant(0.01,'mE',system)
mF = Constant(0.01,'mF',system)

g = Constant(9.81,'g',system) #defines gravity (in m/s^2)
b = Constant(0.965,'b',system)  #defines damping coefficient (in kg/s^2
k = Constant(0.1,'k',system)  #defines spring coefficient (N/m^2)

preload0 = Constant(0*pi/180,'preload0',system) #defines the spring pre
preload1 = Constant(0*pi/180,'preload1',system)
preload2 = Constant(0*pi/180,'preload2',system)
preload3 = Constant(0*pi/180,'preload3',system)
preload4 = Constant(0*pi/180,'preload4',system)
preload5 = Constant(0*pi/180,'preload5',system)
preload6 = Constant(0*pi/180,'preload6',system)

Ixx_A = Constant(2,'Ixx_A',system) #defines the inertia (kg*m^2) of eac
Iyy_A = Constant(2,'Iyy_A',system)
Izz_A = Constant(2,'Izz_A',system)
Ixx_B = Constant(1,'Ixx_B',system)
Iyy_B = Constant(1,'Iyy_B',system)
Izz_B = Constant(1,'Izz_B',system)
```

```
Izz_B = Constant(1,'Izz_B',system)
Ixx_C = Constant(1,'Ixx_C',system)
Iyy_C = Constant(1,'Iyy_C',system)
Izz_C = Constant(1,'Izz_C',system)
Ixx_D = Constant(2,'Ixx_D',system)
Iyy_D = Constant(2,'Iyy_D',system)
Izz_D = Constant(2,'Izz_D',system)
Ixx_E = Constant(1,'Ixx_E',system)
Iyy_E = Constant(1,'Iyy_E',system)
Izz_E = Constant(1,'Izz_E',system)
Ixx_F = Constant(1,'Ixx_F',system)
Iyy_F = Constant(1,'Iyy_F',system)
Izz_F = Constant(1,'Izz_F',system)
```

The following lines of code define precision of integration and the length/steps of the pynamics animation.

```
#This block of code specifies the precision of the integration
tol = 1e-12


#This block of code defines variables for time that can be used through
tinitial = 0
tfinal = 10
fps = 30
tstep = 1/fps
t = numpy.r_[tinitial:tfinal:tstep]
```

The following line of code creates dynamic varibales which will be used whenever the system changes over time.

```
#This block of code creates dynamic state variables for the angles show
q0,q0_d,q0_dd = Differentiable('q0',system) #angle between N and A fram
q1,q1_d,q1_dd = Differentiable('q1',system) #angle between A and B fram
q2,q2_d,q2_dd = Differentiable('q2',system) #angle between B and C fram
q3,q3_d,q3_dd = Differentiable('q3',system) #angle between C and D fram
q4,q4_d,q4_dd = Differentiable('q4',system) #angle between D and E fram
q5,q5_d,q5_dd = Differentiable('q5',system) #angle between E and F fram
```

The following lines of code defines the reference frames of the system.

```
#This block of code initializes frames
N = Frame('N')
A = Frame('A')
B = Frame('B')
C = Frame('C')
D = Frame('D')
E = Frame('E')
F = Frame('F')


#This block of code sets N frame as the newtonian frame (see kinematic
system.set_newtonian(N)
```

The following line of code sets the rotation of each frame relative to the other frames in the system.

```
#This block of code shows frame rotation in the Z direction
A.rotate_fixed_axis_directed(N,[0,0,1],q0,system)  #the A frame rotates
B.rotate_fixed_axis_directed(A,[0,0,1],q1,system)  #the B frame rotates
C.rotate_fixed_axis_directed(B,[0,0,1],q2,system)  #the C frame rotates
D.rotate_fixed_axis_directed(C,[0,0,1],q3,system)  #the D frame rotates
E.rotate_fixed_axis_directed(D,[0,0,1],q4,system)  #the E frame rotates
F.rotate_fixed_axis_directed(E,[0,0,1],q5,system)  #the F frame rotates
```

The following lines of code defines the vectors that make up the system and their cooresponding center of mass.

