Project: Part 4

24-677 Special Topics: Modern Control - Theory and Design

Ryan Wu (ID: weihuanw)

Due: Nov 28, 2023, 11:59 pm

Exercise 1

Given: $X_{t+1} = X_t + \delta t(\dot{x}_t \cos\psi_t - \dot{y}_t \sin\psi_t) + w_t^x$ ($\dot{x}$: $\dot{X}$)

$Y_{t+1} = Y_t + \delta t(\dot{x}_t \sin\psi_t + \dot{y}_t \cos\psi_t) + w_t^y$ ($\dot{y}$: $\dot{Y}$)

$\psi_{t+1} = \psi_t + \delta t \dot{\psi}_t + w_t^\psi$

$$x_t = \begin{bmatrix} X_t \\ Y_t \\ \psi_t \\ m_x^1 \\ m_y^1 \\ m_x^2 \\ m_y^2 \\ \vdots \\ m_x^n \\ m_y^n \end{bmatrix} \quad \begin{matrix} \\ \\ \text{robot's pose} \\ \\ \\ \\ \\ \\ \\ \text{landmark} \end{matrix}$$

$$y_t = \begin{bmatrix} \| m^1 - p_t \| \\ \vdots \\ \| m^n - p_t \| \\ \text{atan2}(m_y^1 - Y_t, m_x^1 - X_t) - \psi_t \\ \vdots \\ \text{atan2}(m_y^n - Y_t, m_x^n - X_t) - \psi_t \end{bmatrix} + \begin{bmatrix} v_t^1, \text{distance} \\ \vdots \\ v_t^n, \text{distance} \\ v_t^1, \text{bearing} \\ \vdots \\ v_t^n, \text{bearing} \end{bmatrix}$$

Find: Derive F_t and H_t for EKF SLAM to estimate the vehicle states X, Y, ψ and feature positions $m^j = [m_x^j \; m_y^j]^T$ simultaneously.

Solution: $F_t = \frac{\partial f}{\partial x}\Big|_{\hat{x}_{t-1|t-1}, u_t}$, $H_t = \frac{\partial h}{\partial x}\Big|_{\hat{x}_{t|t-1}}$

$X_t = f(x_{t-1}, u_t) + w_t^x \rightarrow f(x_{t-1}, u_t) = X_{t-1} + \delta_{t-1}(\dot{x}_t \cos\psi_{t-1} - \dot{y}_t \sin\psi_{t-1})$

$Y_t = f(y_{t-1}, u_t) + w_t^y \rightarrow f(y_{t-1}, u_t) = Y_{t-1} + \delta_{t-1}(\dot{x}_t \sin\psi_{t-1} + \dot{y}_t \cos\psi_{t-1})$

$\psi_t = f(\psi_{t-1}, u_t) + w_t^\psi \rightarrow f(\psi_{t-1}, u_t) = \psi_{t-1} + \delta_{t-1}\dot{\psi}_{t-1}$

$\rightarrow F_t = \frac{\partial f}{\partial x}\Big|_{\hat{x}_{t-1|t-1}, u_t}$

$$F_t = \begin{bmatrix} 1 & 0 & \delta_{t-1}(-\dot{x}_t \sin\psi_{t-1} - \dot{y}_t \cos\psi_{t-1}) & & \\ 0 & 1 & \delta_{t-1}(\dot{x}_t \cos\psi_{t-1} - \dot{y}_t \sin\psi_{t-1}) & 0_{3\times 2n} & \\ 0 & 0 & 1 & & \\ \hline & 0_{2n\times 3} & & I_{2n\times 2n} & \end{bmatrix}$$

F_t

$$y_t = \begin{bmatrix} \| m^1 - p_t \| \\ \vdots \\ \| m^n - p_t \| \\ \operatorname{atan2}(m_y^1 - Y_t,\ m_x^1 - X_t) - \psi_t \\ \vdots \\ \operatorname{atan2}(m_y^n - Y_t,\ m_x^n - X_t) - \psi_t \end{bmatrix} + \begin{bmatrix} v_t^1, \text{distance} \\ \vdots \\ v_t^n, \text{distance} \\ v_t^1, \text{bearing} \\ \vdots \\ v_t^n, \text{bearing} \end{bmatrix}, \quad H_t = \frac{\partial h}{\partial x}\Big|_{\bar{x}_t | t-1}$$

