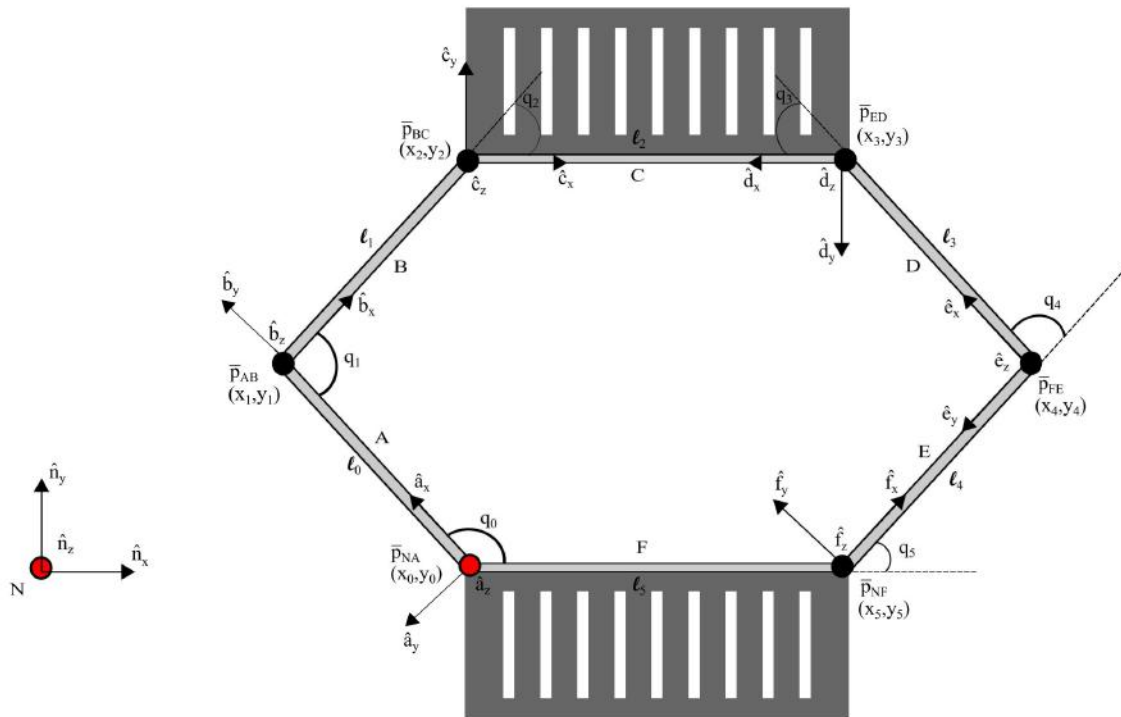


▼ Dynamics I

```
!pip install pypoly2tri idealab_tools foldable_robotics dynamics
```

```
Collecting pypoly2tri
  Downloading https://files.pythonhosted.org/packages/11/33/079f13c
Collecting idealab_tools
  Downloading https://files.pythonhosted.org/packages/c7/00/d34a53c
Collecting foldable_robotics
  Downloading https://files.pythonhosted.org/packages/5d/af/a1cef6c
Collecting dynamics
  Downloading https://files.pythonhosted.org/packages/4e/ef/c2d877c
    |████████████████████████████████████████| 92kB 4.3MB/s
Requirement already satisfied: imageio in /usr/local/lib/python3.7
Collecting ezdxf
  Downloading https://files.pythonhosted.org/packages/6c/67/1a6715c
    |████████████████████████████████████████| 1.8MB 13.3MB/s
Requirement already satisfied: shapely in /usr/local/lib/python3.7
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7
Requirement already satisfied: pyyaml in /usr/local/lib/python3.7/
Requirement already satisfied: numpy 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: pyparsing>=2.0.1 in /usr/local/lib/
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/pytho
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/pytho
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7
Installing collected packages: pypoly2tri, idealab-tools, ezdxf, fo
Successfully installed ezdxf-0.15.2 foldable-robotics-0.0.29 idealab-
```

▼ Dynamics Model



```
%matplotlib inline
```

```
use_constraints = False
```

```
#This block of code imports the necessary modules created by Daniel M. 7
```

```
import dynamics
```

```
from dynamics.frame import Frame
```

```
from dynamics.variable_types import Differentiable,Constant
```

```
from dynamics.system import System
```

```
from dynamics.body import Body
```

```
from dynamics.dyadic import Dyadic
```

```
from dynamics.output import Output,PointsOutput
```

```
from dynamics.particle import Particle
```

```
import dynamics.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 tl
```

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

1. Scale: Ensure your system is using SI units. You should be specifying lengths in meters (so millimeters should be scaled down to the .001 range), forces in Newtons, and radians (not degrees), and masses in kg. You may make educated guesses about mass for now.

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

mA = Constant(0.01,'mA',system) #defines the mass (in kg) of each frame
mB = Constant(0.01,'mB',system) #lg ~ 0.001kg
mC = Constant(0.1,'mC',system)
mD = Constant(0.01,'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(1e-1,'b',system) #defines damping coefficient (in kg/s^2)
k = Constant(0e-1,'k',system) #defines spring coefficient (kg/s^2)

preload0 = Constant(135*pi/180,'preload0',system) #defines the spring p
preload1 = Constant(-90*pi/180,'preload1',system)
preload2 = Constant(-45*pi/180,'preload2',system)
preload3 = Constant(45*pi/180,'preload3',system)
preload4 = Constant(90*pi/180,'preload4',system)
preload5 = Constant(45*pi/180,'preload5',system)

Ixx_A = Constant(1,'Ixx_A',system) #defines the inertia (kg*m^2) of each
Iyy_A = Constant(1,'Iyy_A',system)
Izz_A = Constant(1,'Izz_A',system)
Ixx_B = Constant(1,'Ixx_B',system)
Iyy_B = Constant(1,'Iyy_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(1,'Ixx_D',system)
Iyy_D = Constant(1,'Iyy_D',system)
Izz_D = Constant(1,'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)

```

```

#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 througho
tinitial = 0
tfinal = 10
fps = 30
tstep = 1/fps
t = numpy.r_[tinitial:tfinal:tstep]

```

```

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

```

```

#This block of code sets the initial guess for the mechanisms starting p
initialvalues = {}
initialvalues[q0] = 135*pi/180 #optimal angle is 135
initialvalues[q0_d] = 0*pi/180
initialvalues[q1] = -90*pi/180 #optimal angle is -90
initialvalues[q1_d] = 0*pi/180
initialvalues[q2] = -45*pi/180 #optimal angle is -45
initialvalues[q2_d] = 0*pi/180
initialvalues[q3] = 45*pi/180 #optimal angle is 45
initialvalues[q3_d] = 0*pi/180
initialvalues[q4] = 90*pi/180 #optimal angle is 90
initialvalues[q4_d] = 0*pi/180
initialvalues[q5] = 45*pi/180 #optimal angle is 45
initialvalues[q5_d] = 0*pi/180

```

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

```
#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 control)
system.set_newtonian(N)
```

```
#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
F.rotate_fixed_axis_directed(N,[0,0,1],q5,system) #the F frame rotates
E.rotate_fixed_axis_directed(F,[0,0,1],q4,system) #the E frame rotates
D.rotate_fixed_axis_directed(E,[0,0,1],q3,system) #the D frame rotates
```

```
#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 direction of ref frame
pAB = pNA + l0*A.x #pAB position is pNA's position plus l0 units in the direction of A
pBC = pAB + l1*B.x #pBC position is pAB's position plus l1 units in the direction of B
pNF = 15*N.x + 0*N.y #pNF position is 15 units in the direction of ref frame
pFE = pNF + l4*F.x #pFE position is pNF's position plus l4 units in the direction of F
pED = pFE + l3*E.x #pED position is pFE's position plus l3 units in the direction of E
```

```
#This block of code defines the centers of mass of each link (halfway along each link)
```

```
pAcm=pNA+l0/2*A.x #pA (link A) position is pNA's position plus one half of l0
pBcm=pAB+l1/2*B.x #pB (link B) position is pAB's position plus one half of l1
pCcm=pBC+l2/2*C.x #pC (link C) position is pBC's position plus one half of l2
pDcm=pFE+l3/2*E.x #pD (link D) position is pFE's position plus one half of l3
pEcm=pNF+l4/2*B.x #pE (link E) position is pNF's position plus one half of l4
pFcm=pNA+l5/2*N.x #pF (link F) position is pNA's position plus one half of l5
```

```
#This block of code computes and returns the angular velocity between frames
wNA = N.getw_(A)
wAB = A.getw_(B)
wBC = B.getw_(C)
wNF = N.getw_(F)
wFE = F.getw_(E)
wED = E.getw_(D)
```

2. Define Inertias: Add a center of mass and a particle or rigid body to each rotational frame. You may use particles for now if you are not sure of the inertial properties of your bodies, but you should plan on finding these values soon for any “payloads” or parts of your system that carry extra loads (other than the weight of paper).

```
#This block of code compute the inertia dynamics of each body and define
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)
BodyC = Particle(pCcm,mC,'ParticleC',system) #here, we represent the mass as a particle
BodyD = Body('BodyD',D,pDcm,mD,ID,system)
BodyE = Body('BodyE',E,pEcm,mE,IE,system)
BodyF = Body('BodyF',F,pFcm,mF,IF,system)
```

3. Add Forces: Add the acceleration due to gravity. Add rotational springs in the joints (using $k=0$ is ok for now) and a damper to at least one rotational joint. You do not need to add external motor/spring forces but you should start planning to collect that data.

```
#This block of code adds forces and torques to the system with the general function
#The first parameter supplied is a vector describing the force applied at the center of mass
#The second parameter is the vector describing the linear speed (for an external force)
system.addforce(-b*wNA,wNA)
```

```

system.addforce(-b*wAB,wAB)
system.addforce(-b*wBC,wBC)
system.addforce(-b*wNF,wNF)
system.addforce(-b*wFE,wFE)
system.addforce(-b*wED,wED)

```

```

<physics.force.Force at 0x7f6b01739910>

```

```

#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 defl
#The final parameter is the linear or angular velocity vector (dependin
system.add_spring_force1(k,(q0-preload0)*N.z,wNA)
system.add_spring_force1(k,(q1-preload1)*A.z,wAB)
system.add_spring_force1(k,(q2-preload2)*B.z,wBC)
system.add_spring_force1(k,(q3-preload3)*N.z,wNF)
system.add_spring_force1(k,(q4-preload4)*F.z,wFE)
system.add_spring_force1(k,(q5-preload5)*N.z,wNF)

```

```

(<physics.force.Force at 0x7f6b015223d0>,
 <physics.spring.Spring at 0x7f6b01522610>)

```

```

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

```

4. Constraints: Keep mechanism constraints in, but follow the pendulum example of double-differentiating all constraint equations. If you defined your mechanism as unattached to the Newtonian frame, add enough constraints so that it is fully attached to ground (for now). you will be eventually removing these constraints.

```

#This block of code defines the closed loop kinematics (vectors) of the
eq_vector = pBC - pED #vector from pBC to pED
eq_vector1 = pAB - pFE #vector from pAB to pFE
eq_vector2 = pBC - pNA #vector from pBC to pNA

```

```

#This block of code defines the systems constraints based on the vectors
eq = [] # eq -> equation

```

```

eq.append((eq_vector).dot(N.y)) #constrains pBC and pED to the same Ne
eq.append((eq_vector1).dot(N.y)) #constrains pAB and pFE to the same Ne

```

```

eq.append((eq_vector2).dot(N.x)) #constrains pBC and pNA to the same Ne
eq_d=[(system.derivative(item)) for item in eq]
eq_dd=[(system.derivative(item)) for item in eq_d]

```

5. Solution: Add the code from the bottom of the pendulum example for solving for $f=ma$, integrating, plotting, and animating. Run the code to see your results. It should look similar to the pendulum example with constraints added, as in like a rag-doll or floppy.

