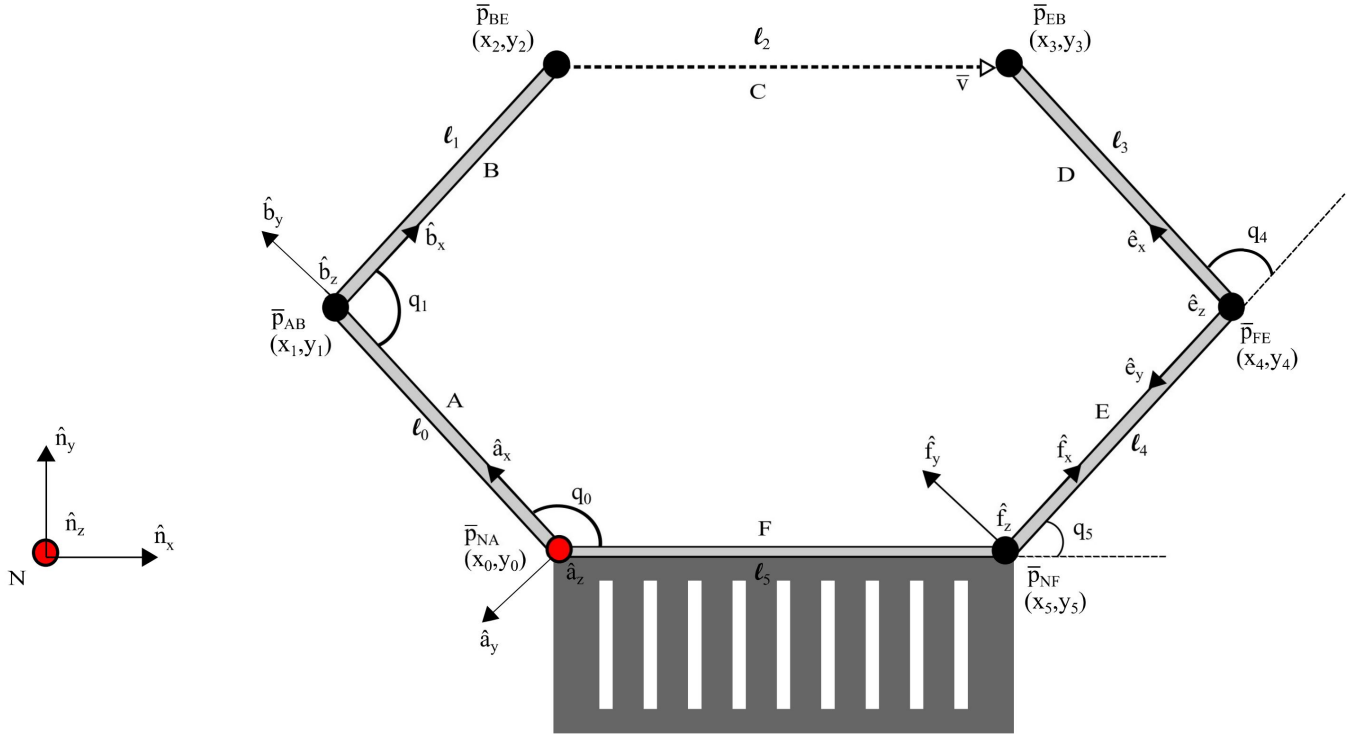
```
!pip install pypoly2tri idealab_tools foldable_robotics pynamics
```

```
Requirement already satisfied: pypoly2tri in /usr/local/lib/python3.7/dist-packages (0.0.3)
Requirement already satisfied: idealab_tools in /usr/local/lib/python3.7/dist-packages (0.0.22)
Requirement already satisfied: foldable_robotics in /usr/local/lib/python3.7/dist-packages (0.0.29)
Requirement already satisfied: pynamics in /usr/local/lib/python3.7/dist-packages (0.0.8)
Requirement already satisfied: imageio in /usr/local/lib/python3.7/dist-packages (from idealab_tools) (2.4.1)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from foldable_robotics) (1.19.5)
Requirement already satisfied: pyyaml in /usr/local/lib/python3.7/dist-packages (from foldable_robotics) (3.13)
Requirement already satisfied: ezdxf in /usr/local/lib/python3.7/dist-packages (from foldable_robotics) (0.15.2)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-packages (from foldable_robotics) (3.2.2)
Requirement already satisfied: shapely in /usr/local/lib/python3.7/dist-packages (from foldable_robotics) (1.7.1)
Requirement already satisfied: scipy in /usr/local/lib/python3.7/dist-packages (from pynamics) (1.4.1)
Requirement already satisfied: sympy in /usr/local/lib/python3.7/dist-packages (from pynamics) (1.7.1)
Requirement already satisfied: pillow in /usr/local/lib/python3.7/dist-packages (from imageio->idealab_tools) (7.0.0)
Requirement already satisfied: pyparsing>=2.0.1 in /usr/local/lib/python3.7/dist-packages (from ezdxf->foldable_robotics) (2.4.7)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib->foldable_robotics) (
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib->foldable_robotics
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-packages (from matplotlib->foldable_robotics) (0.10.
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.7/dist-packages (from sympy->pynamics) (1.2.1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packages (from python-dateutil>=2.1->matplotlib->foldabl
```

