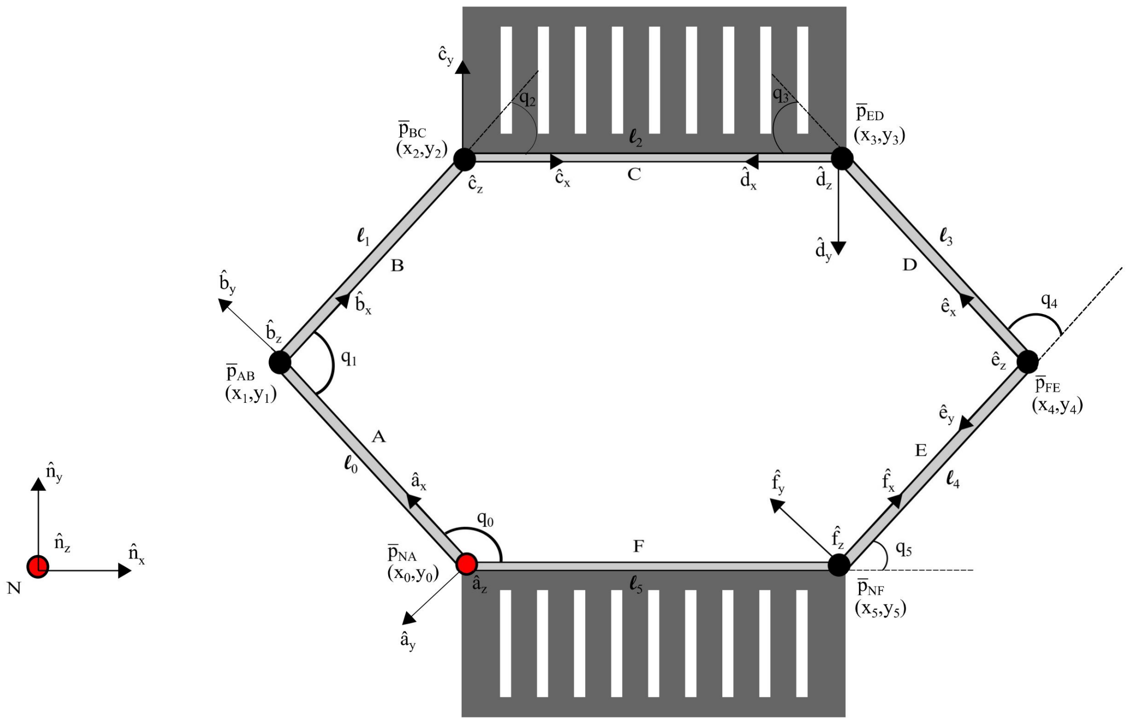
Dynamics I

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

```
    Collecting pypoly2tri
      Downloading https://files.pythonhosted.org/packages/11/33/079f13
    Collecting idealab_tools
      Downloading https://files.pythonhosted.org/packages/c7/00/d34a53
    Collecting foldable_robotics
      Downloading https://files.pythonhosted.org/packages/5d/af/a1cef6
    Collecting pynamics
      Downloading https://files.pythonhosted.org/packages/4e/ef/c2d877
         |████████████████████████████████| 92kB 4.3MB/s
    Requirement already satisfied: imageio in /usr/local/lib/python3.7
    Collecting ezdxf
      Downloading https://files.pythonhosted.org/packages/6c/67/1a6715
         |████████████████████████████████| 1.8MB 13.3MB/s
    Requirement already satisfied: shapely in /usr/local/lib/python3.7
    Requirement already satisfied: matplotlib in /usr/local/lib/python
    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/pyth
    Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib
    Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/pyth
    Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.
    Installing collected packages: pypoly2tri, idealab-tools, ezdxf, f
    Successfully installed ezdxf-0.15.2 foldable-robotics-0.0.29 ideal
```

Dynamics Model



```
%matplotlib inline
```

```
use_constraints = False
```

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

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

```
system = System()
pynamics.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 ea
l1 = Constant(0.0254,'l1',system) #1in ~ 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) #1g ~ 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 pr
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 show
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 tl
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 c
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 direct
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
pNF = l5*N.x + 0*N.y   #pNF position l5 units in the direction of ref fr
pFE = pNF + l4*F.x     #pFE position is pNF's position plus l4 units in
pED = pFE + l3*E.x     #pED position is pFE's position plus l3 units in


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

```
#This block of code computes and returns the angular velocity between fr
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 mas
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 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*wNF,wNF)
system.addforce(-b*wFE,wFE)
system.addforce(-b*wED,wED)
```

```
    <pynamics.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 angluar velocity vector (depending
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)
```

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

```
#This block of code globally applies the force of gravity to all particl
system.addforcegravity(-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
```

```
[illegible]
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 - pynamics.system - INFO - getting dynamic
```