```

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

```

```

2021-03-01 02:37:46,021 - dynamics.system - INFO - getting dynamic

```

```

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

```

```

2021-03-01 02:37:46,632 - dynamics.system - INFO - solving a = f/m
2021-03-01 02:37:46,643 - dynamics.system - INFO - substituting constraints
2021-03-01 02:37:47,329 - dynamics.system - INFO - done solving a = f/m
2021-03-01 02:37:47,330 - dynamics.system - INFO - calculating func1

```

```

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

```

```

2021-03-01 02:37:47,635 - pynamics.integration - INFO - beginning integration
2021-03-01 02:37:47,640 - pynamics.system - INFO - integration at t=0
2021-03-01 02:37:48,241 - pynamics.system - INFO - integration at t=0.01
2021-03-01 02:37:48,371 - pynamics.integration - INFO - finished integration

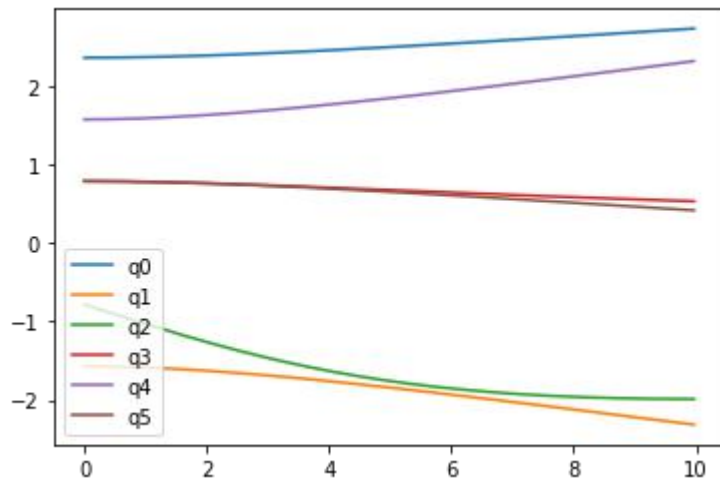
```

```

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

```


<matplotlib.legend.Legend at 0x7f6af34ef750>



#This block of code calculates and plots the energy of system

```
KE = system.get_KE()
```

```
PE = system.getPEGravity(pNA) - system.getPESprings()
```

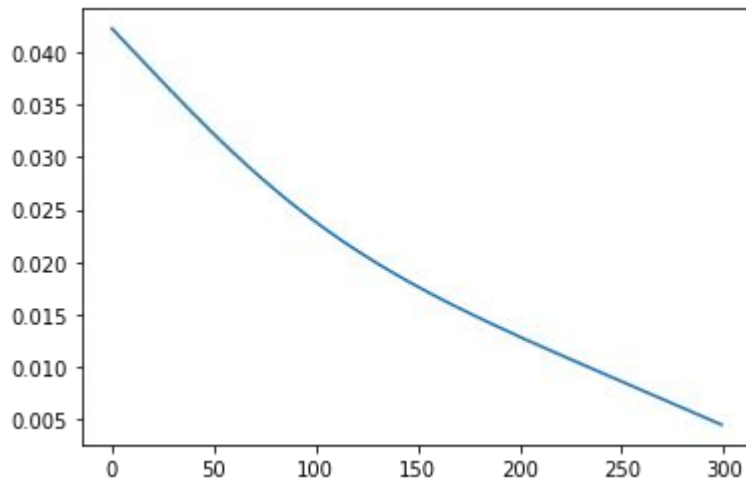
```
energy_output = Output([KE-PE],system)
```

```
energy_output.calc(states)
```

```
energy_output.plot_time()
```

2021-03-01 02:37:48,722 - dynamics.output - INFO - calculating out

2021-03-01 02:37:48,741 - dynamics.output - INFO - done calculatin



#This block of code calculates and plots the motion of the system

```
points = [pNA, pAB, pBC, pED, pFE, pNF]
```

```
points_output = PointsOutput(points,system)
```

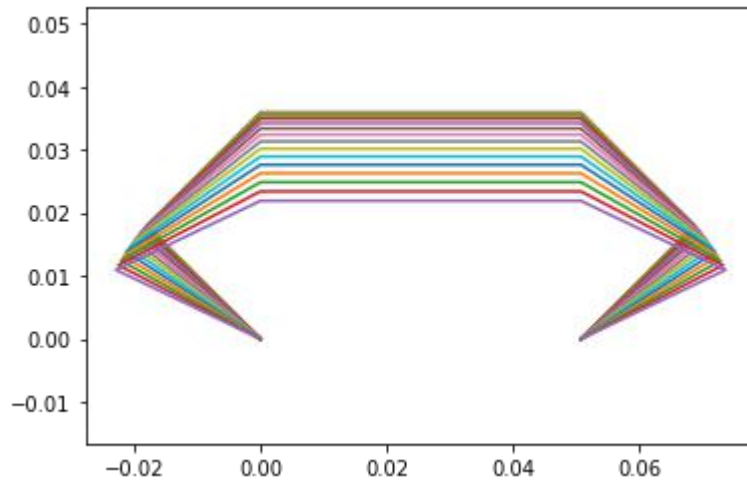
```
y = points_output.calc(states)
```

```
points_output.plot_time(20)
```

```

2021-03-01 02:37:48,967 - pynamics.output - INFO - calculating outj
2021-03-01 02:37:48,983 - pynamics.output - INFO - done calculating
<matplotlib.axes._subplots.AxesSubplot at 0x7f6af2f51f50>

```



```

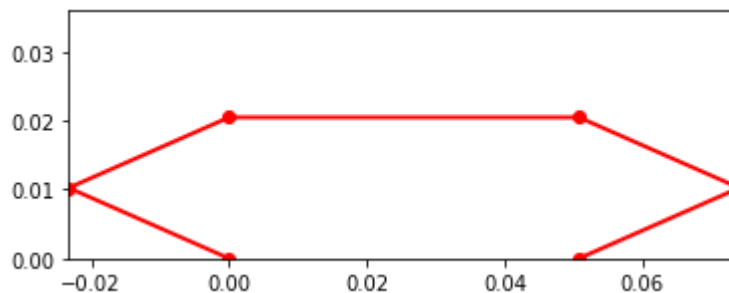
#This block of code produces a figure of the animations end position using
points_output.animate(fps = fps,movie_name = 'render.mp4',lw=2,marker='c')

```

```

<matplotlib.axes._subplots.AxesSubplot at 0x7f6af2f0a750>

```



6. **Tuning:** Now adjust the damper value to something nonzero, that over 10s shows that the system is settling.

```

#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())

```

Note: the system behaves like a rag-doll when we set the damper constant to 0. In the video shown above, the system does not behave in such manner because we have set the damper constant to a value greater than 0 in order to satisfy part 6.



Team 4 Dynamics II Plan

Feba Raju Abraham, Charlotte Deming, Andrei Marinescu

Material Selection

Our team has decided to use cardstock and cardboard for the main materials of fabrication. Each team member will either use cardboard or cardstock to test the dynamics of our system. In addition to these materials, one of the team members has access to a 3D printer. We will use PLA and a 3D printer to create professional parts for our prototype. Additionally, our team is using pen springs, three micro servo motors, a microcontroller, and glue.

Parameters Modeled

Because our team has three members, we decided to split one parameter simulation per member. Andrei will work on actuator fitting, Feba will gather damping information, and Charlotte will collect material link and joint stiffness information. The tests executed for each of these parameter models will be explained below:

- **Actuator Fitting:** because the team is using a spring and motor to contract and expand the sarrus mechanisms, the team must first calculate the force/torque needed to contract the spring. This will be done with theoretical calculations through MATLAB code and physical testing with the use of a scale and ruler to determine force vs displacement. Once the torque required to contract the spring is calculated, a motor will be specced based on its max torque capabilities.
- **Joint Stiffness Calculations:** the team will execute a variety of different cantilever tests using both cardboard and cardstock to determine the superior material of the two. Each material will be clamped to a desk and weights will slowly be attached to the end of the material. The team will gather deflection information and calculate the overall stiffness of the materials using equations and data from the cantilever tests.
- **Damping Calculations:** to determine the damping of the sarrus mechanisms, video software was used to track the motion of the mechanism when a weight is dropped onto it. The video software will output an oscillating signal which will then be used to determine the damping coefficient of the sarrus mechanism.

Prototyping

Because Andrei has a 3D printer, cardstock, cardboard, springs, glue, motors, and microcontrollers, he will be in charge of creating our team's prototype. The team's mechanical design will consist of three sarrus mechanisms placed on top of one another vertically. The sarrus mechanisms are made from cardstock material and the spring/support system is made from PLA printed parts.

Individual Deliverables

- **Andrei:** actuator fitting, updating code, prototyping, compiling information to a report (Jupyter notebook and google doc will be used for this).
- **Feba:** damping calculations, adding model fitting routines
- **Charlotte:** material stiffness calculations and testing, filtering input data

The **Razor Clam** is a common name for a species of cold-blooded muscles or clams known for their ability to descend and ascend through sand with minimal energy. This is achieved with a strong muscle that is used to pull the body of the clam up or down using surrounding sand as an anchor point.

Parameter Identification

Feba Raju Abraham
Arizona State University
fabraha3@asu.edu

Charlotte Deming
Arizona State University
cdeming1@asu.edu

Andrei Marinescu
Arizona State University
amarine3@asu.edu

Abstract

This report examines data from the various aspects of team fours current bio-inspired mechanism. These aspects include: the full system, actuator parameters, energy storage, system and link stiffness, as well as damping and friction coefficients. The data collected throughout this report will be used to further define and improve the robotic clam design that the team is currently developing.

1. TRACKING THE MOTION OF A JOINT

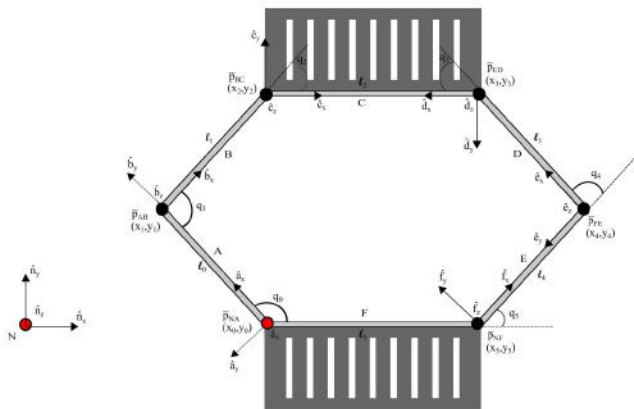


Figure 1. Dynamics model of sarrus link

The aim of the experiment was to track motion of the joint at pAB (shown in figure above) using the software called 'Tracker' [4] and to collect data from the motion. The data is then used in a python program to generate plots.

EXPERIMENT SETUP

Aim:

To create a model of the sarrus link and capture its motion as video.

Materials used:

1. Cardboard
2. Scissors
3. Tape
4. Needle and Thread

5. Coloured paper (orange)
6. Plain paper
7. Stapler
8. Smartphone with video capability
9. Glue

Preparation:

1. Cut the cardboard into a 39cm long strip, with a 3cm width.
2. Cut the coloured paper into three small squares.
3. Fold and tie two long pieces of thread into two thicker strings

Procedure:

1. Mark the long piece of cardboard at regular intervals of 6.5cm each, until six intervals are obtained.
2. These are the six links of the Sarrus link to be modelled
3. Bend the strip at the marked points.
4. Join the ends of the strip with tape to create a sarrus link.
5. Make small holes in the top and bottom links that are relatively vertical to each other.
6. Insert the thread through the holes to connect the top and bottom holes to each other.
7. Place needles on the top link to prevent the threads from slipping.
8. Glue the three orange squares on points pNA, pNF and pAB (shown in figure above). The points pNA and pNF create the base frame.
9. Attach the link to plain paper by stapling them together, make sure the paper does not cover the holes at the bottom.
10. Fix the apparatus on the table with tape.
11. Pull both strings and release them to create an up-down motion of the top link.
12. Capture this motion by taking a video.

Assumptions

- The base frame is fixed and rigid.
- The links are rigid.
- The mass of the needles is neglected.
- The joints are all the same, including the joint that was taped.
- Friction between threads and cardboard is neglected.

Results

The captured video is run through the software, 'Tracker'. The point mass is placed at the joint pAB. The distance between the points in the base frame (pNA, pNF) is measured to be 7cm. The distance measured between the lens of the camera and the experiment setup is 30cm.

Since the joint was moved manually, there is no damping observed in the data plotted from the point mass (Figure 1.b). The plot shown in Fig 1.b is based on the x and y values obtained by the tracker. A part of the huge table of values is shown in Fig 1.c. The data from the table from the tracker software is used to plot data such as time vs x and time vs y (shown in figures 1.d, 1.e)

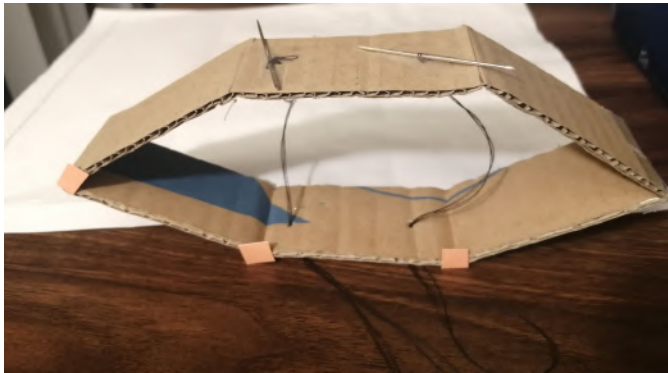


Figure 1.a. Experimental Setup

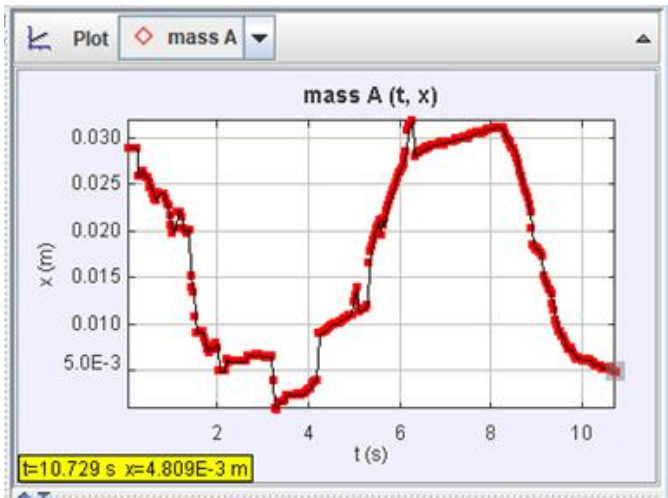


Figure 1.b. Plot of the motion of the joint

t (s)	x (m)	y (m)
0.033	2.898E-2	6.303E-2
0.066	2.896E-2	6.300E-2
0.098	2.895E-2	6.297E-2
0.131	2.898E-2	6.285E-2
0.164	2.899E-2	6.279E-2
0.197	2.896E-2	6.278E-2
0.230	2.893E-2	6.270E-2
0.262	2.589E-2	6.412E-2
0.295	2.589E-2	6.396E-2
0.328	2.651E-2	6.408E-2
0.361	2.651E-2	6.407E-2
0.394	2.586E-2	6.451E-2
0.427	2.582E-2	6.445E-2
0.459	2.582E-2	6.449E-2
0.492	2.582E-2	6.445E-2

Figure 1.c. Table showing a portion of the time and xy position of the joint

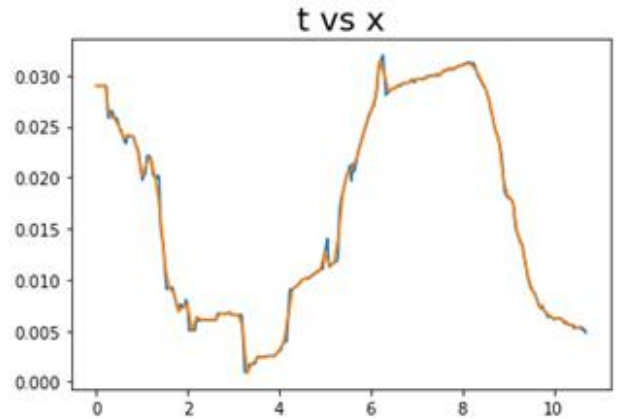


Figure 1.d. Plot of Time vs x axis

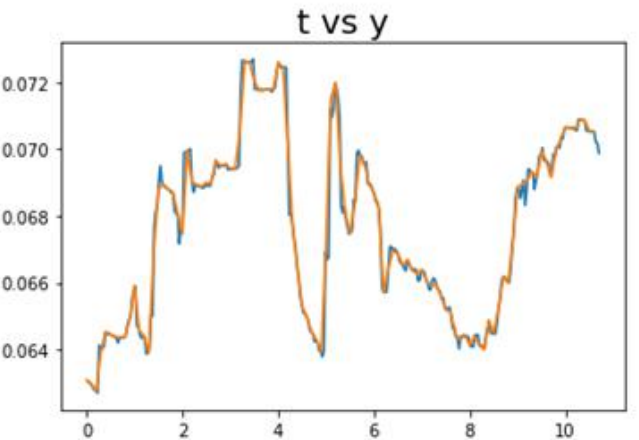


Figure 1.e. Plot of Time vs y axis

