Mallard Duck Wing Dynamics

Team 3

Setup

In [1]:

```
#pip install pynamics
```

In [2]:

```
%matplotlib inline
import scipy.optimize
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 sympy
import matplotlib.pyplot as plt
from matplotlib.patches import Arc
plt.ion()
from math import pi, sin, cos, tan, asin, acos, atan, degrees, radians, pi
```

Procedure

Build 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.**

All units are in SI

1. **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).**

Masses and inertias taken from SolidWorks "corrugated paper" material properties with 5mm thickness

In [3]:

```
# Create a pynamics system
system = System()
pynamics.set_system(__name__,system)
```

In [4]:

```
# Length variables (numerical)
```

```
lA_num = 0.2
lB_num = 0.2
lC_num = 0.22
lD_num = 0.2
lE_num = 0.04

# Lengths
lA = Constant(lA_num,'lA',system)
lB = Constant(lB_num,'lB',system)
lC = Constant(lC_num,'lC',system)
lD = Constant(lD_num,'lD',system)
lE = Constant(lE_num,'lE',system)
```

In [5]:

```
# Masses
mA = Constant(0.0312,'mA',system)
mB = Constant(0.0312,'mB',system)
mC = Constant(0.0312,'mC',system)
mD = Constant(0.039,'mD',system)

# Joint preloads
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)

# Inertia
Ixx_A = Constant(0.00042,'Ixx_A',system)
Iyy_A = Constant(0.00083,'Iyy_A',system)
Izz_A = Constant(0.00042,'Izz_A',system)
Ixx_B = Constant(0.00042,'Ixx_B',system)
Iyy_B = Constant(0.00083,'Iyy_B',system)
Izz_B = Constant(0.00042,'Izz_B',system)
Ixx_C = Constant(0.00042,'Ixx_C',system)
Iyy_C = Constant(0.00083,'Iyy_C',system)
Izz_C = Constant(0.00042,'Izz_C',system)
Ixx_D = Constant(0.00052,'Ixx_D',system)
Iyy_D = Constant(0.00133,'Iyy_D',system)
Izz_D = Constant(0.00081,'Izz_D',system)
```

In [6]:

```
# gravity
g = Constant(9.81,'g',system)
```

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

b=0 and b=1 cases presented below. b=1 is the default value but turned out to be a good number to stabilize the system.

In [7]:

```
b = Constant(1e1,'b',system)
```

1. **(Optional): Adjust joint stiffness values so that your system looks more realistic.**

The links have so little mass that k was reduced to 0.01 to show significant motion

In [8]:

```
k = Constant(1e-2,'k',system)
```

In [9]:

```
# Integration Tolerance
tol = 1e-12
```

In [10]:

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

In [11]:

```
# Create differentiable state variables
qA,qA_d,qA_dd = Differentiable('qA',system)
qB,qB_d,qB_dd = Differentiable('qB',system)
qC,qC_d,qC_dd = Differentiable('qC',system)
qD,qD_d,qD_dd = Differentiable('qD',system)
```

In [12]:

```
# initial values
# qC = 11.97 creates fully extended, horizontal wing
initialvalues = {}
initialvalues[qA]=0*pi/180
initialvalues[qA_d]=0*pi/180
initialvalues[qB]=0*pi/180
initialvalues[qB_d]=0*pi/180

initialvalues[qC]=11.97*pi/180
initialvalues[qC_d]=0*pi/180
initialvalues[qD]=0*pi/180
initialvalues[qD_d]=0*pi/180
```

In [13]:

```
# Retrieve state variables in the order they are stored in the system
# Create a list of initial values ini0 in the order of the system's state variables
statevariables = system.get_state_variables()
ini0 = [initialvalues[item] for item in statevariables]
```

In [14]:

```
# Create frames
N = Frame('N')
A = Frame('A')
B = Frame('B')
C = Frame('C')
D = Frame('D')
```

In [15]:

```
# Declare N as the Newtonian (fixed) frame
system.set_newtonian(N)
```