```
#This block of code defines the points needed to create the mechanism
pNA = 0*N.x + 0*N.y    #pNA (point NA) position is 0 units in the direc
pAB = pNA + l0*A.x     #pAB position is pNA's position plus l0 units in
pBC = pAB + l1*B.x     #pBC position is pAB's position plus l1 units in
pCD = pBC + l2*C.x     #pCD position is pBC's position plus l2 units in
pDE = pCD + l3*D.x     #pDE position is pCD's position plus l3 units in
pEF = pDE + l4*E.x     #pEF position is pDE's position plus l4 units in
pFtip = pEF + l5*F.x   #pFtip position is pEF's position plus l5 units


#This block of code defines the centers of mass of each link (halfway a
pAcm=pNA+l0/2*A.x  #pA (link A) position is pNA's position plus one hal
pBcm=pAB+l1/2*B.x  #pB (link B) position is pAB's position plus one hal
```

```
pBcm pAB+l1/2*B.x  #pB (link B) position is pAB's position plus one hal
pCcm=pBC+l2/2*C.x  #pC (link C) position is pBC's position plus one hal
pDcm=pCD+l3/2*D.x  #pD (link D) position is pCD's position plus one hal
pEcm=pDE+l4/2*E.x  #pE (link E) position is pDE's position plus one hal
pFcm=pEF+l5/2*F.x  #pF (link F) position is pEF's position plus one hal


#This block of code defines the points in the system in order (counter-
points = [pNA,pAB,pBC,pCD,pDE,pEF,pFtip]
```

The following lines of code defines the initial angle values taht make up the initial guess for the system. The values stored are then stored as a list.

```
#This block of code sets the initial guess for the mechanisms starting 
initialvalues = {}
initialvalues[q0] = 0*pi/180  #optimal angle is 0
initialvalues[q0_d] = 0*pi/180
initialvalues[q1] = 45*pi/180  #optimal angle is 45
initialvalues[q1_d] = 0*pi/180
initialvalues[q2] = 90*pi/180  #optimal angle is 90
initialvalues[q2_d] = 0*pi/180
initialvalues[q3] = 45*pi/180   #optimal angle is 45
initialvalues[q3_d] = 0*pi/180
initialvalues[q4] = 45*pi/180   #optimal angle is 45
initialvalues[q4_d] = 0*pi/180
initialvalues[q5] = 90*pi/180   #optimal angle is 90
initialvalues[q5_d] = 0*pi/180


#This block of code orders the initial values in a list in such a way t
statevariables = system.get_state_variables()
ini = [initialvalues[item] for item in statevariables]


#This block of code declares the independent variable and dependent var
qi = [q1]
qd = [q0,q2,q3,q4,q5]


#This block of code reformats constants
constants = system.constant_values.copy()
defined = dict([(item,initialvalues[item]) for item in qi])
constants.update(defined)
```

The following lines of code define the angular velocities and moment of inertias of the systems. It influences the physics of the system by determining how the system moves under specific loads.

```
#This block of code computes and returns the angular velocity between f
wNA = N.getw_(A)
wAB = A.getw_(B)
wBC = B.getw_(C)
wCD = C.getw_(D)
wDE = D.getw_(E)
wEF = E.getw_(F)
```

```
#This block of code compute the inertia dynamics of each body and defin
IA = Dyadic.build(A,Ixx_A,Iyy_A,Izz_A)
IB = Dyadic.build(B,Ixx_B,Iyy_B,Izz_B)
IC = Dyadic.build(C,Ixx_C,Iyy_C,Izz_C)
ID = Dyadic.build(D,Ixx_D,Iyy_D,Izz_D)
IE = Dyadic.build(E,Ixx_E,Iyy_E,Izz_E)
IF = Dyadic.build(F,Ixx_F,Iyy_F,Izz_F)

BodyA = Body('BodyA',A,pAcm,mA,IA,system)
BodyB = Body('BodyB',B,pBcm,mB,IB,system)
BodyC = Body('BodyC',C,pCcm,mC,IC,system)
BodyD = Body('BodyD',D,pDcm,mD,ID,system)
BodyE = Body('BodyE',E,pEcm,mE,IE,system)
BodyF = Body('BodyF',F,pFcm,mF,IF,system)
```

The following lines of code add forces to the system.