```
In [1]: import pandas as pd
import numpy
import matplotlib.pyplot as plt
import scipy.interpolate as si

df=pd.read_csv(r"C:\Users\FebaraJu\Documents\data1.csv', sep=',')

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

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

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

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

plt.figure()
plt.title('x vs y',fontsize = 20)
plt.plot(x,y)
```

Figure 1.f. Snippet of the code used for plotting time vs x and time vs y

2. Actuator Fitting

The team is using a compression spring in combination with a servo motor as the main actuation method of the system. Custom three-dimensional printed PLA parts will be used to house the spring and motor relative to the cardstock sarrus mechanism. The design is simple, easy to assemble, and customizable to multiple design and spring types. The sarrus link dimensions that the team decided to use is shown below in Figure 2. The figure depicts the top and bottom links of the sarrus mechanism to be two inches by two inches. Using these dimensions, three-dimensional parts were printed and shown in Figure 3. These parts will keep the top and bottom links stiff in order to allow for contraction and expansion using a spring. The spring is inserted in between these parts and a lock pin holds the parts together. A small hole in the two parts allows for the addition of a string. This string will be used to contract the spring with a servo motor. The assembled system is shown in a CAD rendering to emphasize hidden details (see Figure 4).

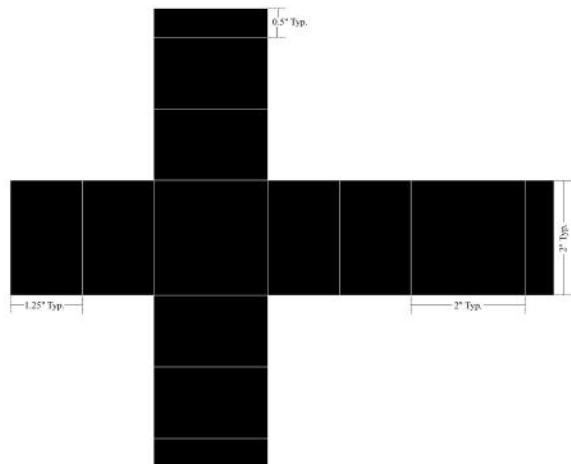


Figure 2. Two Dimensional Cardstock Sarrus Outline



Figure 3. Three Dimensional PLA Printed Parts

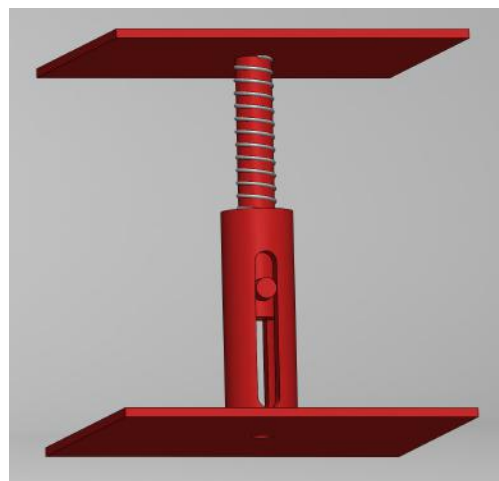


Figure 4. CAD Rendering of Assembled Printed Parts

In order to keep the sarrus link close to its fully extended position at rest, the three dimensional printed parts have a height of 2.250 inches. As seen in Figure 2, with two side links of 1.250 inches, the max height the sarrus link can extend to is 2.500 inches. The team does not want the system to fully extend. As a result, the assembled printed parts have a height 0.250 inches shorter than the sarrus mechanisms max height. The team temporarily selected two compression springs with a combined starting height of 1.000 inches. They were selected because of the low spring constant (based on physical resistance). This will allow for easy compression which a servo can achieve without overcurrent limitations. When fully compressed, the two springs have a combined height of 0.2800 inches. Each spring has six active coils and eight total coils, where the other two are used to close the ends of the spring. We assume the springs are made of zinc-coated steel (ASTM A591 Zinc Coated Steel, Commercial) [2]. The outer diameter of the springs are 0.28125 inches. Additionally, the wire diameter is 0.02 inches.

Unfortunately, the springs used by the team did not have a manufacturer spring constant. As a result, the team had to calculate this value using physical testing. Using the spring information from above, and the formulas described below, the team was able to calculate the combined springs rate of the two springs used (k).

Using MATLAB, the team declared all of the known variables. Then the team used three additional formulas to calculate the spring rate (k) [3]. Figure 5 shows the code used and the resulting k value of 194.56 Newtons per meter. To further confirm these calculations, the team physically compared the displacement versus force of the system using Hooke's Law. As shown in Figure 6, the team used a scale and ruler to gather data on the springs displacement relative to the force applied. Figure 7 shows a plot of the physical test results including the spring constant (k) and the elastic potential energy. Taking the average of the physical testing and theoretical calculations, we conclude that the spring constant is approximately 200.45 Newtons per meter.


```

Editor - C:\Users\drema\Desktop\Parameter Identification Images\Spring_Constant_Calculator.m
Spring_Constant_Calculator.m
1 %Spring Constant Calculator:
2
3 %Constants:
4 d = 0.02/25.4; %wire diameter (mm)
5 Do = 0.28125/25.4; %outer diameter (mm)
6 E = 220.0; %Young's Modulus of material (GPa)
7 v = 0.29; %Poison's Ratio of material (unitless)
8 Lf = 1.0/25.4; %free length (mm)
9 na = 12; %number of active coils
10
11 %Formulas:
12 D = Do - d; %mean diameter (mm)
13 G = E/(2*(1+v)); %Shear Modulus of material (GPa)
14 k = 1000*((G*d)^4)/(8*(D^3)*na); %spring rate (spring constant k) (N/m)
15
16 %Results
17 fprintf('Calculated Shear Modulus is %.2f GPa.\n',G);
18 fprintf('Calculated Spring Rate is %.2f N/m.\n',k);

```

Command Window