$$\frac{\partial}{\partial x}(|x|) = \frac{x}{|x|}$$

$$\frac{\partial}{\partial x}(\arctan x) = \frac{1}{1+x^2}$$

$$H_t \in \mathbb{R}^{2n \times (2n+3)}$$

$$H_t = \begin{bmatrix} \frac{-(m_x^1 - X_t)}{\sqrt{(m_x^1 - x_t)^2 + (m_y^1 - y_t)^2}} & \frac{-(m_y^1 - Y_t)}{\sqrt{(m_x^1 - x_t)^2 + (m_y^1 - y_t)^2}} & 0 & \frac{m_x^1 - X_t}{\sqrt{(m_x^1 - x_t)^2 + (m_y^1 - y_t)^2}} & \frac{m_y^1 - Y_t}{\sqrt{(m_x^1 - x_t)^2 + (m_y^1 - y_t)^2}} & 0 & 0 & 0 & \cdots \\ \frac{-(m_x^2 - X_t)}{\sqrt{(m_x^2 - x_t)^2 + (m_y^2 - y_t)^2}} & \frac{-(m_y^2 - Y_t)}{\sqrt{(m_x^2 - x_t)^2 + (m_y^2 - y_t)^2}} & 0 & 0 & 0 & \frac{m_x^2 - X_t}{\sqrt{(m_x^2 - x_t)^2 + (m_y^2 - y_t)^2}} & \frac{m_y^2 - Y_t}{\sqrt{(m_x^2 - x_t)^2 + (m_y^2 - y_t)^2}} & 0 & 0 & \cdots \\ \vdots & \vdots & 0 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \cdots \\ \frac{-(m_x^n - X_t)}{\sqrt{(m_x^n - x_t)^2 + (m_y^n - y_t)^2}} & \frac{-(m_y^n - Y_t)}{\sqrt{(m_x^n - x_t)^2 + (m_y^n - y_t)^2}} & 0 & \vdots & \vdots & \vdots & \vdots & \cdots & \frac{m_x^n - X_t}{\sqrt{(m_x^n - x_t)^2 + (m_y^n - y_t)^2}} & \frac{m_y^n - Y_t}{\sqrt{(m_x^n - x_t)^2 + (m_y^n - y_t)^2}} \\ \frac{m_y^1 - Y_t}{(m_x^1 - x_t)^2 + (m_y^1 - y_t)^2} & \frac{-(m_x^1 - X_t)}{(m_x^1 - x_t)^2 + (m_y^1 - y_t)^2} & -1 & \frac{-(m_y^1 - Y_t)}{(m_x^1 - x_t)^2 + (m_y^1 - y_t)^2} & \frac{m_x^1 - X_t}{(m_x^1 - x_t)^2 + (m_y^1 - y_t)^2} & 0 & 0 & 0 & 0 & \cdots \\ \frac{m_y^2 - Y_t}{(m_x^2 - x_t)^2 + (m_y^2 - y_t)^2} & \frac{-(m_x^2 - X_t)}{(m_x^2 - x_t)^2 + (m_y^2 - y_t)^2} & -1 & 0 & 0 & \frac{-(m_y^2 - Y_t)}{(m_x^2 - x_t)^2 + (m_y^2 - y_t)^2} & \frac{m_x^2 - X_t}{(m_x^2 - x_t)^2 + (m_y^2 - y_t)^2} & 0 & 0 & \cdots \\ \vdots & \vdots & -1 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \cdots \\ \frac{m_y^n - Y_t}{(m_x^n - x_t)^2 + (m_y^n - y_t)^2} & \frac{-(m_x^n - X_t)}{(m_x^n - x_t)^2 + (m_y^n - y_t)^2} & -1 & \vdots & \vdots & \vdots & \vdots & \cdots & \frac{-(m_y^n - Y_t)}{(m_x^n - x_t)^2 + (m_y^n - y_t)^2} & \frac{m_x^n - X_t}{(m_x^n - x_t)^2 + (m_y^n - y_t)^2} \end{bmatrix}$$

\# H_t

Exercise 2.