```
#This block of code adds spring forces
#The first value is the linear spring constant
#The second value is the “stretch” vector, indicating the amount of def
#The final parameter is the linear or angluar velocity vector (dependin
system.add_spring_force1(k,(q0-preload1)*N.z,wNA)
system.add_spring_force1(k,(q1-preload2)*A.z,wAB)
system.add_spring_force1(k,(q2-preload3)*B.z,wBC)
system.add_spring_force1(k,(q3-preload4)*C.z,wCD)
system.add_spring_force1(k,(q4-preload5)*E.z,wDE)
system.add_spring_force1(k,(q5-preload6)*F.z,wEF)
```

```
(<pynamics.force.Force at 0x7f1f89fcb290>,
 <pynamics.spring.Spring at 0x7f1f89fcbed0>)
```

```
#This block of code adds forces and torques to the system with the gener
#The first parameter supplied is a vector describing the force applied a
#The second parameter is the vector describing the linear speed (for an
system.addforce(-b*wNA,wNA)
system.addforce(-b*wAB,wAB)
system.addforce(-b*wBC,wBC)
system.addforce(-b*wCD,wCD)
system.addforce(-b*wDE,wDE)
system.addforce(-b*wEF,wEF)
```

```
<pynamics.force.Force at 0x7f1f8a037cd0>
```

```
#This block of code globally applies the force of gravity to all partic
system.addforcegravity(-g*N.y)
```

The following lines of code set the constarints for the system.

```
#This block of code defines the closed loop kinematics (vectors) of the
#Constraint 1:
eq_vector=pFtip-pNA
#Constraint 2:
eq_vector2= pDE-pCD
#Constraint 3:
eq_vector3= pEF-pBC
#Constraint 4:
eq_vector4= pDE-pNA
```

```
#This block of code defines the systems constraints based on the vector
eq = []

#pFtip and pNA have to be on the same point
eq.append((eq_vector).dot(N.x))
eq.append((eq_vector).dot(N.y))

#pDE and pCD must have the same y coordinate in the N frame
eq.append((eq_vector2).dot(N.y))
#pEF and pBC must have the same y coordinate in the N frame
eq.append((eq_vector3).dot(N.y))
```

```
#pDE and pNA must have the same x coordinate in the N frame
eq.append((eq_vector4).dot(N.x))


eq_d=[(system.derivative(item)) for item in eq]
eq_dd=[(system.derivative(item)) for item in eq_d]
```

The following lines of code create figures and animations of the system.

```
#This block of code calculates the symbolic expression for F and ma
f,ma = system.getdynamics()
```

```
    2021-03-20 06:54:58,998 - pynamics.system - INFO - getting dynamic
```

```
#This block of code solves the system of equations F=ma plus any constr
#It returns one or two variables.
#func1 is the function that computes the velocity and acceleration give
#lambda1(optional) supplies the function that computes the constraint f
#The below function inverts the mass matrix numerically every time step
func1,lambda1 = system.state_space_post_invert(f,ma,eq_dd,return_lambda
```

```
    2021-03-20 06:55:00,903 - pynamics.system - INFO - solving a = f/m
    2021-03-20 06:55:00,931 - pynamics.system - INFO - substituting cc
    2021-03-20 06:55:14,755 - pynamics.system - INFO - done solving a
    2021-03-20 06:55:14,757 - pynamics.system - INFO - calculating fun
```

```
#This block of code integrates the function calculated above
states=pynamics.integration.integrate(func1,ini,t,rtol=tol,atol=tol, ar
```

```
    2021-03-20 06:55:14,783 - pynamics.integration - INFO - beginning
    2021-03-20 06:55:14,784 - pynamics.system - INFO - integration at
    2021-03-20 06:55:22,561 - pynamics.integration - INFO - finished i
```

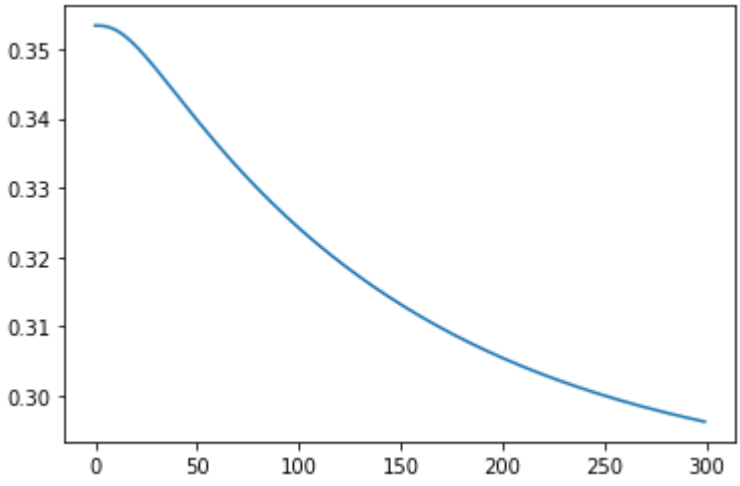
```
#This block of code calculates and plots a variety of data from the pre
plt.figure()
artists = plt.plot(t,states[:,:6])
plt.legend(artists,['q0','q1','q2', 'q3', 'q4', 'q5'])
```