```

>> Spring_Constant_Calculator
Calculated Shear Modulus is 85.27 GPa.
Calculated Spring Rate is 194.56 N/m.

```

Figure 5. MATLAB Spring Constant Code

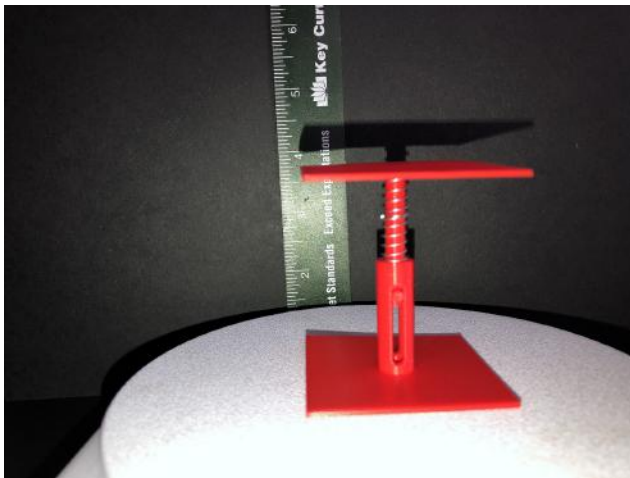


Figure 6. Physical Spring Constant Test Setup

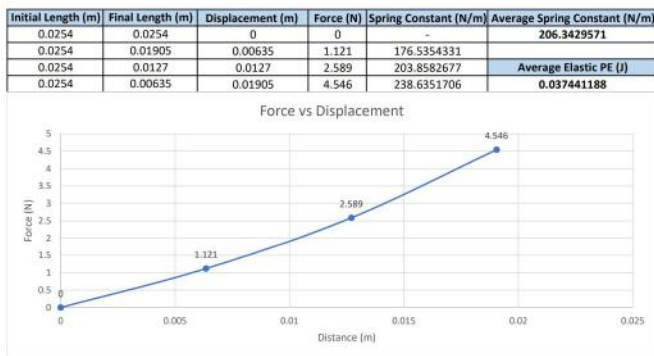


Figure 7. Resulting Plot of Physical Spring Constant Test

From the tests executed above, the team was able to spec a motor capable of providing the necessary force needed to fully compress the spring. With a spring constant of approximately 200.45 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. A common micro servo has a stall torque of 2.1 kilogram-centimeters at 4.8 volts. This equates to about 0.21 Newton-meters which is almost nine times the required force we need to compress the systems springs. As a result, the team decided to spec Adafruit's MG90D Metal Gear Micro Servo shown in Figure 8. This 22.8 by 12.2 by 28.5 millimeter servo can accept an input voltage anywhere from 4.8v to 6 volts DC.



Figure 8. Adafruit's MG90D Metal Gear Micro Servo [1]

3. Cantilever Beams Process

One material the team is considering using is corrugated cardboard. This material is both thicker and sturdier than cardstock. To test the stiffness of this material, a cantilever beams process was performed.

Materials used:

1. 4 rectangular strips of cardstock of the same size (25 mm x 125 mm)
2. Clamp
3. String
4. Adhesive, such as tape
5. Various masses; The masses used by the team can be seen below in Figure 9

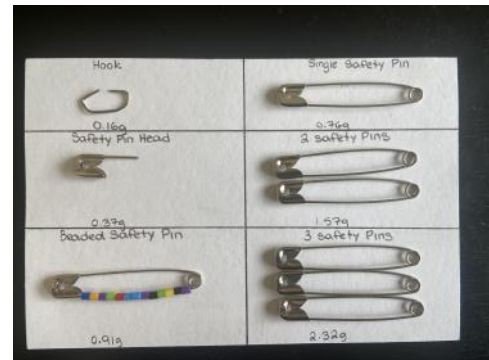


Figure 9. 6 masses used for loading the samples, with no sample being greater than 3g

Procedure [5]:

1. Prepare the 4 strips of material, one of which will be the base, and three of which will be the samples from which masses will be hung (Figure 10)

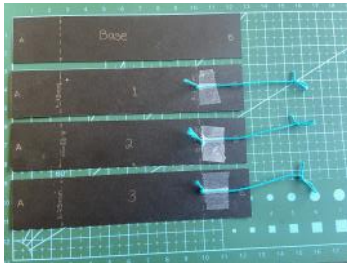


Figure 10 The base and 3 samples

2. Measure the thickness of each sample at 2 separate locations.
 - a. Each sample was measured 25mm from each end and resulted in the same value of 0.23mm with each measurement
3. Tape a string to the top of each of the samples as a loop from which to hang the masses. (Figure 10)
 - a. The strings were weighed and did not register on the gram scale.
4. Clamp the base material and one of the sample materials such that 100mm is hanging off the edge of the table or counter it was clamped to. As seen in Figure 11.

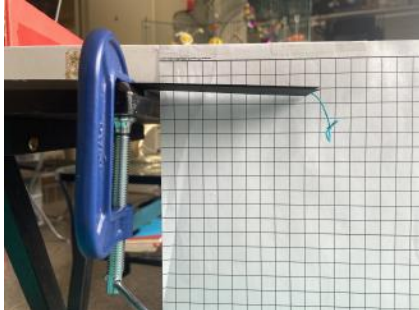


Figure 11. Sample 1 clamped with the base to the table

5. Mount a camera in front of the beam in a stable location to take multiple pictures across tests
6. Take a picture of the unloaded beam, obtaining a picture similar to that of Figure 11.
7. Load the sample with the masses (Figure 9) one at a time, measuring the deflection of the sample with respect to the base, taking a picture of each load. An example of this can be seen below in Figure 12.

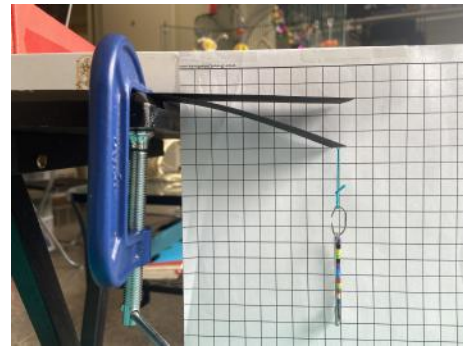


Figure 12. A mass loaded onto sample 1

8. Repeat the test for the other 2 samples

Results:

The deflection tests use the equations found in Figure 13. Looking at the maximum deflection equation, it can be seen that all values are easily measurable given the test setup above (Figure 11), leaving only Young's Modulus (E) to for calculation. From there, the team can reorder these equations so that it is solving for E, as seen in Figure 12, assuming I is equal to that in Figure 13.


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 used to calculate young's modulus [5]

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

Figure 12. Young's Modulus (E)

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

Figure 13. The given equation for I

To use these equations, the data from the tests had to be compiled, which can be found in Figure 14, which is the deflection of the sample from the base, removing the initial deflection that had occurred due to imperfections in the material. The masses labeled 1-6 in Figure 14 are labeled in Figure 15.

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 14. Data of the deflection of each mass on each sample.

Loads and weights		
Load	Item	Weight in kg (with hook)
1	Hook	0.00016
2	Safety Pin Head	0.00053
3	1 Safety Pin (empty)	0.00092
4	Beaded Safety Pin	0.00107
5	2 Safety Pins (empty)	0.00173
6	3 Safety Pins (empty)	0.00248

Figure 15. Masses in kg

A visual representation of the deflection over the mass can be found in Figure 16. It can be seen that there is a quadratic relationship between the two values.

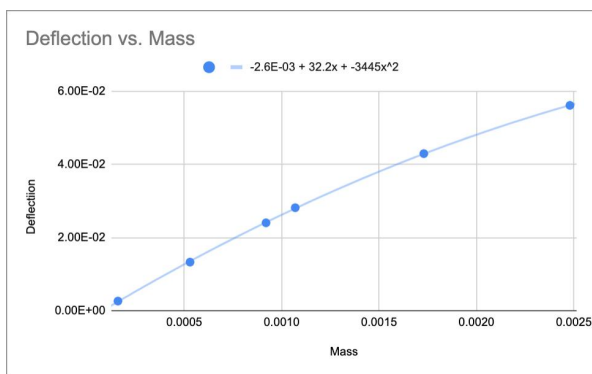


Figure 16. Graph of deflection over mass

With the data from Figures 14 and 15, using the equations to Find E, the team was able to calculate the average Young's Modulus for the material, which came out to be $6.98 \times 10^3 \text{ N/M}^2$.

4. Euler-Bernoulli Beams Code

The entirety of the Euler-Bernoulli Beams code can be found in the Dynamics II section of the code. The end of the code can be found below in Figure 17.

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

0.0542171836310331
```

Figure 17. Some of the Euler Bernoulli Beams Code Results

5. System Level Prototype

The team's mechanical design consists of three sarrus mechanisms placed on top of one another vertically as shown in Figure 18. 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 19. 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 20. 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:

1. All sarrus mechanisms will begin motion in an extended position, with the spring mechanism forcing the sarrus link open.
2. 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 21.
3. The middle sarrus mechanism will then contract its spring in order to pull the bottom-most link upwards.
4. The bottom-most sarrus mechanism will then contract its spring in order to lock itself against the surrounding walls.
5. The middle and top-most sarrus mechanisms will then expand their springs in order to unlock from the surrounding walls and move upwards.
6. 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 4. This housing will hold the servo motor and pulley.

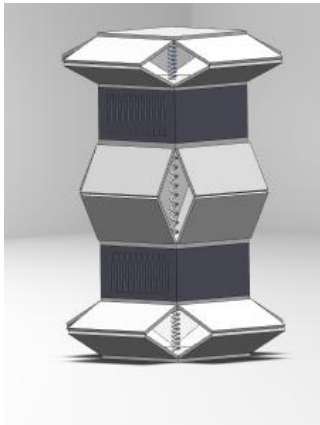


Figure 18. Three Sarrus Mechanism CAD Model

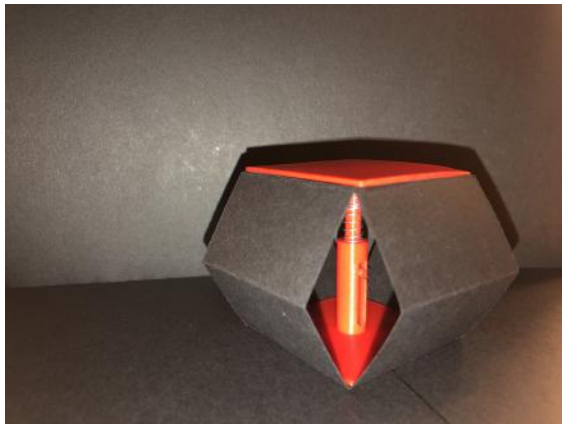


Figure 19. Sarrus Mechanism at Rest

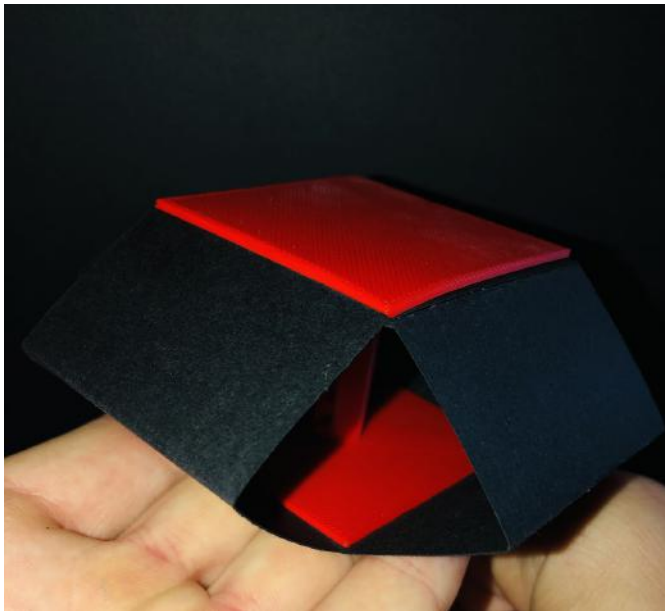


Figure 20. Sarrus Mechanism Contracted



Figure 21. Sarrus Mechanism Anchored to Walls

7. Contributions

All team members contributed to the research discussed throughout this report. Specifically, Feba Raja Abraham extracted data from a motion capture of our mechanism. Andrei Marinescu created a system-level prototype and developed a model for our actuator. Charlotte Deming modeled our systems link stiffness via Cantilever Beams tests. She also wrote and ran the Euler-Bernoulli Beams code on our mechanism.

References

- [1] A. Industries, "Micro Servo - MG90D High Torque Metal Gear," *adafruit industries blog RSS*. [Online]. Available: <https://www.adafruit.com/product/1143>. [Accessed: 08-Mar-2021].
- [2] "The Online Materials Information Resource," *MatWeb*, 1996. [Online]. Available: <http://www.matweb.com/search/datasheet.aspx?matguid=af58cf14010141b1a1cd94def4826389&ckck=1>. [Accessed: 08-Mar-2021].
- [3] "Spring Rate Calculator," *The Spring Store*. [Online]. Available: <https://www.thespringstore.com/spring-rate-calculator.html>. [Accessed: 08-Mar-2021].
- [4] "Basic Tracker Tutorial", Daniel Aukes, *Foldable Robotics* [Online]. Available: <https://egr557.github.io/modules/validation/Tracker%20tutorial.html> [Accessed: 07- Mar-2021]

[5] “Approximating compliant beams with the pseudo-rigid body model[Online].
Available:<https://egr557.github.io/modules/compliance/generated/prbm.html>[Accessed: 08- Mar-2021]