1. Create a figure (either in python, in a vector-based drawing program like inkscape or illustrator, or as a solidworks rendering) of your system kinematics.

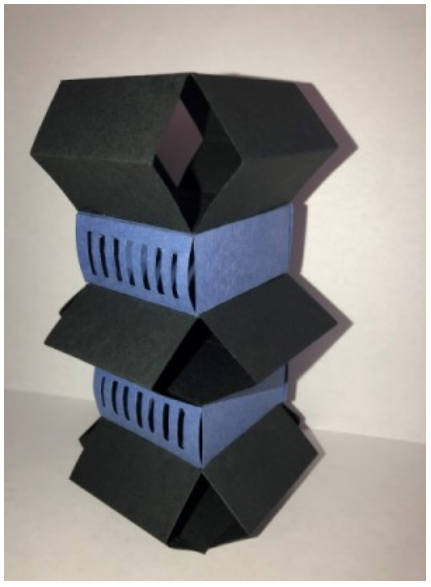
+ Code + Text

Kinematics Model

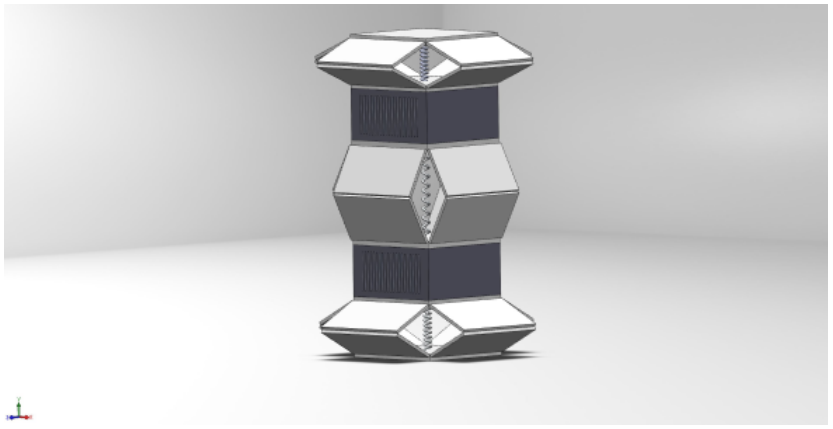


2. Make the device in paper or cardboard. You need an up-to-date model if it has changed from your individual assignments. The paper model should dimensionally match your code.