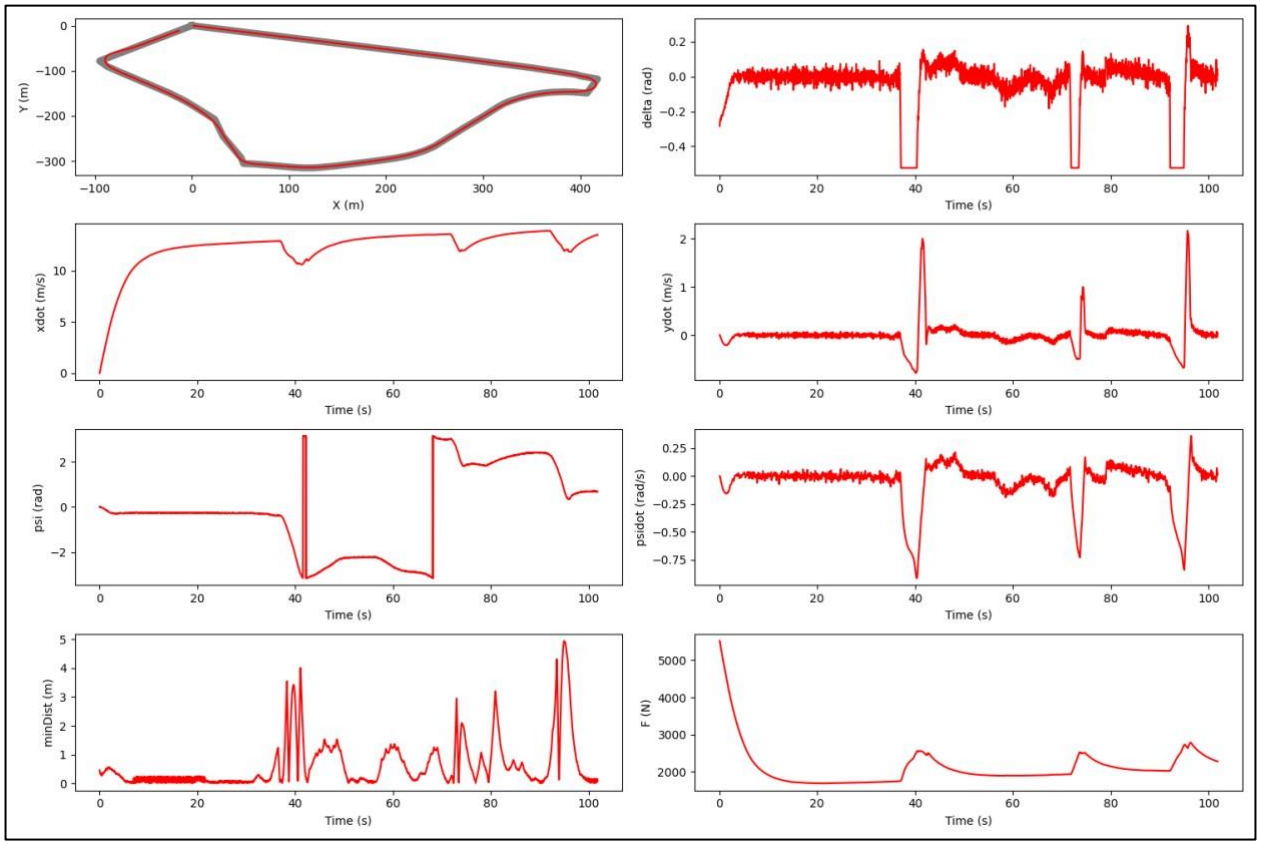

Figure 1. The final completion plots.

```
Evaluating...
Score for completing the loop: 30.0/30.0
Score for average distance: 30.0/30.0
Score for maximum distance: 30.0/30.0
Your time is  101.85600000000001
Your total score is : 100.0/100.0
total steps:  101856
maxMinDist:  4.942361096716563
avgMinDist:  0.636678165232065
INFO: 'main' controller exited successfully.
```

Figure 2. The Webots terminal output for completion time.

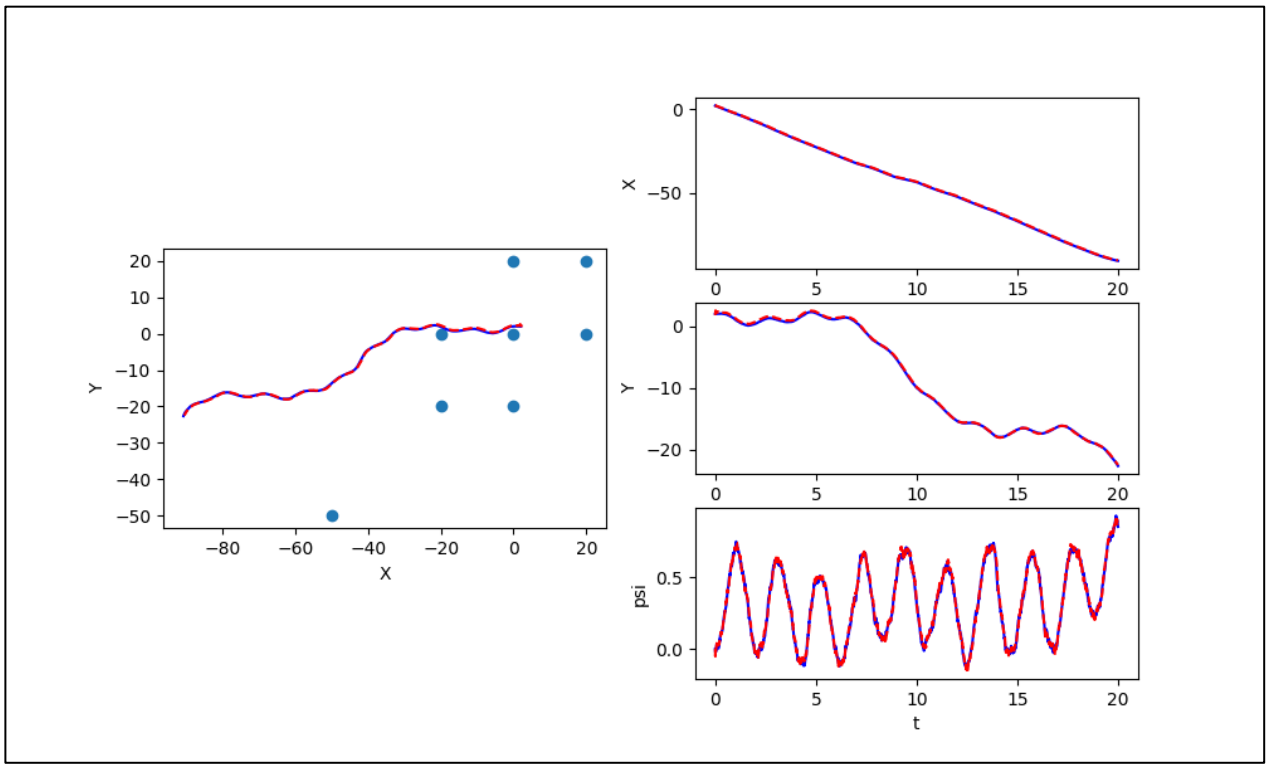

Figure 3. Plots from the efk_slam.py test script.