▼ System Dynamics II

```
!pip install pypoly2tri idealab_tools foldable_robotics dynamics
```

```
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: dynamics 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: cyceler>=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 contact 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.

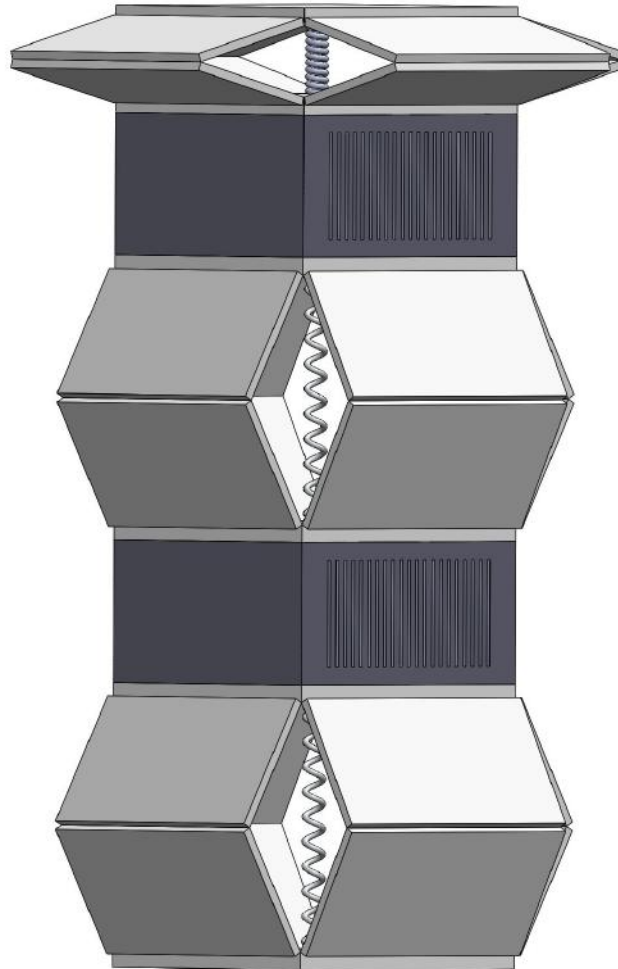


Figure 1a: Three Sarrus Mechanism CAD Model

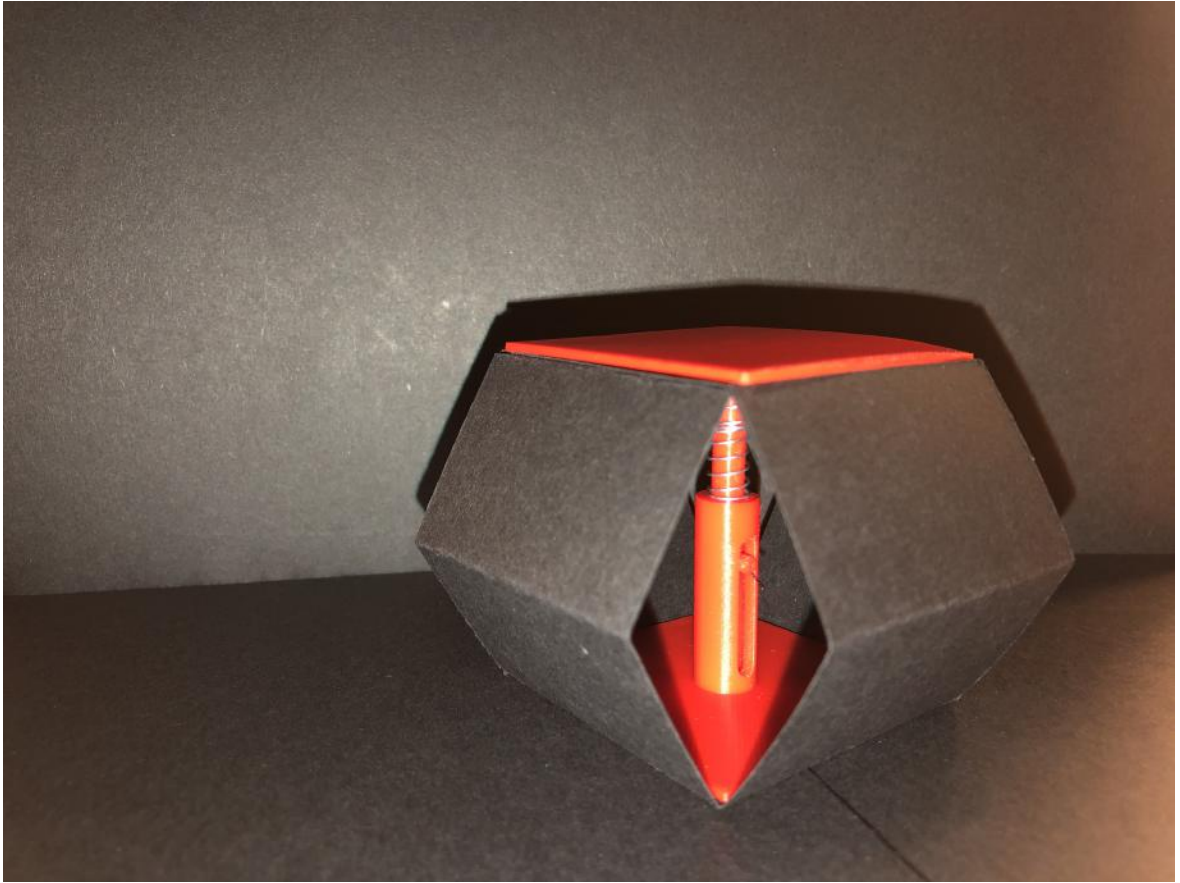


Figure 1b: Sarrus Mechanism at Rest

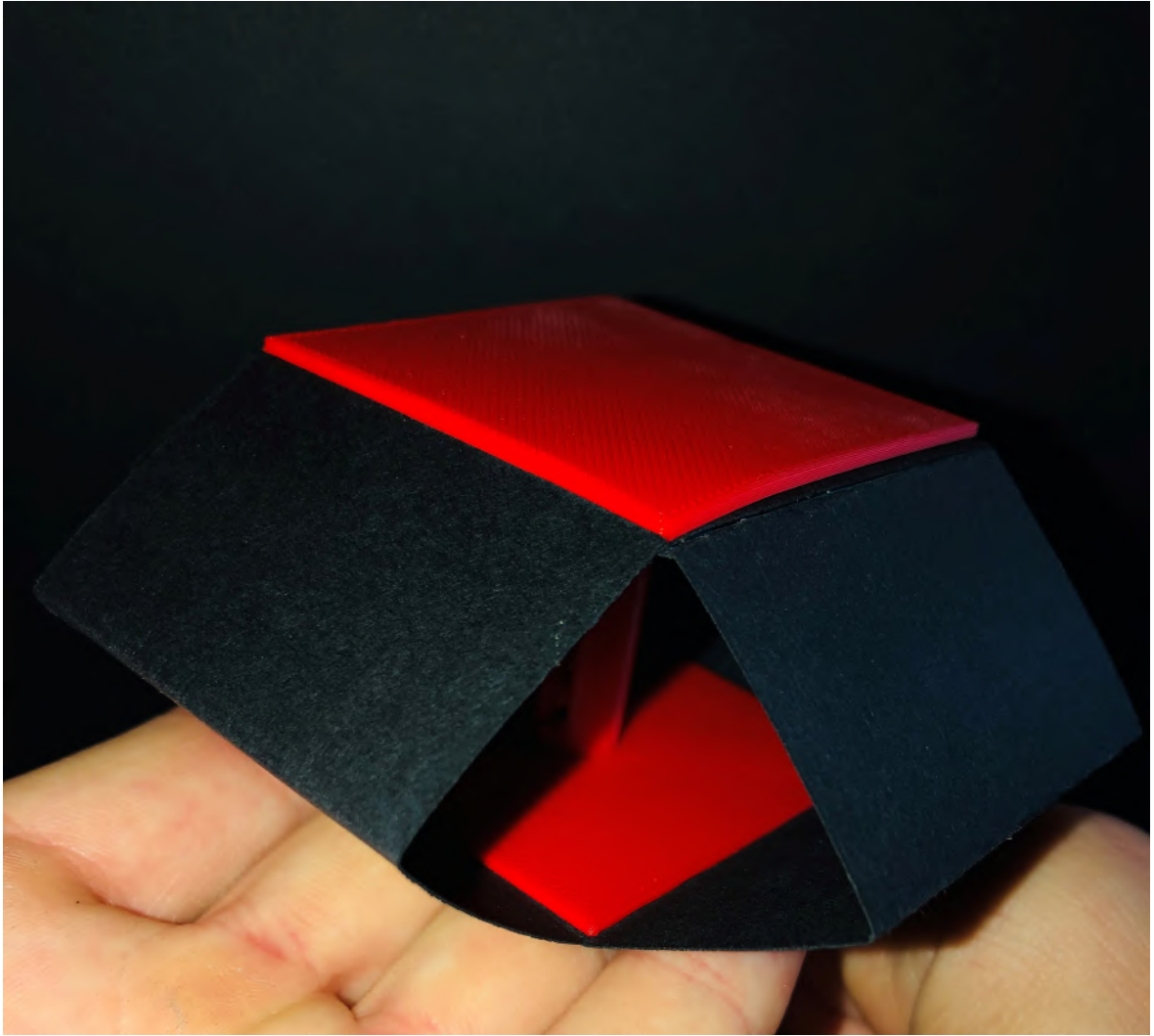


Figure 1c: Sarrus Mechanism Contracted

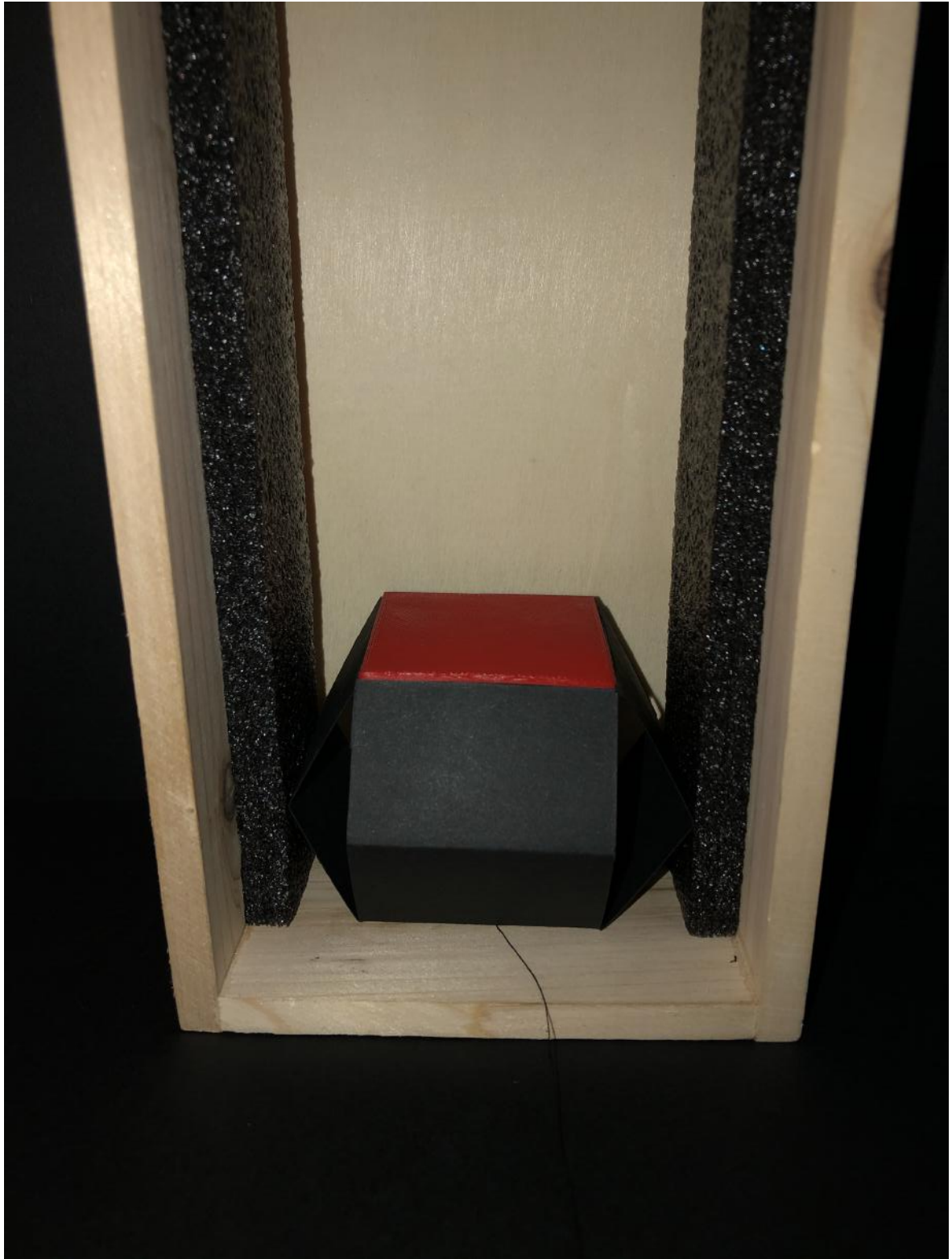


Figure 1d: Sarrus Mechanism Anchored to Walls

▼ 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 pulley 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.

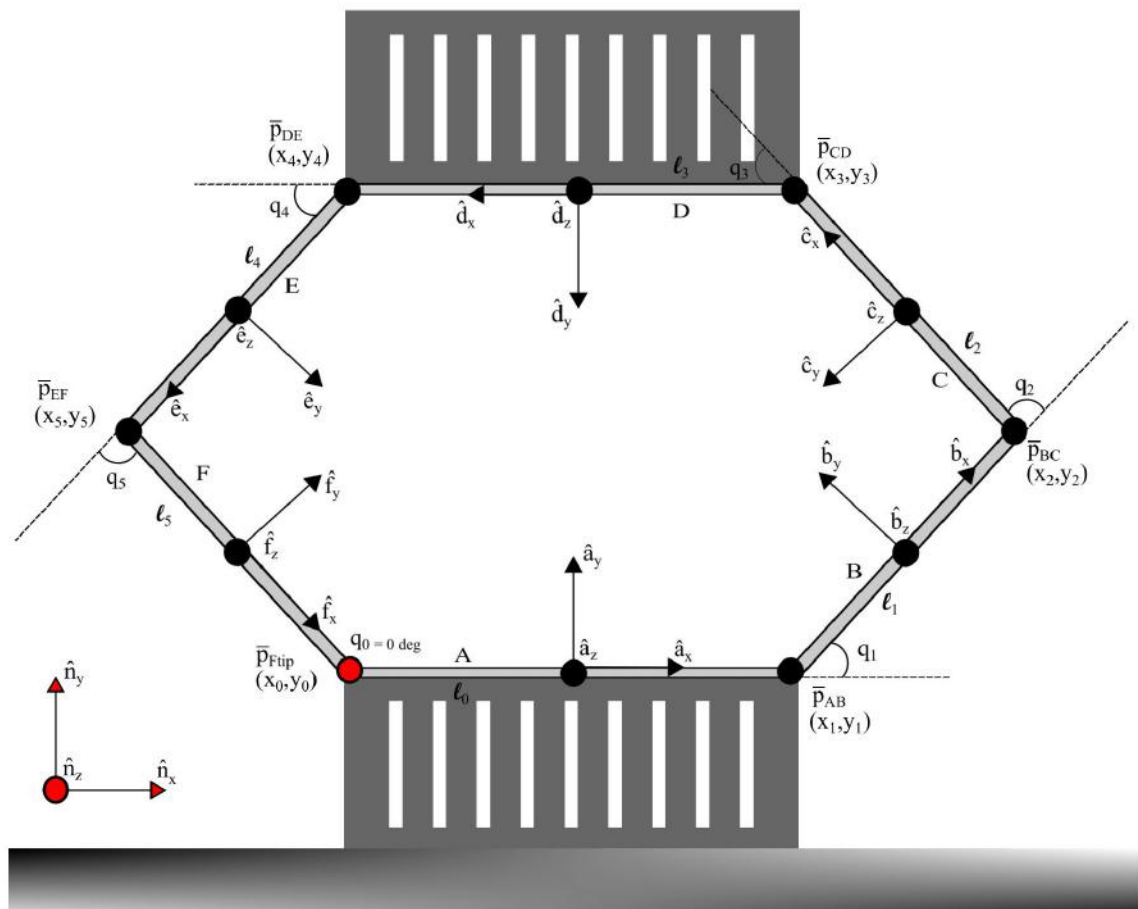


Figure 3a: 2D Dynamics Model of System