```
#This block of code solves the system of equations F=ma plus any constra
#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 fo
#The below function inverts the mass matrix numerically every time step.
func1,lambda1 = system.state_space_post_invert(f,ma,eq_dd,return_lambda
```

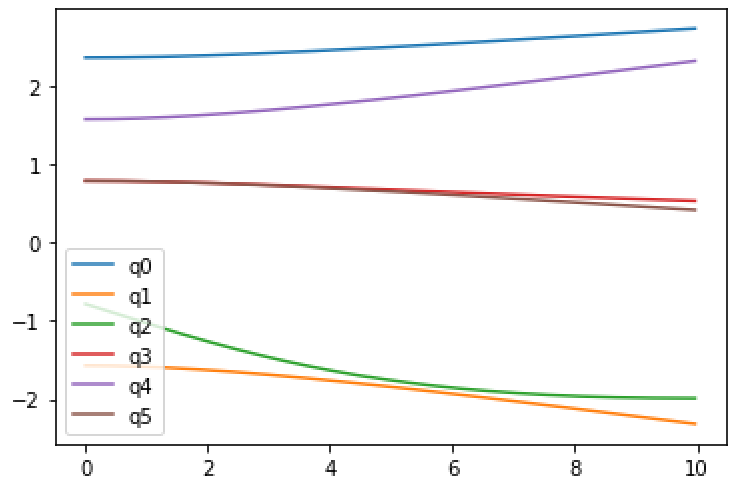
```
    2021-03-01 02:37:46,632 - pynamics.system - INFO - solving a = f/m
    2021-03-01 02:37:46,643 - pynamics.system - INFO - substituting co
    2021-03-01 02:37:47,329 - pynamics.system - INFO - done solving a
    2021-03-01 02:37:47,330 - pynamics.system - INFO - calculating fun
```

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

```
    2021-03-01 02:37:47,635 - pynamics.integration - INFO - beginning
    2021-03-01 02:37:47,640 - pynamics.system - INFO - integration at
    2021-03-01 02:37:48,241 - pynamics.system - INFO - integration at
    2021-03-01 02:37:48,371 - pynamics.integration - INFO - finished i
```

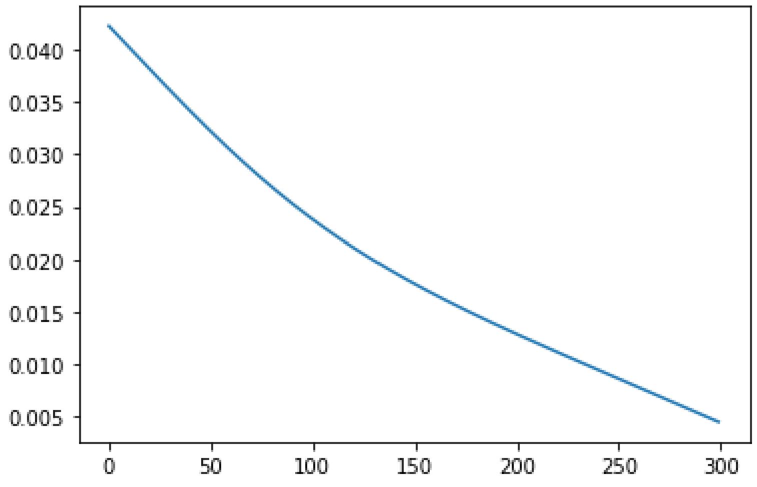
```
#This block of code calculates and plots a variety of data from the prev
plt.figure()
artists = plt.plot(t,states[:,:6])
plt.legend(artists,['q0','q1','q2', 'q3', 'q4', 'q5'])
```

```
    <matplotlib.legend.Legend at 0x7f6af34ef750>
```



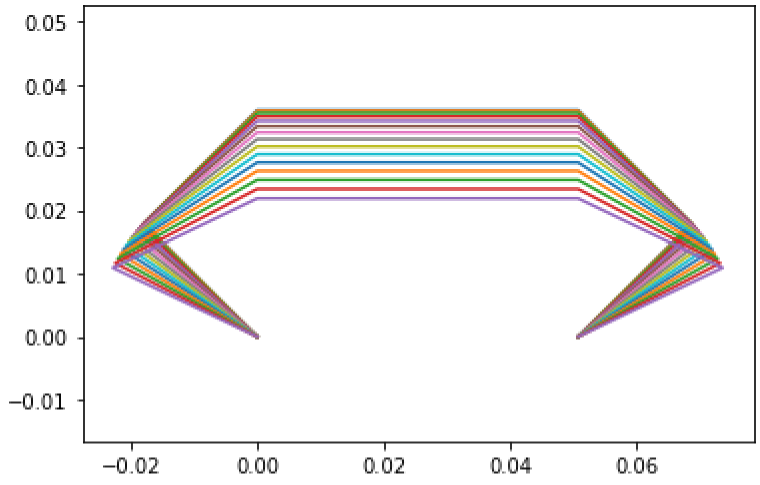
```
#This block of code calculates amd plots the energy of system
KE = system.get_KE()
PE = system.getPEGravity(pNA) - system.getPESprings()
energy_output = Output([KE-PE],system)
energy_output.calc(states)
energy_output.plot_time()
```

```
    2021-03-01 02:37:48,722 - pynamics.output - INFO - calculating out
    2021-03-01 02:37:48,741 - pynamics.output - INFO - done calculatin
```



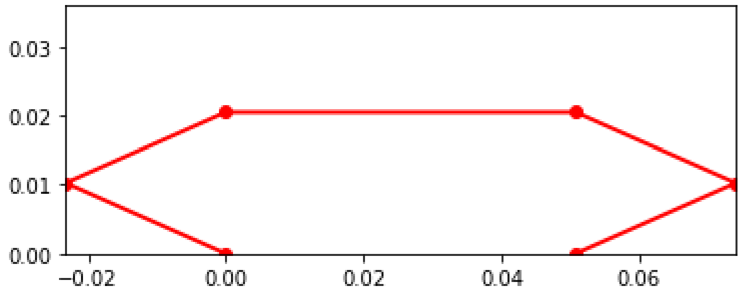
```
#This block of code calculates and plots the motion of the system
points = [pNA, pAB, pBC, pED, pFE, pNF]
points_output = PointsOutput(points,system)
y = points_output.calc(states)
points_output.plot_time(20)
```

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



```
#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 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.