WORKSHOP #1

RYAN MATTHIESSEN

Lehigh University

CSE460 – Spring 2022

GitHub Repo Link

Problem #1

The derivation for the ellipse's control policy is shown below:

Ellipse Center: $\begin{bmatrix} 3 \\ 2 \end{bmatrix}$

Ellipse position governed by: $\begin{bmatrix} M\cos(t) \\ m\sin(t) \end{bmatrix}$ where M is the major axis, m is the minor axis $\Rightarrow$ Ellipse position: $\begin{bmatrix} 4\cos(t) \\ 2\sin(t) \end{bmatrix}$

rotational matrix: (30° ccw) $\begin{bmatrix} \cos 30 & -\sin 30 \\ +\sin 30 & \cos 30 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{3}}{2} & -1/2 \\ +1/2 & \frac{\sqrt{3}}{2} \end{bmatrix}$

Rotate ellipse w/ multiplication: $\begin{bmatrix} \frac{\sqrt{3}}{2} & -1/2 \\ +1/2 & \frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} 4\cos(t) \\ 2\sin(t) \end{bmatrix} = \begin{bmatrix} 2\sqrt{3}\cos(t) - \sin(t) \\ +2\cos(t) + \sqrt{3}\sin(t) \end{bmatrix}$

Differentiate to find control policy: $\begin{bmatrix} -2\sqrt{3}\sin(t) - \cos(t) \\ -2\sin(t) + \sqrt{3}\cos(t) \end{bmatrix}$

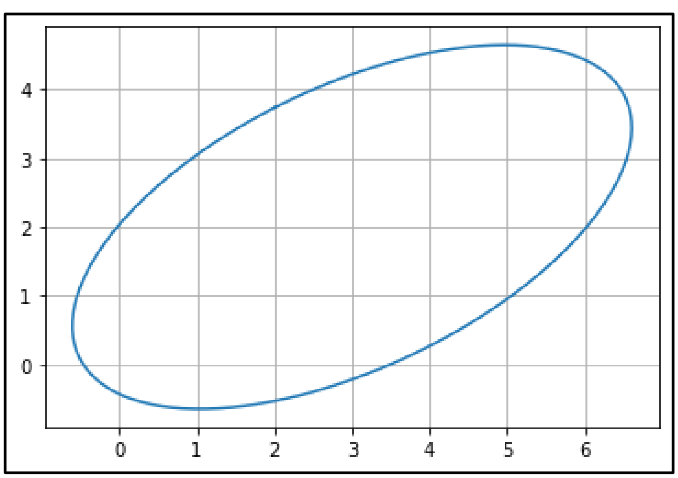
Add center point: $\boxed{\begin{bmatrix} -2\sqrt{3}\sin(t) - \cos(t) + 3 \\ -2\sin(t) + \sqrt{3}\cos(t) + 2 \end{bmatrix} = u(t)}$

The full code implementing the control policy can be found HERE.

The control policy code snippet and its output are below.

Question 1: Rotated Ellipse Control Policy

```
tf = 2*pi    # Simulation time
Δt = 0.05    # Time step
time = linspace(0,tf, int(tf / Δt) + 1)  # Time interval

## Initial Conditions
p = array([-2*sqrt(3)*sin(0) - cos(0) + 3, -2*sin(0) + sqrt(3)*cos(0) + 2]) #robot location (t=0)
p_log = [copy(p)]

## Update position array for each time (t)
for t in time:
    y = sense(p)

    # Desired point to achieve (changes based on equation which is a function of t)
    p_d = [-2*sqrt(3)*sin(t) - cos(t) + 3, -2*sin(t) + sqrt(3)*cos(t) + 2]

    # Implement P-Controller to find policy, progress and update position array
    u = Pcontrol(t, y, p_d)
    p = simulate(Δt, p, u)
    p_log.append(copy(p))
p_log = array(p_log)
```



Problem #2

The derivation for the starfishes' control policy is shown below.