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 $2us - mg = 0$. *In the horizontal direction, the two normal forces must add up to zero. From the free body diagram we see that $2usF_{normal} = mg$, therefore, $F_{normal} = mg/(2us)$.* 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 F_{push} . Therefore, each of the two sarrus joints must push with a magnitude of **$F_{push} = (mg/2) * \sqrt{1 + (1/us^2)}$** .

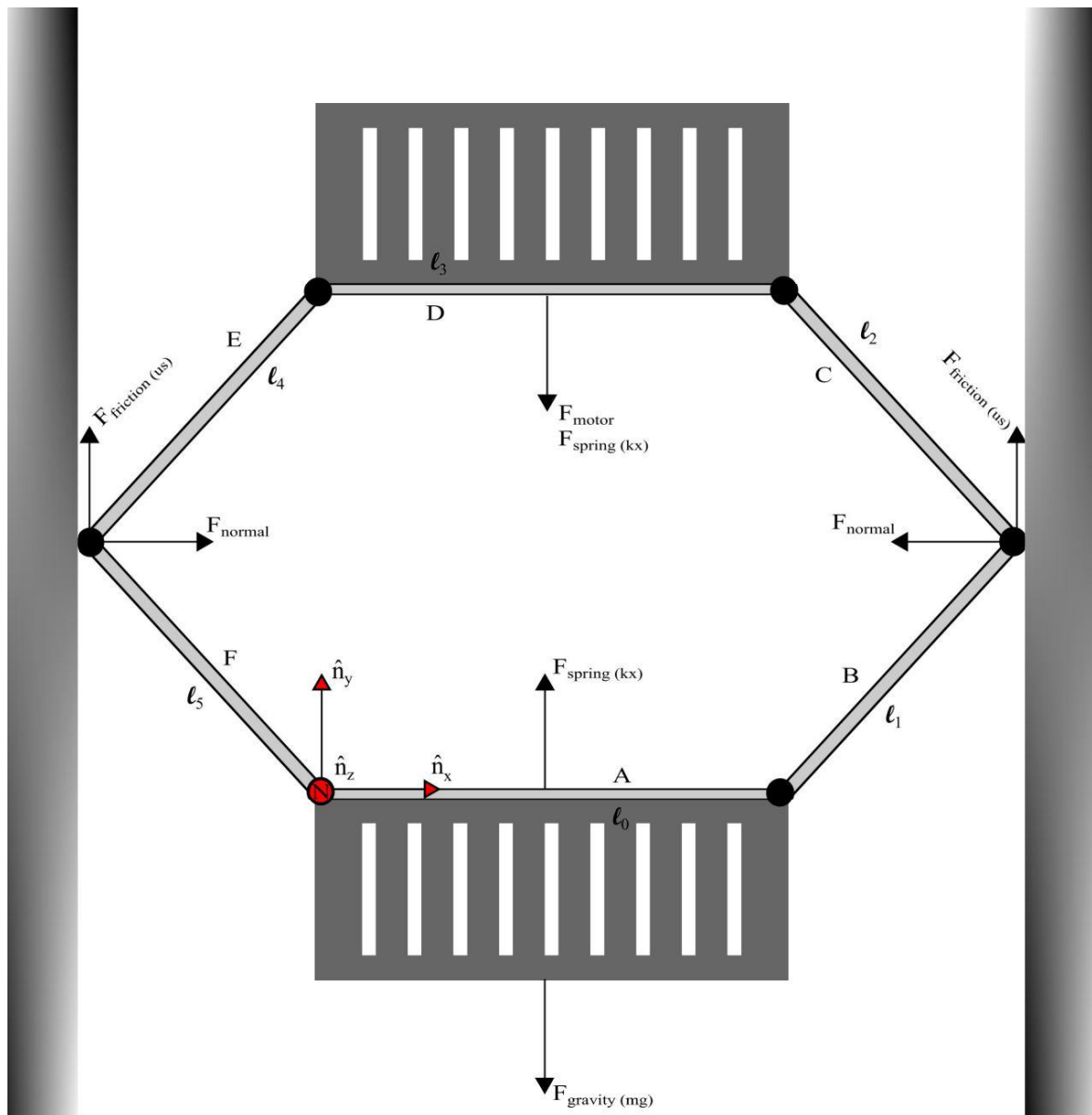


Figure 3b: 2D Free Body Diagram of System

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 our 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-identical 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.

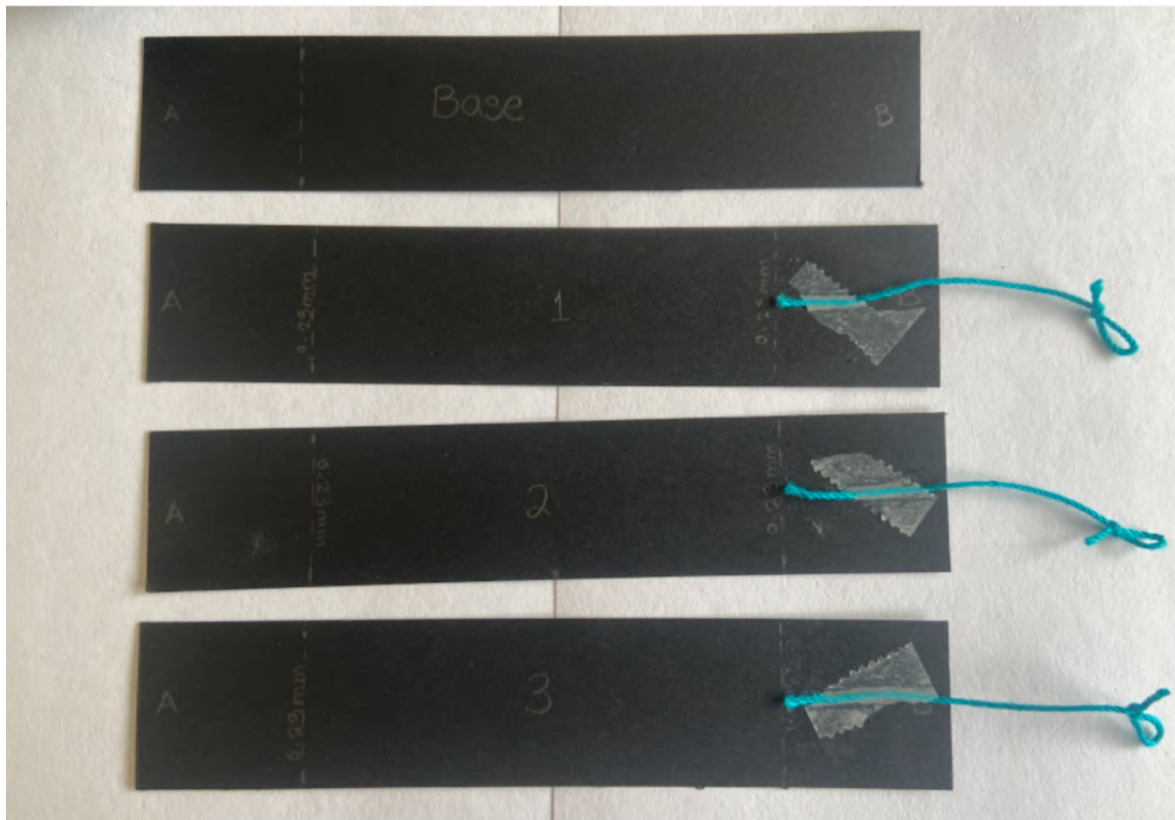


Figure 4: The three 125mm x 25mm samples of cardstock used for the cantilever tests

2. Measure the thickness of each sample in two places

The thicknesses of each of the samples was surprisingly 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 the highest being 2.32g, as can be seen in Figure 6.