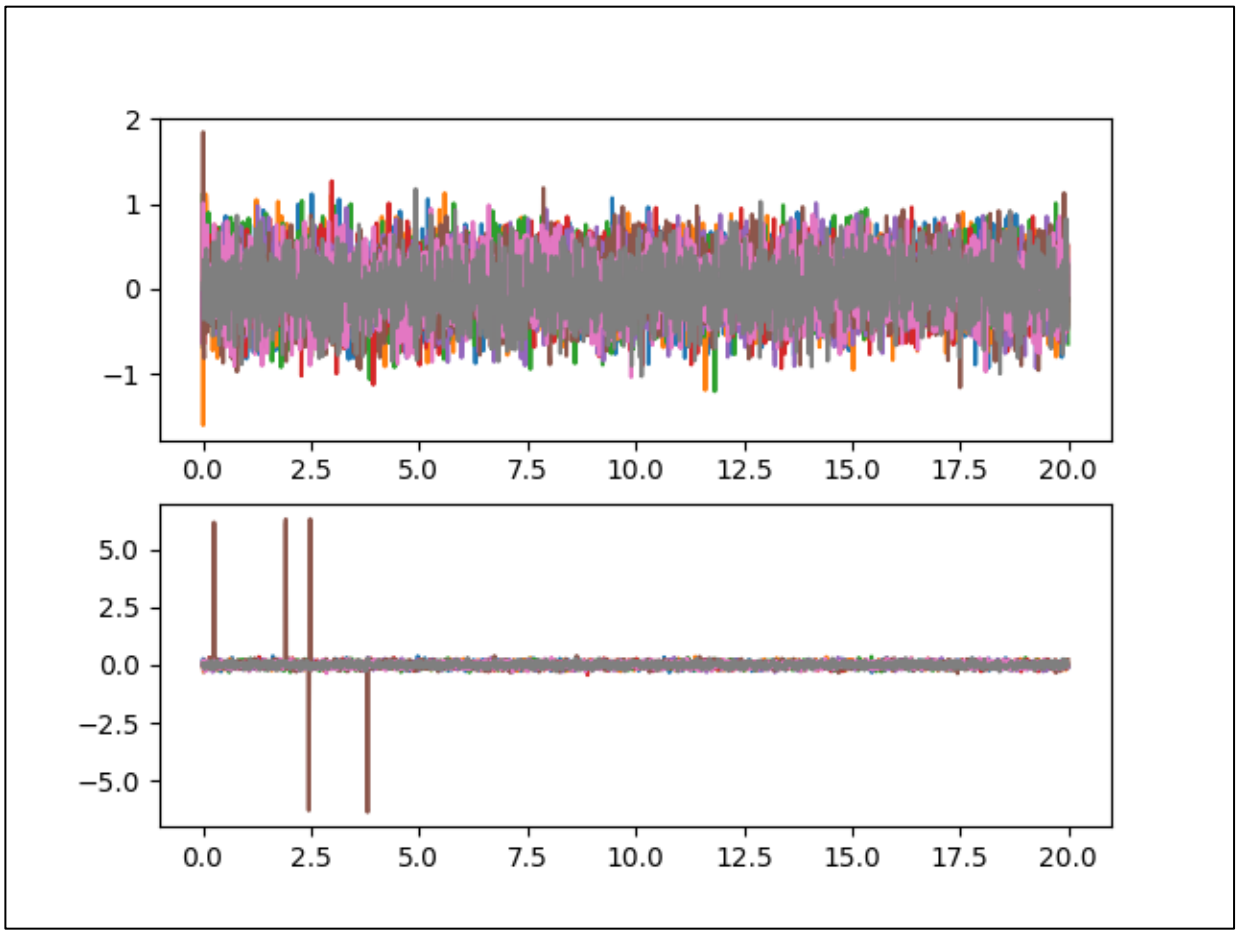

Figure 4. Plots from the efk_slam.py test script.

```python
import numpy as np

class EKF_SLAM():
    def __init__(self, init_mu, init_P, dt, W, V, n):
        """Initialize EKF SLAM

        Create and initialize an EKF SLAM to estimate the robot's pose and
        the location of map features

        Args:
            init_mu: A numpy array of size (3+2*n, ). Initial guess of the mean
            of state.
            init_P: A numpy array of size (3+2*n, 3+2*n). Initial guess of
            the covariance of state.
            dt: A double. The time step.
            W: A numpy array of size (3+2*n, 3+2*n). Process noise
            V: A numpy array of size (2*n, 2*n). Observation noise
            n: A int. Number of map features


        Returns:
            An EKF SLAM object.
        """
        self.mu = init_mu  # initial guess of state mean
        self.P = init_P  # initial guess of state covariance
        self.dt = dt  # time step
        self.W = W  # process noise
        self.V = V  # observation noise
        self.n = n  # number of map features

    # TODO: complete the function below
    def _f(self, x, u):
        """Non-linear dynamic function.
```

```
34
35          Compute the state at next time step
   according to the nonlinear dynamics f.
36
37          Args:
38              x: A numpy array of size (3+2*n, ).
   State at current time step.
39              u: A numpy array of size (3, ). The
   control input [\dot{x}, \dot{y}, \dot{\psi}]
40
41          Returns:
42              x_next: A numpy array of size (3+2*n
   , ). The state at next time step
43          """
44          angle = self._wrap_to_pi(x[2])  # wrap the
   angle between [-pi, pi]
45          x_next = np.zeros(3+2*self.n)
46          x_next[0] = x[0] + self.dt * (u[0] * np.cos
   (angle) - u[1] * np.sin(angle))
47          x_next[1] = x[1] + self.dt * (u[0] * np.sin
   (angle) + u[1] * np.cos(angle))
48          x_next[2] = self._wrap_to_pi(x[2] + self.dt
    * u[2])
49          x_next[3:] = x[3:]
50
51          return x_next
52
53      # TODO: complete the function below
54      def _h(self, x):
55          """Non-linear measurement function.
56
57          Compute the sensor measurement according to
    the nonlinear function h.
58
59          Args:
60              x: A numpy array of size (3+2*n, ).
   State at current time step.
61
62          Returns:
63              y: A numpy array of size (2*n, ). The
   sensor measurement.
```

```
64          """
65          y = np.zeros(2*self.n)
66
67          # inter-landmark measurement
68          for i in range(self.n):
69              # distance
70              y[i] = np.sqrt(pow((x[2 * i + 3] - x[0
    ]), 2) + pow((x[2 * i + 4] - x[1]), 2))
71
72          # inter-landmark measurement
73          for i in range(self.n, 2*self.n):
74              # bearing
75              # y[self.n + 1] = self._wrap_to_pi(np.
    arctan2(x[4 + (i - self.n) * 2] - x[1], x[3 + (i
     - self.n) * 2] - x[0]) - x[2])
76              y[i] = self._wrap_to_pi(np.arctan2(x[4
     + (i - self.n) * 2] - x[1], x[3 + (i - self.n) *
    2] - x[0]) - x[2])
77          return y
78
79      # TODO: complete the function below
80      def _compute_F(self, x, u):
81          """Compute Jacobian of f
82
83          Args:
84              x: A numpy array of size (3+2*n, ).
    The state vector.
85              u: A numpy array of size (3, ). The
    control input [\dot{x}, \dot{y}, \dot{\psi}]
86
87          Returns:
88              F: A numpy array of size (3+2*n, 3+2*n
    ). The Jacobian of f evaluated at x_k.
89          """
90
91          F = np.zeros((3+2*self.n, 3+2*self.n))
92
93          # identity matrix
94          for i in range(3+2*self.n):
95              F[i,i] = 1
96
```

```