The control policy of the starfish is governed by: $r = \sin(k\theta) + 2$

where $k = \#$ petals (5)

$\therefore$ The equation is $r = \sin(5\theta) + 2$ in polar coordinates

Turn polar coordinates into cartesian: $x = r\cos\theta \Rightarrow r = \frac{x}{\cos\theta}$, $y = r\sin\theta \Rightarrow r = \frac{y}{\sin\theta}$ sub in: $\frac{x}{\cos\theta} = \sin(5\theta) + 2$, $\frac{y}{\sin\theta} = \sin(5\theta) + 2$

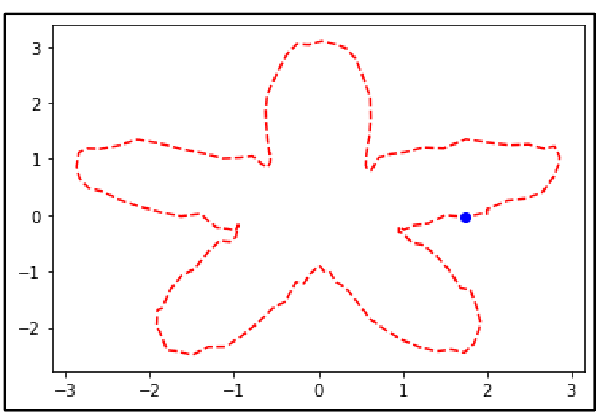
Control policy when we solve and replace $\theta = t$ is:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \boxed{\begin{bmatrix} \sin(5t)\cos(t) + 2\cos(t) \\ \sin(5t)\sin(t) + 2\sin(t) \end{bmatrix} = u(t)}$$

The full code implementing the control policy can be found HERE. The control policy snippet and its output for both parts are below.

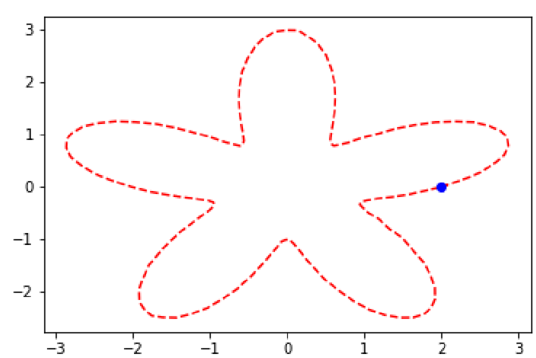
Question 2a: Random vertical wind without compensation

```
tf = 2*pi     # Simulation time
Δt = 0.05     # Time step
time = linspace(0.,tf, int(tf / Δt) + 1)  # Time interval

## Initial Conditions
p = array([sin(0)*cos(0)+2*cos(0), sin(0)*sin(0)+2*sin(0)]) #robot location (t=0)
p_log = [copy(p)]

## Update position array for each time (t)
for t in time:
    y = sense(p)

    # Desired point to achieve (changes based on equation which is a function of t)
    p_d = [sin(5*t)*cos(t)+2*cos(t), sin(5*t)*sin(t)+2*sin(t)]

    # Implement PI-Controller to find policy, progress and update position array
    u = Pcontrol(t, y, p_d)
    p = simulate(Δt, p, u + [0, random.randint(0,5)]) # Added random vertical wind
    p_log.append(copy(p))
p_log = array(p_log)
```



Question 2b: Random vertical wind with compensating PI-Controller

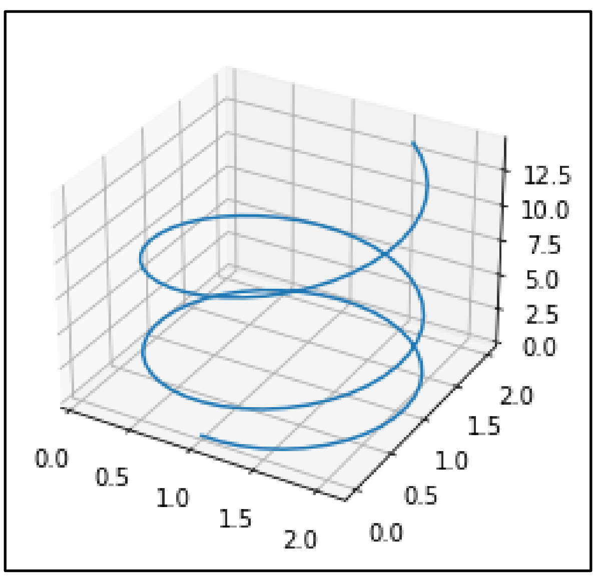
```
tf = 2*pi     # Simulation time
Δt = 0.05     # Time step
time = linspace(0.,tf, int(tf / Δt) + 1)  # Time interval

## Initial Conditions
p = array([sin(0)*cos(0)+2*cos(0), sin(0)*sin(0)+2*sin(0)]) #robot location (t=0)
p_log = [copy(p)]

## Update position array for each time (t)
for t in time:
    y = sense(p)

    # Desired point to achieve (changes based on equation which is a function of t)
    p_d = [sin(5*t)*cos(t)+2*cos(t), sin(5*t)*sin(t)+2*sin(t)]

    # Implement PI-Controller to find policy, progress and update position array
    u = PIcontrol(t, y, p_d,[0, random.randint(0,5)]) # Added random vertical wind
    p = simulate(Δt, p, u)
    p_log.append(copy(p))
p_log = array(p_log)
```