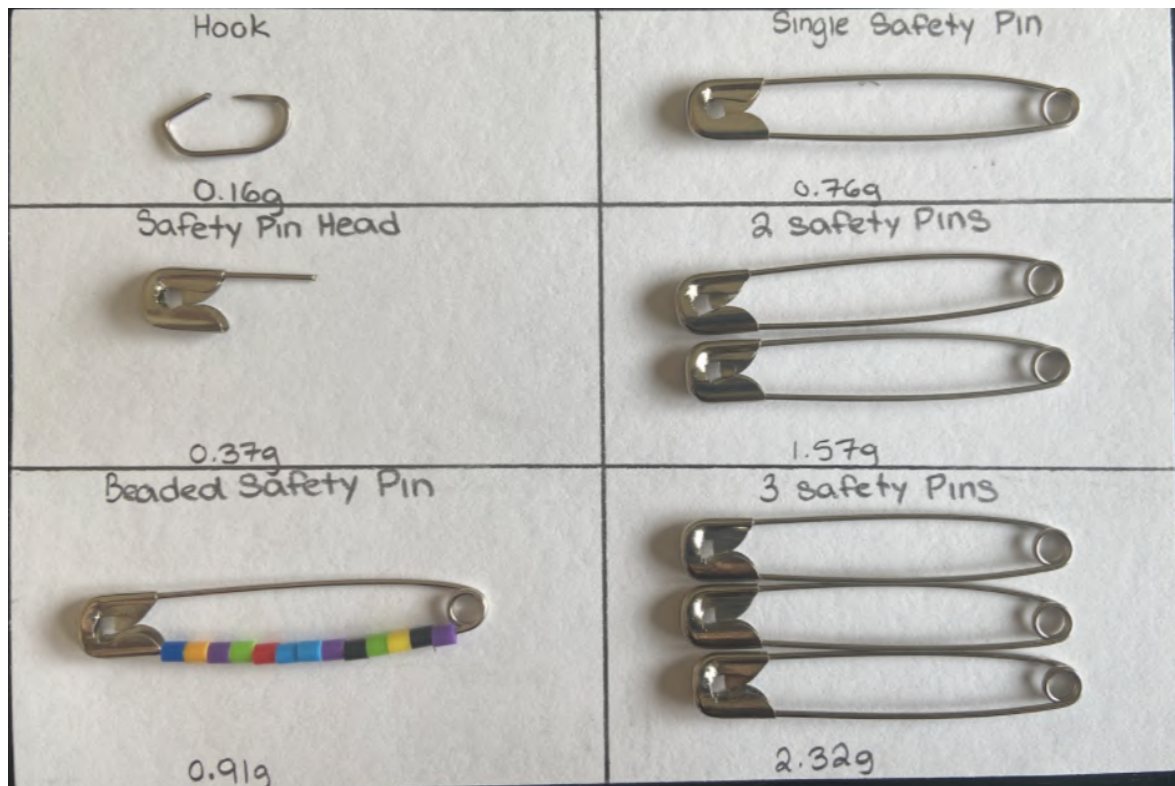


Figure 6: The six masses that were used in the cantilever tests

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 deflection based on imperfections in the sample and base.

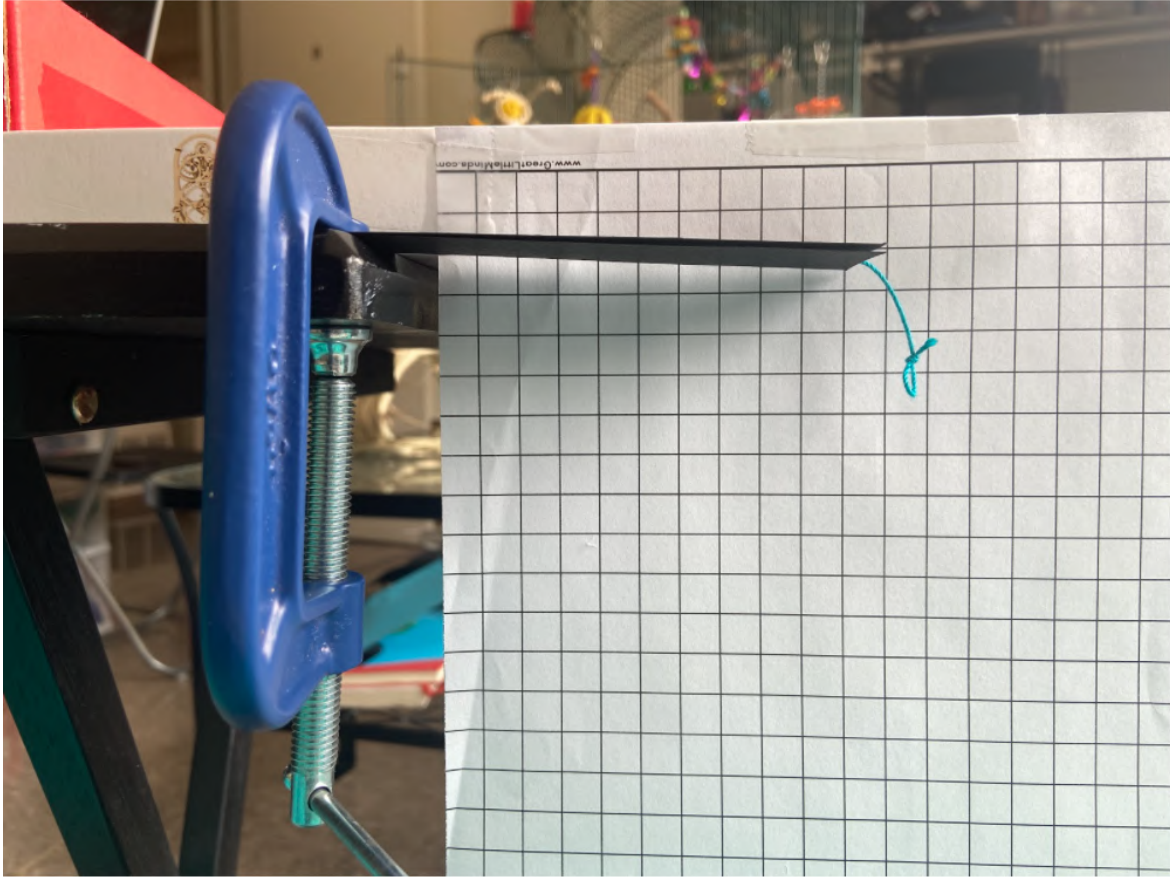


Figure 7: Setup for the cantilever test without a load

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.

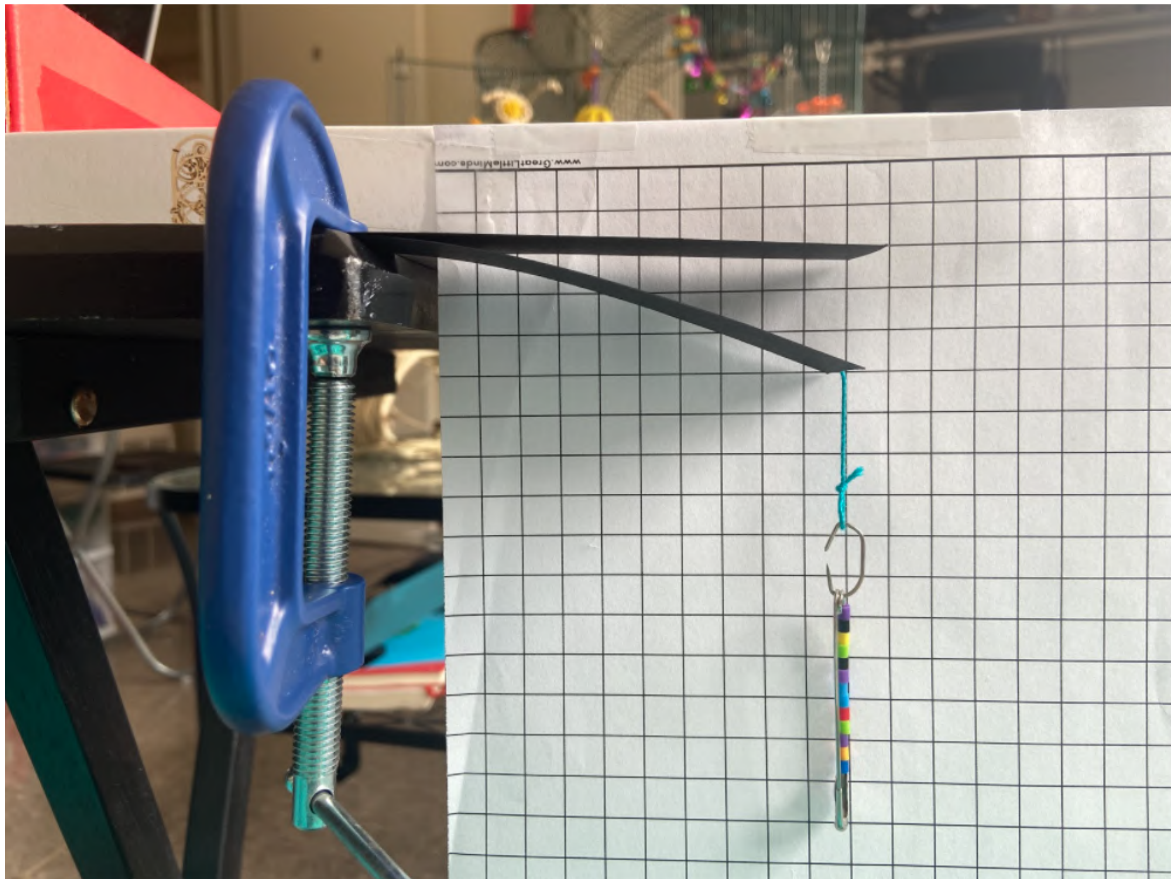


Figure 8: Image of one of the tests run with a load

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

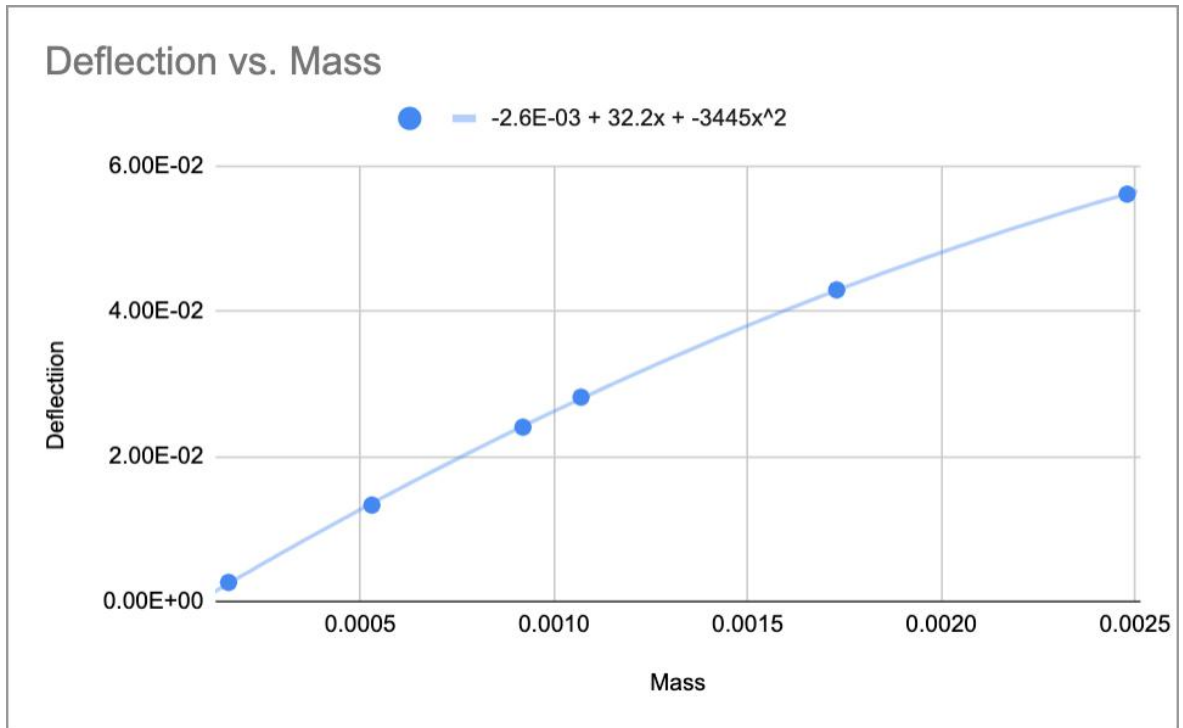


Figure 10: Graphical representation of the deflection versus mass graph

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 the 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 initially find the cross sectional moment of inertia. 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