 97          # variables
 98          x, y, psi = x[0], x[1], x[2]
 99          u1, u2, u3 = u[0], u[1], u[2]
100
101          # Jacobian calculation for F
102          angle = self._wrap_to_pi(self.mu[2])  # wrap the angle between [-pi, pi]
103          F[0, 2] = -self.dt * (u[0] * np.sin(angle) + u[1] * np.cos(angle))
104          F[1, 2] = self.dt * (u[0] * np.cos(angle) - u[1] * np.sin(angle))
105
106          return F
107
108      # TODO: complete the function below
109      def _compute_H(self, x):
110          """Compute Jacobian of h
111
112          Args:
113              x: A numpy array of size (3+2*n, ). The state vector.
114
115          Returns:
116              H: A numpy array of size (2*n, 3+2*n). The jacobian of h evaluated at x_k.
117          """
118
119          H = np.zeros((2*self.n, 3+2*self.n))
120          X = self.mu[0]
121          Y = self.mu[1]
122
123          for i in range(self.n):
124              x = self.mu[3 + i * 2]
125              y = self.mu[4 + i * 2]
126              # distance
127              H[i, 0] = (X - x) / np.sqrt((X - x)**2 + (Y - y)**2)
128              H[i, 1] = (Y - y) / np.sqrt((X - x)**2 + (Y - y)**2)
129              H[i, 2] = 0
130              H[i, 3 + i * 2] = -(X - x) / np.sqrt((
```

```
130 X - x)**2 + (Y - y)**2)
131                 H[i, 4 + i * 2] = -(Y - y) / np.sqrt((
    X - x)**2 + (Y - y)**2)
132 
133             for i in range(self.n, 2*self.n):
134                 x = self.mu[3+(i-self.n)*2]
135                 y = self.mu[4+(i-self.n)*2]
136                 # bearing
137                 H[i, 0] = -(Y - y) / ((X - x)**2 + (Y
     - y)**2)
138                 H[i, 1] = (X - x) / ((X - x)**2 + (Y
     - y)**2)
139                 H[i, 2] = -1
140                 H[i, 3+(i-self.n)*2] = (Y - y) / ((X
     - x)**2 + (Y - y)**2)
141                 H[i, 4+(i-self.n)*2] = -(X - x) / ((X
     - x)**2 + (Y - y)**2)
142 
143             return H
144 
145 
146     def predict_and_correct(self, y, u):
147         """Predice and correct step of EKF
148 
149         Args:
150             y: A numpy array of size (2*n, ). The
    measurements according to the project description.
151             u: A numpy array of size (3, ). The
    control input [\dot{x}, \dot{y}, \dot{\psi}]
152 
153         Returns:
154             self.mu: A numpy array of size (3+2*n
    , ). The corrected state estimation
155             self.P: A numpy array of size (3+2*n,
    3+2*n). The corrected state covariance
156         """
157 
158         # compute F
159         F = self._compute_F(self.mu, u)
160 
161         #****************** Predict step
```

```
161  ******************#
162          # predict the state
163          self.mu = self._f(self.mu, u)
164          self.mu[2] =  self._wrap_to_pi(self.mu[2])
165          # predict the error covariance
166          self.P = F @ self.P @ F.T + self.W
167
168          #****************** Correct step
     ******************#
169          # compute H matrix
170          H = self._compute_H(self.mu)
171
172          # compute the Kalman gain
173          L = self.P @ H.T @ np.linalg.inv(H @ self.
    P @ H.T + self.V)
174
175          # update estimation with new measurement
176          diff = y - self._h(self.mu)
177          diff[self.n:] = self._wrap_to_pi(diff[self
    .n:])
178          self.mu = self.mu + L @ diff
179          self.mu[2] =  self._wrap_to_pi(self.mu[2])
180
181          # update the error covariance
182          self.P = (np.eye(3+2*self.n) - L @ H) @
    self.P
183
184          return self.mu, self.P
185
186
187      def _wrap_to_pi(self, angle):
188          angle = angle - 2*np.pi*np.floor((angle+np
    .pi )/(2*np.pi))
189          return angle
190
191
192  if __name__ == '__main__':
193      import matplotlib.pyplot as plt
194