Problem #3

The full code implementing the helix can be found HERE. The main simulator 3D extension changes (1, 2, 3, 4) and the helix trajectory are below:

```
### (1) Control policy added z-coordinate
def control(t, y):

    # Helix (control policy is the version found in our textbook: top of page 12)
    ux = cos(t)
    uy = sin(t)
    uz = (2*t)/(5*pi)

    return array([ux, uy, uz])


### (2) Animate added p_log[:,2] and p_log[t,2] for z-coordinates
def animate(t):
    ax.clear()

    # Path
    plot(p_log[:,0], p_log[:,1], p_log[:,2], 'r--')

    # Initial conditions
    plot(p_log[t,0], p_log[t,1], p_log[t,2], 'bo')

### (3) Initial conditions added z-coordinate
p = array([cos(0), sin(0), (2*0)/(5*pi)]) #robot location (t=0)


## (4) Plotting controls added 3d projection and plot of z-coordinate (p_log[:,2])
fig = plt.figure()
ax = plt.axes(projection ='3d')
plot(p_log[:,0], p_log[:,1], p_log[:,2])

fig, ax = plt.subplots()
anim = animation.FuncAnimation(fig, animate, frames=len(time), interval=60)
HTML(anim.to_jshtml())
```



Problem #4

The full code implementing the polyline trajectory can be found HERE. The trajectory initialization (code & equation), polyline creation method, obstacle creation method, and code output are below.

Straight-Line Trajectory Creation

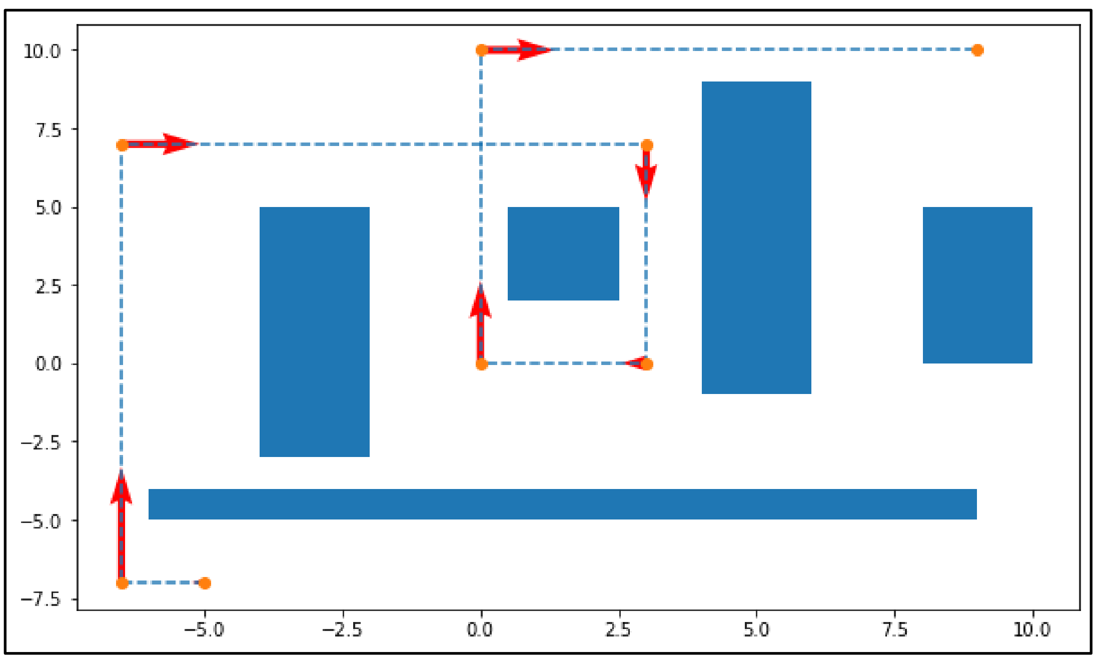
```python
# Waypoints
p1 = [-5.,-7.] # Starting Position
p2 = [-6.5,-7.]
p3 = [-6.5,7.]
p4 = [3.,7.]
p5 = [3.,0.]
p6 = [0.,0.]
p7 = [0.,10.]
p8 = [9.,10.] # Final Position

# Velocities
v0 = [0,0]
v1 = [-1.5,0]
v2 = [0,14]
v3 = [9.5,0]
v4 = [0,-7]
v5 = [-3,0]
v6 = [0,10]
v7 = [9,0]
v8 = [0,0]

# Time
t0 = 0
t1 = 1
t2 = 2
t3 = 3
t4 = 4
t5 = 5
t6 = 6
t7 = 7
t8 = 8
```