```
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
```

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

```
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

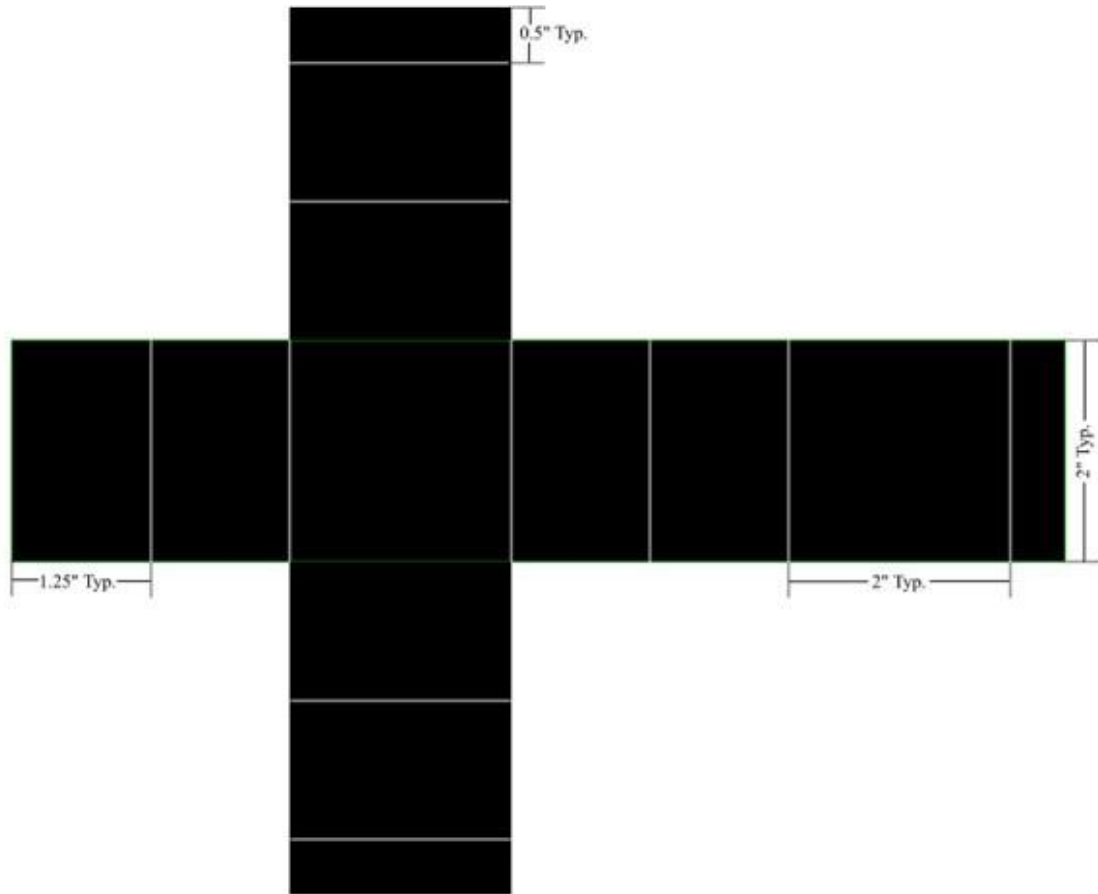


Figure 2: 2D schematic of 3D Sarrus system

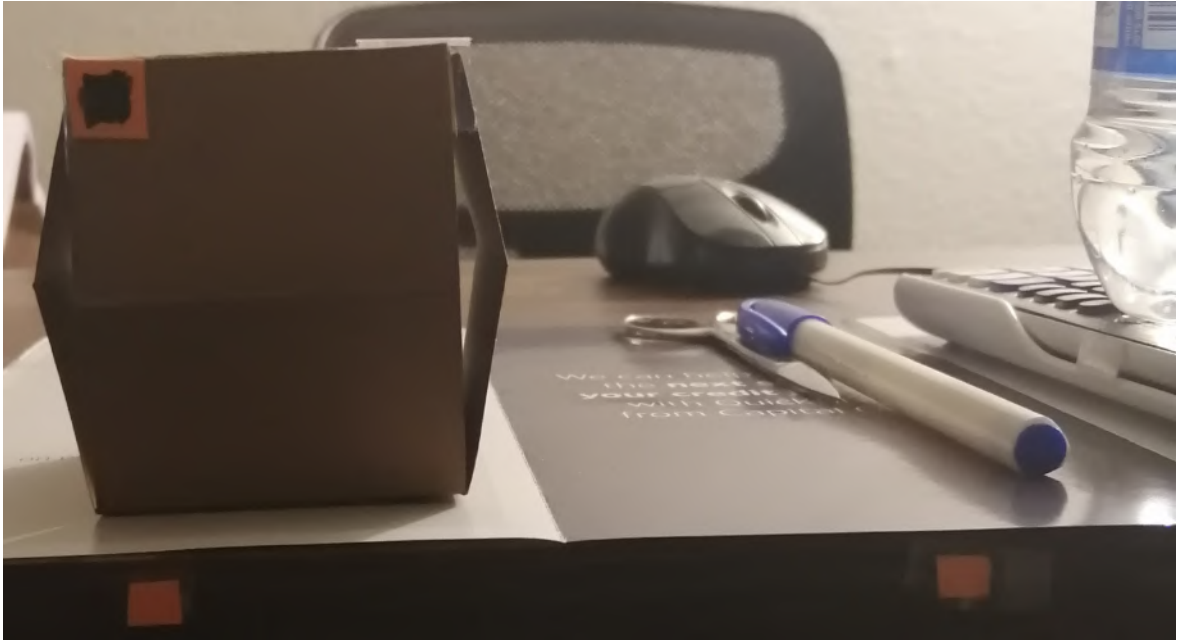


Figure 3: The setup of the experiment



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()
```

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']))
```

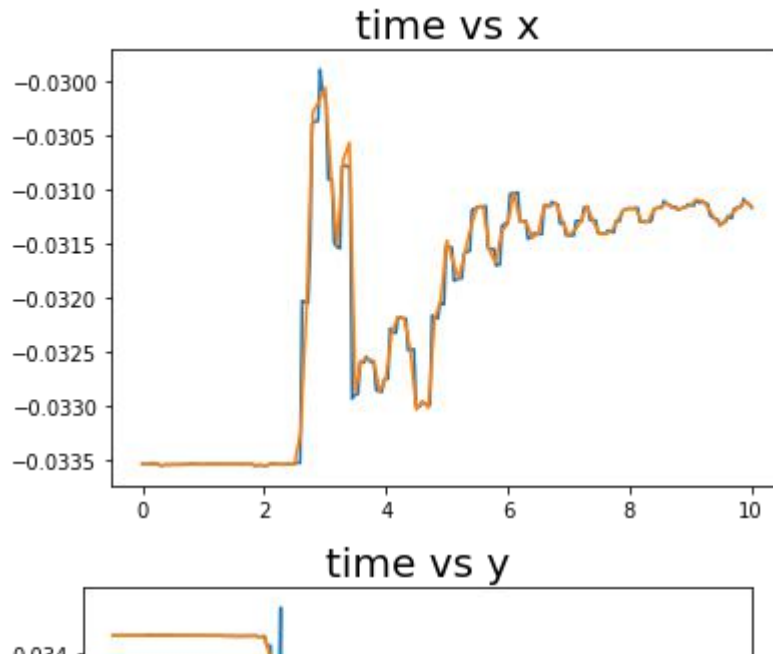
```
x = df2.x.to_numpy()
y = df2.y.to_numpy()
t = df2.t.to_numpy()
```

```
xy = numpy.array([x,y]).T
f = si.interpld(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]

▼ Dynamics Code

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

```
%matplotlib inline
```

```
#This block of code imports the necessary modules created by Daniel M.
import dynamics
from dynamics.frame import Frame
from dynamics.variable_types import Differentiable, Constant
from dynamics.system import System
from dynamics.body import Body
from dynamics.dyadic import Dyadic
from dynamics.output import Output, PointsOutput
from dynamics.particle import Particle
import dynamics.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) #lin ~ 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) #lg ~ 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_D = Constant(1, 'Izz_D', 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 dynamics 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 variables 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
nBcm=nAB+l1/2*B.x #nB (link B) position is nAB'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_forcel(k,(q0-preload1)*N.z,wNA)
system.add_spring_forcel(k,(q1-preload2)*A.z,wAB)
system.add_spring_forcel(k,(q2-preload3)*B.z,wBC)
system.add_spring_forcel(k,(q3-preload4)*C.z,wCD)
system.add_spring_forcel(k,(q4-preload5)*E.z,wDE)
system.add_spring_forcel(k,(q5-preload6)*F.z,wEF)
```

```
(<physics.force.Force at 0x7f1f89fcb290>,  
 <physics.spring.Spring at 0x7f1f89fcbcd0>)
```

```
#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)
```

```
<physics.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 constraints 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
```

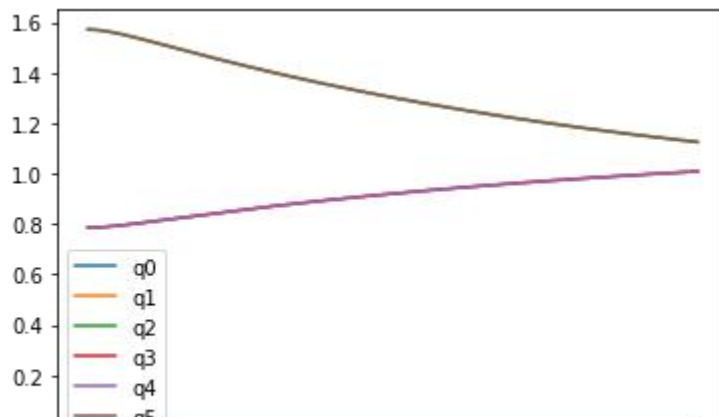
```
2021-03-20 06:55:00,903 - pynamics.system - INFO - solving a = f/r
2021-03-20 06:55:00,931 - pynamics.system - INFO - substituting cc
2021-03-20 06:55:14,755 - pynamics.system - INFO - done solving a
2021-03-20 06:55:14,757 - pynamics.system - INFO - calculating fun
```

```
#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'])
```

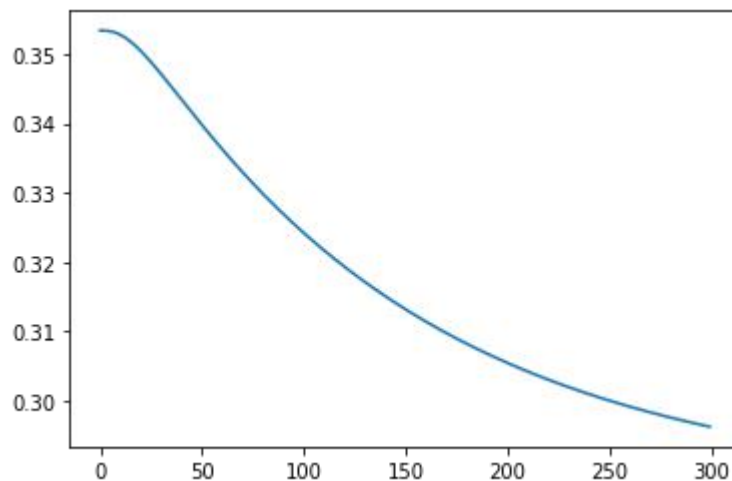

<matplotlib.legend.Legend at 0x7f1f89224990>



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

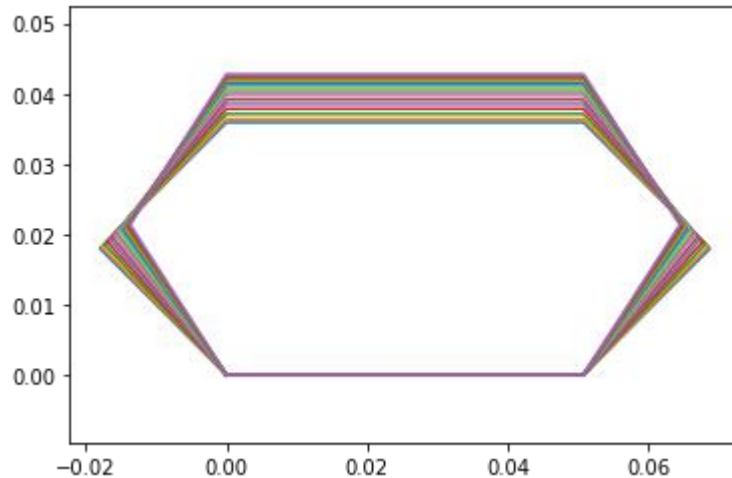


```
#This block of code calculates and plots the motion of the system
points = [pNA,pAB,pBC,pCD,pDE,pEF,pNA]
points_output = PointsOutput(points,system)
y = points_output.calc(states)
points_output.plot_time(20)
```

```

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>

```



```

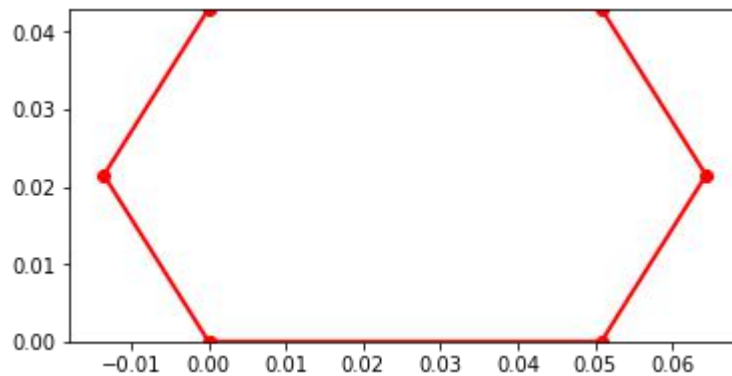
#This block of code produces a figure of the animations end position us
points_output.animate(fps = fps,movie_name = 'render.mp4',lw=2,marker='

```

```

<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 contrctated 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