Paper Model



CAD Model



3. Using a pynamics-based script, develop a kinematic model for your device.

```
%matplotlib inline
```

```
import pynamics
from pynamics.frame import Frame
from pynamics.variable_types import Differentiable,Constant
from pynamics.system import System
from pynamics.output import Output,PointsOutput
import pynamics.integration
import sympy
import numpy
import matplotlib.pyplot as plt
plt.ion()
from math import pi
import scipy.optimize
import math as m
```

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

```
#This block of code defines the lengths (in meters) of each frame (1:1 ratio of paper model)
#1in ~ 0.0254m, but when we use meters here it ruins the results, so use constant values instead
l0 = Constant(1,'l0',system)
```

```
l1 = Constant(1,'l1',system)
l2 = Constant(2,'l2',system)
l3 = Constant(1,'l3',system)
l4 = Constant(1,'l4',system)
l5 = Constant(2,'l5',system)


#This block of code creates dynamic state variables for the angles shown in our kinematics model
q0,q0_d,q0_dd = Differentiable('q0',system) #angle between N and A frames
q1,q1_d,q1_dd = Differentiable('q1',system) #angle between A and B frames
q4,q4_d,q4_dd = Differentiable('q4',system) #angle between E and F frames
q5,q5_d,q5_dd = Differentiable('q5',system) #angle between N and F frames


#This block of code sets the initial guess for the mechanisms starting position (in radians) and velocity (in m/s)
initialvalues = {}
initialvalues[q0] = 135*pi/180  #optimal angle is 135 but change to 155 to show the code solving successfully
initialvalues[q0_d] = 0*pi/180
initialvalues[q1] = -90*pi/180  #optimal angle is -90
initialvalues[q1_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 recieves state variables in the order they are stored
statevariables = system.get_state_variables()


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


#This block of code sets N frame as the newtonian frame (see kinematic diagram from above)
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 about the N frame in the Z direction (0,0,1) by q0 degrees
B.rotate_fixed_axis_directed(A,[0,0,1],q1,system)  #the B frame rotates about the A frame in the Z direction (0,0,1) by q1 degrees
F.rotate_fixed_axis_directed(N,[0,0,1],q5,system)  #the F frame rotates about the N frame in the Z direction (0,0,1) by q5 degrees
E.rotate_fixed_axis_directed(F,[0,0,1],q4,system)  #the E frame rotates about the F frame in the Z direction (0,0,1) by q4 degrees


#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 N reference frame's x direction (0*N.x) and y direction
pAB = pNA + l0*A.x     #pAB position is pNA's position plus l0 units in the direction of ref frame A's x direction
pBE = pAB + l1*B.x     #pBE position is pAB's position plus l1 units in the direction of ref frame B's x direction
pNF = l5*N.x + 0*N.y   #pNF position l5 units in the direction of ref frame N's x direction and 0 units in N's y direction
pFE = pNF + l4*F.x     #pFE position is pNF's position plus l4 units in the direction of ref frame F's x direction
pEB = pFE + l3*E.x     #pEB position is pFE's position plus l3 units in the direction of ref frame E's x direction


#This block of code declares the end-effector (pBE) and lists the frames in the mechanism
pout = pBE
points = [pNA,pAB,pBE,pEB,pFE,pNF] #in order from starting joint to end joint


#This block of code creates a list of initial values
statevariables = system.get_state_variables()
ini0 = [initialvalues[item] for item in statevariables]


#This block of code defines the closed loop kinematics (vectors) of the sarrus mechanism.
eq_vector = pEB - pBE   #vector from pEB to pBE
eq_vector1 = pBE - pNA  #vector from pBE to pNA



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

```
eq.append(((eq_vector).dot(eq_vector))-l2**2)  # (eq_vector) . (eq_vector) = l2**2 -> dot product of a vector with itself gives square
eq.append(((eq_vector).dot(N.x))-l2)           # (eq_vector) . (N.x) = l2 -> dot product of a vector with the unit vector gives the le
eq.append((eq_vector1).dot(N.x))               # (eq_vector1) . (N.x) = 0 -> dot product of vectors perpendicular to each other gives 0

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


#This block of code defines the dependent and independent values
qi = [q0]
qd = [q1,q4,q5] #number of items in qd should equal the number of constraints above


#This block of code recalls and stores link lengths or constants declared earlier
constants = system.constant_values.copy()
defined = dict([(item,initialvalues[item]) for item in qi])
constants.update(defined)


#This block of code substitutes constants into the equation from above
eq = [item.subs(constants) for item in eq]