$$\gamma(t) = \begin{cases} \begin{pmatrix}-1.5\\0\end{pmatrix} * t + \begin{pmatrix}-5\\-7\end{pmatrix} & 0 < t \leq 1 \\ \begin{pmatrix}0\\14\end{pmatrix} * (t-1) + \begin{pmatrix}-6.5\\-7\end{pmatrix} & 1 \leq t < 2 \\ \begin{pmatrix}9.5\\0\end{pmatrix} * (t-2) + \begin{pmatrix}-6.5\\7\end{pmatrix} & 2 \leq t < 3 \\ \begin{pmatrix}0\\-7\end{pmatrix} * (t-3) + \begin{pmatrix}3\\7\end{pmatrix} & 3 \leq t < 4 \\ \begin{pmatrix}-3\\0\end{pmatrix} * (t-4) + \begin{pmatrix}3\\0\end{pmatrix} & 4 \leq t < 5 \\ \begin{pmatrix}0\\10\end{pmatrix} * (t-5) + \begin{pmatrix}0\\0\end{pmatrix} & 5 \leq t < 6 \\ \begin{pmatrix}9\\0\end{pmatrix} * (t-6) + \begin{pmatrix}0\\10\end{pmatrix} & 6 \leq t < 7 \\ \begin{pmatrix}0\\0\end{pmatrix} * (t-7) + \begin{pmatrix}9\\10\end{pmatrix} & 7 \leq t < \infty \end{cases}$$

*Trajectories are separated by colors

```python
# Straight line trajectories
def point_to_point_traj(x1, x2, v1, v2, delta_t):
    numPts = 50
    t = np.linspace(0, delta_t, numPts)
    a0 = x1
    a1 = v1

    line = a0 + a1*t
    derivative = [a1] * numPts
    return line, derivative
```

```python
# Setting up blocking rectangle points as [bot. left x, bot. left y, top right x, top right y]
def rectangle_set_up(ax):
    rect1 = [4.,-1.,6.,9.]
    rect2 = [0.5,2.,2.5,5.]
    rect3 = [-6.,-5.,9.,-4.]
    rect4 = [8.,0.,10.,5.]
    rect5 = [-4.,-3.,-2.,5.]
    rect = [rect1,rect2,rect3,rect4,rect5]

    # Compute rectangles and add them to ax. Then return the plot
    for i in range(5):
        Rectangles = Rectangle((rect[i][0],rect[i][1]),rect[i][2]-rect[i][0],rect[i][3]-rect[i][1])
        ax.add_patch(Rectangles)

    return ax.plot()
```



Problem #5

The full code implementing the spline trajectory can be found HERE. The waypoint/velocity/time initializations and obstacle creation method are the same as problem #4 and will not be displayed again, however, the spline creation method, and code output are below.

```
def point_to_point_traj(x1, x2, v1, v2, delta_t):
    numPts = 100
    t = np.linspace(0, delta_t, numPts)
    a0 = x1
    a1 = v1
    a2 = (3*x2 - 3*x1 - 2*v1*delta_t - v2 * delta_t) / (delta_t**2)
    a3 = (2*x1 + (v1 + v2) * delta_t  - 2 * x2) / (delta_t**3)

    polynomial = a0 + a1 * t + a2 * t**2 + a3 * t**3
    derivative = a1 + 2*a2 * t + 3 * a3 * t**2
    return polynomial, derivative


def piecewise2D (X,Y, Vx, Vy, T):
    theta_x, theta_y, dx, dy = [], [], [], []

    for i in range(len(P)-1):
        theta_xi, dxi = point_to_point_traj(X[i], X[i+1], Vx[i], Vx[i+1], T[i+1] - T[i])
        theta_yi, dyi = point_to_point_traj(Y[i], Y[i+1], Vy[i], Vy[i+1], T[i+1] - T[i])

        theta_x += theta_xi.tolist()
        theta_y += theta_yi.tolist()
        dx += dxi.tolist()
        dy += dyi.tolist()

        plot(theta_xi, theta_yi)
    return theta_x, theta_y, dx, dy
```