195      m = np.array([[0.,  0.],
196                    [0.,  20.],
```

```
197                     [20., 0.],
198                     [20., 20.],
199                     [0,  -20],
200                     [-20, 0],
201                     [-20, -20],
202                     [-50, -50]]).reshape(-1)
203 
204     dt = 0.01
205     T = np.arange(0, 20, dt)
206     n = int(len(m)/2)
207     W = np.zeros((3+2*n, 3+2*n))
208     W[0:3, 0:3] = dt**2 * 1 * np.eye(3)
209     V = 0.1*np.eye(2*n)
210     V[n:,n:] = 0.01*np.eye(n)
211 
212     # EKF estimation
213     mu_ekf = np.zeros((3+2*n, len(T)))
214     mu_ekf[0:3,0] = np.array([2.2, 1.8, 0.])
215     # mu_ekf[3:,0] = m + 0.1
216     mu_ekf[3:,0] = m + np.random.
    multivariate_normal(np.zeros(2*n), 0.5*np.eye(2*n
    ))
217     init_P = 1*np.eye(3+2*n)
218 
219     # initialize EKF SLAM
220     slam = EKF_SLAM(mu_ekf[:,0], init_P, dt, W, V
    , n)
221 
222     # real state
223     mu = np.zeros((3+2*n, len(T)))
224     mu[0:3,0] = np.array([2, 2, 0.])
225     mu[3:,0] = m
226 
227     y_hist = np.zeros((2*n, len(T)))
228     for i, t in enumerate(T):
229         if i > 0:
230             # real dynamics
231             u = [-5, 2*np.sin(t*0.5), 1*np.sin(t*3
    )]
232             # u = [0.5, 0.5*np.sin(t*0.5), 0]
233             # u = [0.5, 0.5, 0]
```

```
234             mu[:,i] = slam._f(mu[:,i-1], u) + \
235                 np.random.multivariate_normal(np.
    zeros(3+2*n), W)
236 
237             # measurements
238             y = slam._h(mu[:,i]) + np.random.
    multivariate_normal(np.zeros(2*n), V)
239             y_hist[:,i] = (y-slam._h(slam.mu))
240             # apply EKF SLAM
241             mu_est, _ = slam.predict_and_correct(y
    , u)
242             mu_ekf[:,i] = mu_est
243 
244 
245     plt.figure(1, figsize=(10,6))
246     ax1 = plt.subplot(121, aspect='equal')
247     ax1.plot(mu[0,:], mu[1,:], 'b')
248     ax1.plot(mu_ekf[0,:], mu_ekf[1,:], 'r--')
249     mf = m.reshape((-1,2))
250     ax1.scatter(mf[:,0], mf[:,1])
251     ax1.set_xlabel('X')
252     ax1.set_ylabel('Y')
253 
254     ax2 = plt.subplot(322)
255     ax2.plot(T, mu[0,:], 'b')
256     ax2.plot(T, mu_ekf[0,:], 'r--')
257     ax2.set_xlabel('t')
258     ax2.set_ylabel('X')
259 
260     ax3 = plt.subplot(324)
261     ax3.plot(T, mu[1,:], 'b')
262     ax3.plot(T, mu_ekf[1,:], 'r--')
263     ax3.set_xlabel('t')
264     ax3.set_ylabel('Y')
265 
266     ax4 = plt.subplot(326)
267     ax4.plot(T, mu[2,:], 'b')
268     ax4.plot(T, mu_ekf[2,:], 'r--')
269     ax4.set_xlabel('t')
270     ax4.set_ylabel('psi')
271 
```

```
272     plt.figure(2)
273     ax1 = plt.subplot(211)
274     ax1.plot(T, y_hist[0:n, :].T)
275     ax2 = plt.subplot(212)
276     ax2.plot(T, y_hist[n:, :].T)
277
278     plt.show()
279
```

```python
# Fill in the respective function to implement the LQR/EKF SLAM controller

# Import libraries
import numpy as np
from base_controller import BaseController
from scipy import signal, linalg
from scipy.spatial.transform import Rotation
from util import *
from ekf_slam import EKF_SLAM

# CustomController class (inherits from BaseController)
class CustomController(BaseController):

    def __init__(self, trajectory, look_ahead_distance=190):

        super().__init__(trajectory)

        # Define constants
        # These can be ignored in P1
        self.lr = 1.39
        self.lf = 1.55
        self.Ca = 20000
        self.Iz = 25854
        self.m = 1888.6
        self.g = 9.81

        self.counter = 0
        np.random.seed(99)

        # Add additional member variables according to your need here.
        self.look_ahead_distance = look_ahead_distance
        self.previous_psi = 0
        self.velocity_start = 58
        self.velocity_integral_error = 0
        self.velocity_previous_step_error = 0