In [16]:

```
# Rotate other frames about their local Z axes.
# Not global q
A.rotate_fixed_axis_directed(N,[0,0,1],qA,system)
B.rotate_fixed_axis_directed(A,[0,0,1],qB,system)
C.rotate_fixed_axis_directed(N,[0,0,1],qC,system)
D.rotate_fixed_axis_directed(C,[0,0,1],qD,system)
```

In [17]:

```
# Define rigid body kinematics
# position vectors
p0 = 0*N.x #Normal
p1 = p0 + lA*A.x #Link A
p2A = p1 + lB*B.x #Link B - OUTPUT
```

```
p4 = p0 + lE*N.y #Link E
p3 = p4 + lC*C.x #Link C
p2C = p3 + lD*D.x #Link D - OUTPUT
```

In [18]:

```
# CoM vectors
pAcom=p0 + lA/2*A.x #Link A
pBcom=p1 + lB/2*B.x #Link B
pCcom=p4 + lC/2*C.x #Link C
pDcom=p3 + lD/2*D.x #Link D
```

In [19]:

```
# Angular Velocity
wNA = N.getw_(A)
wAB = A.getw_(B)
wNC = N.getw_(C)
wCD = C.getw_(D)
```

In [20]:

```
# Build inertia tensors
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)

BodyA = Body('BodyA',A,pAcom,mA,IA,system)
BodyB = Body('BodyB',B,pBcom,mB,IB,system)
BodyC = Body('BodyC',C,pCcom,mC,IC,system)
BodyD = Body('BodyD',D,pDcom,mD,ID,system)
#BodyC = Particle(pCcm,mC,'ParticleC',system)
```

1. **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.**

In [21]:

```
# Forces
system.addforce(-b*wNA,wNA)
system.addforce(-b*wAB,wAB)
system.addforce(-b*wNC,wNC)
system.addforce(-b*wCD,wCD)
```

Out[21]:

```
<pynamics.force.Force at 0x20fa55ef9a0>
```

In [22]:

```
# Spring Forces
# Not global q
system.add_spring_force1(k,(qA-preload1)*N.z,wNA)
system.add_spring_force1(k,(qB-preload2)*A.z,wAB)
system.add_spring_force1(k,(qC-preload3)*N.z,wNC)
system.add_spring_force1(k,(qD-preload4)*C.z,wCD)
```

Out[22]:

```
(<pynamics.force.Force at 0x20fa57a2c70>,
 <pynamics.spring.Spring at 0x20fa57a2e80>)
```

In [23]:

```
# Gravity
system.addforcegravity(-g*N.y)
```

1. **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.

In [24]:

```
# Constraints
eq_vector = p2A-p2C
eq = []
eq.append((eq_vector).dot(N.x))
eq.append((eq_vector).dot(N.y))
eq_d=[(system.derivative(item)) for item in eq]
eq_dd=[(system.derivative(item)) for item in eq_d]
```

Solve for valid initial condition

In [25]:

```
# Declare a list of points that will be used for plotting
points = [p0, p1, p2A, p3, p4]
```

In [26]:

```
# Identify independent and dependent variables
qi = [qA, qC]
qd = [qB, qD]
```

In [27]:

```
# Create a copy of symbolic constants dictionary and add the initial value of qi to it
constants = system.constant_values.copy()
defined = dict([(item,initialvalues[item]) for item in qi])
constants.update(defined)
```

In [28]:

```
# Substitute constants in equation
eq = [item.subs(constants) for item in eq]
```

In [29]:

```
# Convert to numpy array
# Sum the error
error = (numpy.array(eq)**2).sum()
```

In [30]:

```
# Convert to a function that scipy can use.
# Sympy has a "labmdify" function that evaluates an expression, but scipy needs a slightl
y different format.
f = sympy.lambdify(qd,error)

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

In [31]:

```
# Take the derivative of the equations to linearize with regard to the velocity variables
guess = [initialvalues[item] for item in qd]
result = scipy.optimize.minimize(function,guess)
if result.fun>1e-3:
    raise(Exception("out of tolerance"))
ini = []
for item in system.get_state_variables():
    if item in qd:
        ini.append(result.x[qd.index(item)])
    else:
        ini.append(initialvalues[item])
```

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

```
2021-02-28 15:40:08,393 - pynamics.output - INFO - calculating outputs
2021-02-28 15:40:08,394 - pynamics.output - INFO - done calculating outputs
```

Out[31]:

```
array([[[0.        , 0.        ],
        [0.2       , 0.        ],
        [0.4       , 0.        ],
        [0.21521639, 0.08562789],
        [0.        , 0.04      ]],

       [[0.        , 0.        ],
        [0.2       , 0.        ],
        [0.39981349, 0.00863542],
        [0.21521639, 0.08562789],
        [0.        , 0.04      ]]])
```

In [32]:

```
# Valid initial condition
x = numpy.array([points.y[1][0][0], points.y[1][1][0], points.y[1][2][0], points.y[1][3
][0], points.y[1][4][0],0])
y = numpy.array([points.y[1][0][1], points.y[1][1][1], points.y[1][2][1], points.y[1][3
][1], points.y[1][4][1],0])
#prettyplot(x,y)
```

Solution: b=1

1. **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**

In [33]:

```
# F=ma
f,ma = system.getdynamics()
```

```
2021-02-28 15:40:08,428 - pynamics.system - INFO - getting dynamic equations
```

In [34]:

```
# Acceleration
func1,lambda1 = system.state_space_post_invert(f,ma,eq_dd,return_lambda = True)
```

```
2021-02-28 15:40:08,604 - pynamics.system - INFO - solving a = f/m and creating function
2021-02-28 15:40:08,606 - pynamics.system - INFO - substituting constrained in Ma-f.
2021-02-28 15:40:08,756 - pynamics.system - INFO - done solving a = f/m and creating func
tion
2021-02-28 15:40:08,756 - pynamics.system - INFO - calculating function for lambdas
```

In [35]:

```
# Integrate
states=pynamics.integration.integrate(func1,ini,t,rtol=tol,atol=tol, args=({'constants':s
ystem.constant_values},))
```

```
2021-02-28 15:40:08,777 - pynamics.integration - INFO - beginning integration
2021-02-28 15:40:08,777 - pynamics.system - INFO - integration at time 0000.00
2021-02-28 15:40:09,048 - pynamics.integration - INFO - finished integration
```

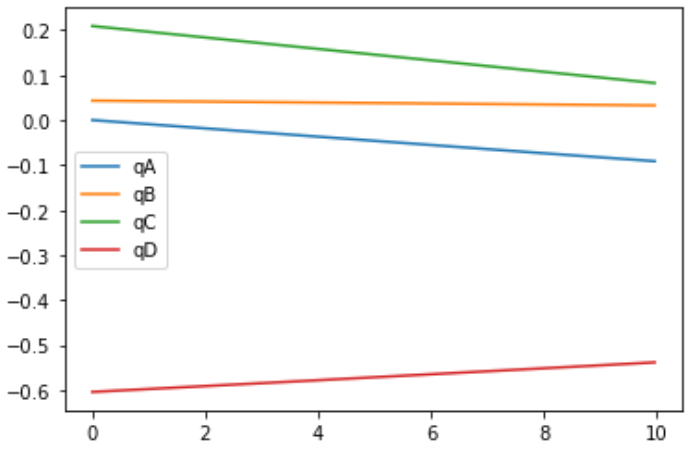
In [36]:

```
# Outputs
plt.figure()
artists = plt.plot(t,states[:,:4])
```

```
plt.legend(artists,['qA','qB','qC','qD'])
```

Out[36]:

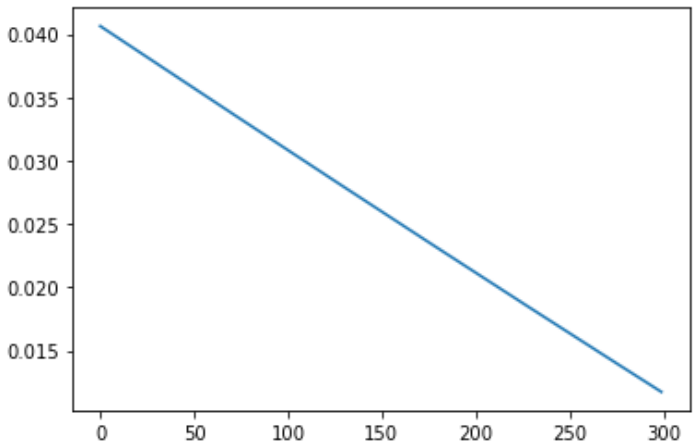
```
<matplotlib.legend.Legend at 0x20fa58d9250>
```



In [37]:

```
# Energy
KE = system.get_KE()
PE = system.getPEGravity(p0) - system.getPESprings()
energy_output = Output([KE-PE],system)
energy_output.calc(states)
energy_output.plot_time()
```

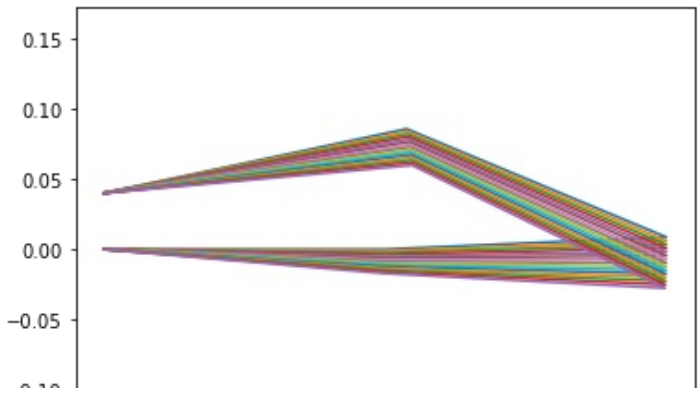
```
2021-02-28 15:40:09,220 - pynamics.output - INFO - calculating outputs
2021-02-28 15:40:09,228 - pynamics.output - INFO - done calculating outputs
```

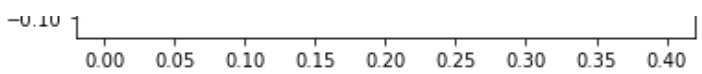


In [38]:

```
# Motion
points = [p0,p1,p2A,p3,p4]
points_output = PointsOutput(points,system)
y = points_output.calc(states)
points_output.plot_time(20)
```

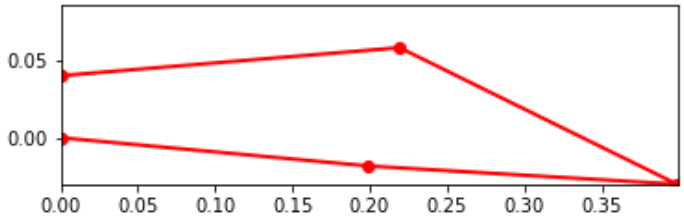
```
2021-02-28 15:40:09,331 - pynamics.output - INFO - calculating outputs
2021-02-28 15:40:09,339 - pynamics.output - INFO - done calculating outputs
```





In [39]:

```
# Motion Animation
points_output.animate(fps = fps,movie_name = 'render.mp4',lw=2,marker='o',color=(1,0,0,1
),linestyle='-')
```



In [40]:

```
from matplotlib import animation, rc
from IPython.display import HTML
HTML(points_output.anim.to_html5_video())
```

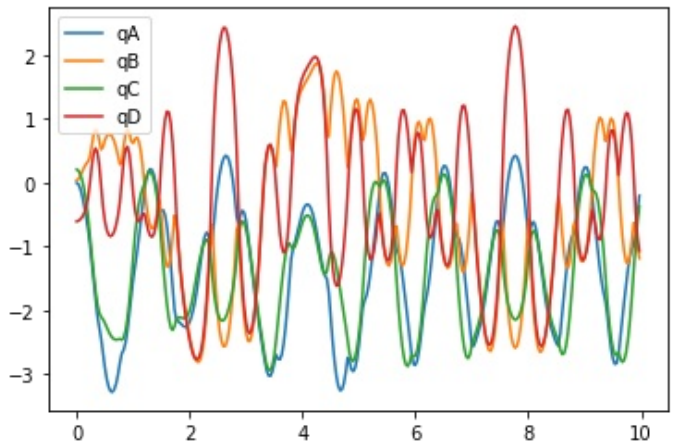
Out[40]:

Your browser does not support the video tag.

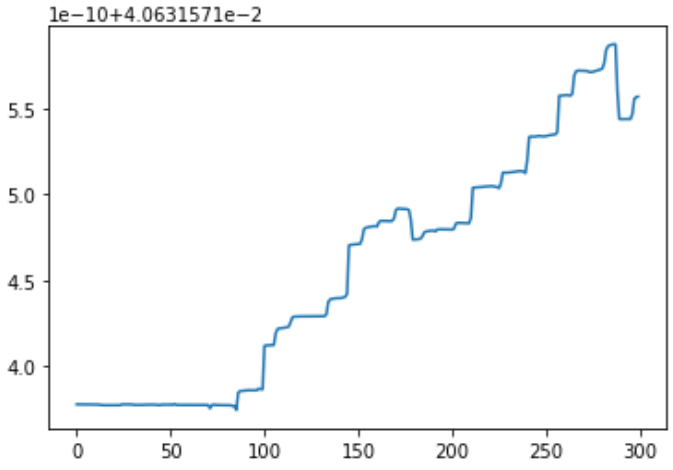
With nonzero damping coefficient, the wing sags and then settles.

Solution: b=0

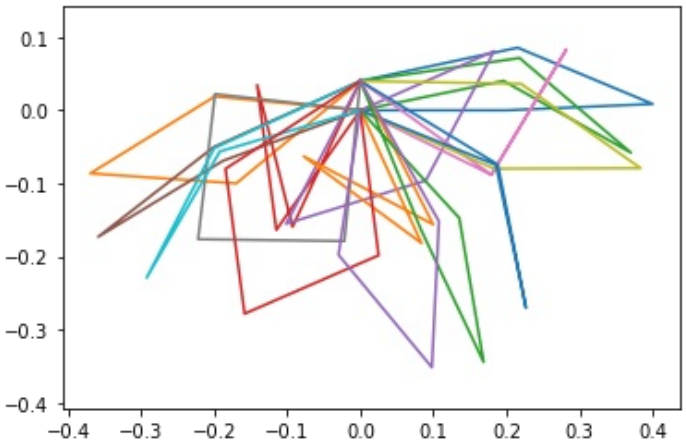
States



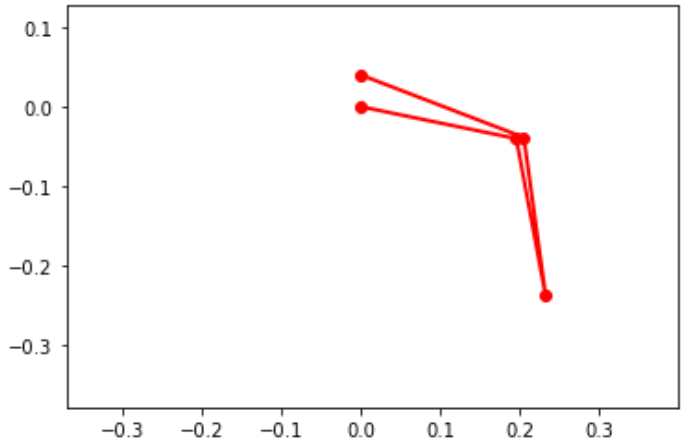
Energy



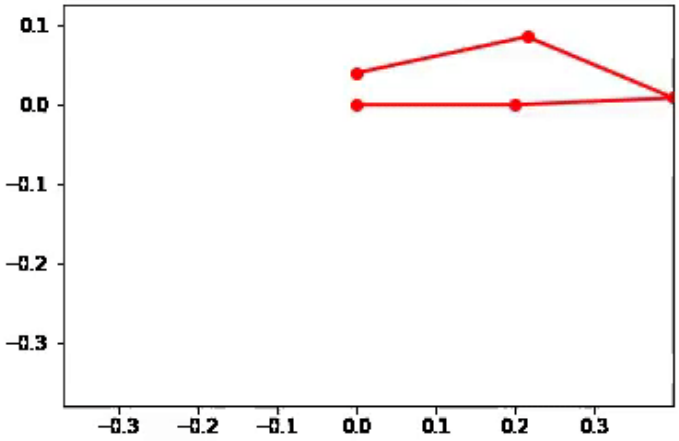
Frames



Final Frame



Animation



With zero damping coefficient, the wing swings chaotically