```
<matplotlib.legend.Legend at 0x7f1f89224990>
```

A graph of the energy shows the energy high when the motor is contracting the spring and slowly decreasing as the spring is released.

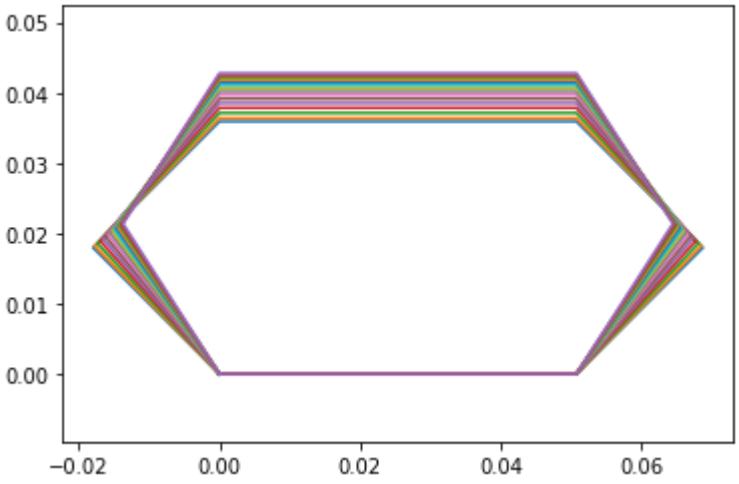
```
#This block of code calculates amd plots the energy of system
KE = system.get_KE()
PE = system.getPEGravity(pNA) - system.getPESprings()
energy_output = Output([KE-PE],system)
energy_output.calc(states)
energy_output.plot_time()
```

```
2021-03-20 06:55:23,159 - pynamics.output - INFO - calculating out
2021-03-20 06:55:23,331 - pynamics.output - INFO - done calculatin
```



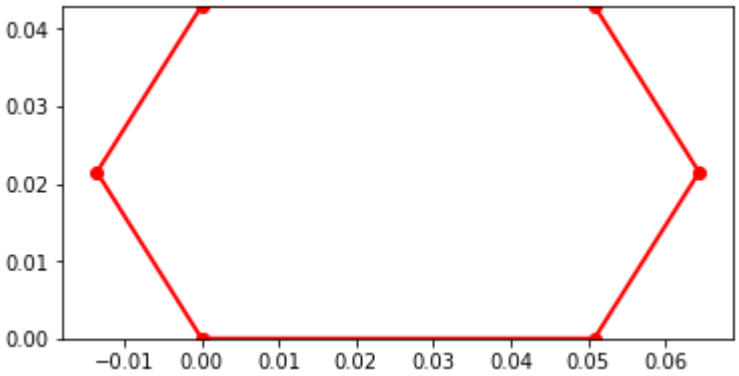

```
#This block of code calculates and plots the motion of the system
points = [pNA,pAB,pBC,pCD,pDE,pEF,pNA]
points_output = PointsOutput(points,system)
y = points_output.calc(states)
points_output.plot_time(20)
```

```
2021-03-20 06:55:23,595 - pynamics.output - INFO - calculating out
2021-03-20 06:55:23,673 - pynamics.output - INFO - done calculatin
<matplotlib.axes._subplots.AxesSubplot at 0x7f1f89119150>
```




```
#This block of code produces a figure of the animations end position us
points_output.animate(fps = fps,movie_name = 'render.mp4',lw=2,marker='
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f1f890b3bd0>
```



An animation of the sarrus link shows the motor contracting the spring and then slowly releasing it. The sarrus link starts contrcated and touches the outer walls. As the motor releases the spring, the sarrus link will extend vertically.

```
#This block of code animates the figure above
from matplotlib import animation, rc
from IPython.display import HTML
HTML(points_output.anim.to_html5_video())
```


0:00 / 0:10