```

```
37      def getStates(self, timestep, use_slam=False):
38
39          delT, X, Y, xdot, ydot, psi, psidot = super
    ().getStates(timestep)
40
41          # Initialize the EKF SLAM estimation
42          if self.counter == 0:
43              # Load the map
44              minX, maxX, minY, maxY = -120., 450., -
    500., 50.
45              map_x = np.linspace(minX, maxX, 7)
46              map_y = np.linspace(minY, maxY, 7)
47              map_X, map_Y = np.meshgrid(map_x, map_y
    )
48              map_X = map_X.reshape(-1,1)
49              map_Y = map_Y.reshape(-1,1)
50              self.map = np.hstack((map_X, map_Y)).
    reshape((-1))
51
52              # Parameters for EKF SLAM
53              self.n = int(len(self.map)/2
    )
54              X_est = X + 0.5
55              Y_est = Y - 0.5
56              psi_est = psi - 0.02
57              mu_est = np.zeros(3+2*self.n)
58              mu_est[0:3] = np.array([X_est, Y_est,
    psi_est])
59              mu_est[3:] = np.array(self.map)
60              init_P = 1*np.eye(3+2*self.n)
61              W = np.zeros((3+2*self.n, 3+2*self.n))
62              W[0:3, 0:3] = delT**2 * 0.1 * np.eye(3)
63              V = 0.1*np.eye(2*self.n)
64              V[self.n:, self.n:] = 0.01*np.eye(self.
    n)
65              # V[self.n:] = 0.01
66              print(V)
67
68              # Create a SLAM
69              self.slam = EKF_SLAM(mu_est, init_P,
    delT, W, V, self.n)
```

```
 70             self.counter += 1
 71         else:
 72             mu = np.zeros(3+2*self.n)
 73             mu[0:3] = np.array([X,
 74                                 Y,
 75                                 psi])
 76             mu[3:] = self.map
 77             y = self._compute_measurements(X, Y,
    psi)
 78             mu_est, _ = self.slam.
    predict_and_correct(y, self.previous_u)
 79
 80         self.previous_u = np.array([xdot, ydot,
    psidot])
 81
 82         print("True      X, Y, psi:", X, Y, psi)
 83         print("Estimated X, Y, psi:", mu_est[0],
    mu_est[1], mu_est[2])
 84         print(
    "-----------------------------------------------
    ------")
 85
 86         if use_slam == True:
 87             return delT, mu_est[0], mu_est[1],
    xdot, ydot, mu_est[2], psidot
 88         else:
 89             return delT, X, Y, xdot, ydot, psi,
    psidot
 90
 91     def _compute_measurements(self, X, Y, psi):
 92         x = np.zeros(3+2*self.n)
 93         x[0:3] = np.array([X, Y, psi])
 94         x[3:] = self.map
 95
 96         p = x[0:2]
 97         psi = x[2]
 98         m = x[3:].reshape((-1,2))
 99
100         y = np.zeros(2*self.n)
101
102         for i in range(self.n):
```

```
103                 y[i] = np.linalg.norm(m[i, :] - p)
104                 y[self.n+i] = wrapToPi(np.arctan2(m[i,
    1]-p[1], m[i,0]-p[0]) - psi)
105 
106         y = y + np.random.multivariate_normal(np.
    zeros(2*self.n), self.slam.V)
107         # print(np.random.multivariate_normal(np.
    zeros(2*self.n), self.slam.V))
108         return y
109 
110     def update(self, timestep):
111 
112         trajectory = self.trajectory
113 
114         lr = self.lr
115         lf = self.lf
116         Ca = self.Ca
117         Iz = self.Iz
118         m = self.m
119         g = self.g
120 
121         # Fetch the states from the newly defined
    getStates method
122         delT, X, Y, xdot, ydot, psi, psidot = self
    .getStates(timestep, use_slam=True)
123         # You must not use true_X, true_Y and
    true_psi since they are for plotting purpose
124         # _, true_X, true_Y, _, _, true_psi, _ =