#This block of code converts to numpy array and sums the error
error = (numpy.array(eq)**2).sum()


#This block of code converts to a function that scipy can use
#Sympy has a "labmdify" function that evaluates an expression
#Scipy uses a different format
f = sympy.lambdify(qd,error)

def function(args):
    return f(*args)


guess = [initialvalues[item] for item in qd]


result = scipy.optimize.minimize(function,guess)


#This block of code solves for the desired mechanism (orange) givin the initial guesses (blue)
#An incorrect initial guess was used on purpose to visualize that the system was solved correctly
ini = []
for item in system.get_state_variables():
    if item in qd:
        ini.append(result.x[qd.index(item)])
    else:
        ini.append(initialvalues[item])


#The following chuncks of code find the jacobian
eq_d = sympy.Matrix(eq_d)

qi = sympy.Matrix([q0_d])
qd = sympy.Matrix([q1_d,q4_d,q5_d])

AA = eq_d.jacobian(qi)
AA.simplify()

BB = eq_d.jacobian(qd)
BB.simplify()

J = -BB.inv()*AA
J.simplify()

qd2 = J*qi

subs = dict([(ii,jj) for ii,jj in zip(qd,qd2)])

vout = pout.time_derivative()
vout = vout.subs(subs)
```

Note: this may take some time to calculate due to the complexity of the system

```
J
```

$$\begin{bmatrix} -\left(\frac{l_0 \sin(q_0)}{\sin(q_0 + q_1)} + l_1\right) \\ \hline l_1 \\ \hline -l_0 \cdot (l_3 \sin(q_4 + q_5) + l_4 \sin(q_5)) \sin(q_1) \\ \hline l_3 l_4 \sin(q_4) \sin(q_0 + q_1) \\ \frac{l_0 \sin(q_1) \sin(q_4 + q_5)}{l_4 \sin(q_4) \sin(q_0 + q_1)} \end{bmatrix}$$

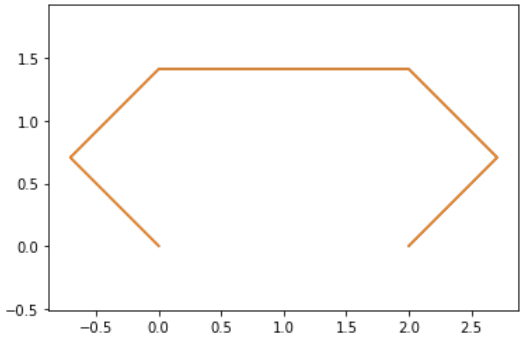
4. Select or Solve for a valid initial condition that represents the system in the middle of a typical gait, when it is both moving and when forces are being applied to it.

We selected our initial condition based on the half way point of the razor clams motion. In order to achieve motion that is in the middle of the typical gait of a razor clam, our team decided to select values that match a sarrus links angles in its half way point (neither contracted, nor extended). This means that the two horizontal plates that translate in the y direction are half way to one another. As a result, a 90 degree angle will form between the top and bottom links.

5. Plot the system in this position.

```
points = PointsOutput(points, constant_values=system.constant_values)
points.calc(numpy.array([ini0,ini]))
points.plot_time()
```

```
2021-02-28 22:57:04,900 - pynamics.output - INFO - calculating outputs
2021-02-28 22:57:04,902 - pynamics.output - INFO - done calculating outputs
<matplotlib.axes._subplots.AxesSubplot at 0x7fa95d2c9c90>
```



6. From your biomechanics-based specifications, define one or more force vector estimates (one for each end effector) that the system should be expected to experience.

Our team will assume that the natural forces applied to the system are gracity and friction. Because the mechanism is traveling upwards, gravity will be pulling down on it. Additionally when the sarrus links extend, friction will be created on the links which will prevent the mechanism from slipping. Based on prior research we know that the average velocity of a razor clam is 1 cm/s.

7. Calculate the force or torque required at the input to satisfy the end-effector force requirements

```
#This block of code defines the force at the end-effector based on data from the biomechanics research conducted by the team
F_max = sympy.Matrix([0,10,0])
```

```
#This block of code solves for the max torque at the end-effector
t_max = J.T*F_max
t_max
```

$$\begin{bmatrix} \frac{-10 \cdot l_0 \cdot (l_3 \sin(q_4 + q_5) + l_4 \sin(q_5)) \sin(q_1)}{l_3 l_4 \sin(q_4) \sin(q_0 + q_1)} \end{bmatrix}$$

8. Estimate the velocity of the end-effector in this configuration. Using the Jacobian, calculate the speed required by the input(s) to achieve that output motion.

```
#This block of code defines the speed at the end-effector based on data from the biomechanics research conducted by the team
v_max = sympy.Matrix([0,20.3,0])
qA_d_max = J.T*v_max
qA_d_max
```

$$\begin{bmatrix} \frac{-20.3 \cdot l_0 \cdot (l_3 \cdot \sin(q_4 + q_5) + l_4 \cdot \sin(q_5)) \cdot \sin(q_1)}{l_3 \cdot l_4 \cdot \sin(q_4) \cdot \sin(q_0 + q_1)} \end{bmatrix}$$

9. Finally, using the two estimates about force and speed at the input, compute the required power in this configuration

```
#This block of code solves for the required power needed to produce the motion
P_max = t_max*qA_d_max.T
P_max
```

$$\begin{bmatrix} \frac{203.0 \cdot l_0^2 \cdot (l_3 \cdot \sin(q_4 + q_5) + l_4 \cdot \sin(q_5))^2 \cdot \sin^2(q_1)}{l_3^2 \cdot l_4^2 \cdot \sin^2(q_4) \cdot \sin^2(q_0 + q_1)} \end{bmatrix}$$

Discussion

1. How many degrees of freedom does your device have? How many motors? If the answer is not the same, what determines the state of the remaining degrees of freedom? How did you arrive at that number?

Our mechanism has has only one degree of freedom. The links will be constrained to allow the output to translate in the y direction. We modeled the mechanism with one sarrus link, but our final design will consist of three. In order to achieve a climbing motion, we need to move each sarrus link individually. As a result, we will be moving our mechanism with three motors (one for each sarrus link).

2. If your mechanism has more than one degree of freedom, please describe how those multiple degrees of freedom will work togehter to create a locomotory gait or useful motion. What is your plan for synchonizing, especially if passive energy storage?

Our system has only one degree of freedom in the y direction. When the motor is running, the y distance between links A and B decreases and the x distance between pBC and pEF will increase. As shown in the video above, when the motor is active the sarrus link expands in the x direction making contact with the outer walls of a crevice. The frictional force with the walls will prevent the sarrus link from slipping. The outer most sarrus links are used to lock into the walls of a crevice. The middle sarrus link is used to increase the distance between the two outer sarrus links. As one of the outer sarrus links locks into place, the middle link will expand in the y direction, and the other outer link will lock. As a result, the device can manipulate each of the sarrus links contraction and extension to create a translation in the y direction, similar to that of a razor clam. A motor will control the input of the three sarrus links. A spring will be used to extend the sarrus links and the motor will be used to contract them.

3. How did you estimate your expected end-effector forces?

After revising our concept, our team's robot kinematics was modeled after the razor clam. We used prior research on the biomechanics of a razor clam to determine speed, forces, etc. The primary difference between the bioinspired animal and the robot is the razor clam burrows into sand while the team's robot is climbing vertically between two surfaces. The top, or end effector, of the robot would be moving vertically with only gravity as the opposing force. The end effector force can be expressed in the simulation as a function of velocity due to 'Force = Mass * Acceleration', whereas acceleration is substituted for given velocity over time.

4. How did you estimate your expected end-effector speeds?

We found the velocity of the end effector through the python simulation shown above. The geometric relationships for angular speed lead us to determine the output velocity at a specific input. We estimated velocity values based on the values we determined from the biomechanics

assignment. From this research we found that the razor clam can move upward a distance of 5mm anywhere from 1.6 to 7.5 seconds. We took the average of this speed and used it as a reference.