    self.getStates(timestep, use_slam=False)
125 
126         # You are free to reuse or refine your
    code from P3 in the spaces below.
127         # Design your controllers in the spaces
    below.
128         # Remember, your controllers will need to
    use the states
129         # to calculate control inputs (F, delta).
130 
131         # ---------------|Lateral Controller
    |-------------------------
132 
```

```
133         # Please design your lateral controller
    below.
134
135         # state space model for lateral control
136         A = np.array(
137             [[0, 1, 0, 0], [0, -4 * Ca / (m * xdot
    ), 4 * Ca / m, (-2 * Ca * (lf - lr)) / (m * xdot
    )], [0, 0, 0, 1],
138              [0, (-2 * Ca * (lf - lr)) / (Iz *
    xdot), (2 * Ca * (lf - lr)) / Iz,
139               (-2 * Ca * (lf ** 2 + lr ** 2)) / (
    Iz * xdot)]])
140         B = np.array([[0], [2 * Ca / m], [0], [2
     * Ca * lf / Iz]])
141         C = np.eye(4)
142         D = np.zeros((4, 1))
143
144         # discretize the state space model
145         sys_continuous = signal.StateSpace(A, B, C
    , D)
146         sys_discretize = sys_continuous.
    to_discrete(delT)
147         A_discretize = sys_discretize.A
148         B_discretize = sys_discretize.B
149
150         #  Set the look-ahead distance and find
    the closest index to the current position
151         look_ahead_distance = 190
152         _, closest_index = closestNode(X, Y,
    trajectory)
153
154         # stop look-ahead distance from going out
    of bounds
155         max_allowed_look_ahead = min(
    look_ahead_distance, len(trajectory) -
    closest_index - 1)
156         look_ahead_distance = max(0,
    max_allowed_look_ahead)
157
158         # calculate the desired heading angle (
    psi_desired) (referencing Project 2 solution)
```

```
159         psi_desired = np.arctan2(trajectory[
    closest_index + look_ahead_distance, 1] -
    trajectory[closest_index, 1],
160                                  trajectory[
    closest_index + look_ahead_distance, 0] -
    trajectory[closest_index, 0])
161
162         # error calculation (referencing Project 2
     solution)
163         e1 = (Y - trajectory[closest_index +
    look_ahead_distance, 1]) * np.cos(psi_desired) - (
164                 X - trajectory[closest_index +
    look_ahead_distance, 0]) * np.sin(psi_desired)
165         e2 = wrapToPi(psi - psi_desired)
166         e1_dot = ydot + xdot * e2
167         e2_dot = psidot
168
169         # LQR controller design
170         Q = np.eye(4)
171         R = 40
172
173         # solve for P and gain matrix K
174         P = linalg.solve_discrete_are(A_discretize
    , B_discretize, Q, R)
175         K = linalg.inv(R + B_discretize.T @ P @
    B_discretize) @ (B_discretize.T @ P @ A_discretize
    )
176
177         # control delta calculation
178         delta = (-K @ np.array([[e1], [e1_dot], [
    e2], [e2_dot]]))[0, 0]
179         delta = np.clip(delta, -np.pi / 6, np.pi
     / 6)
180
181         # ---------------|Longitudinal Controller
    |-------------------------
182
183         # Please design your longitudinal
    controller below.
184
185         # declaring PID variables
```

```
186         Kp_velocity = 95
187         Ki_velocity = 1
188         Kd_velocity = 0.005
189 
190         # velocity error calculation
191         velocity = np.sqrt(xdot ** 2 + ydot ** 2
    ) * 3.6
192         velocity_error = self.velocity_start -
    velocity
193         self.velocity_integral_error +=
    velocity_error * delT
194         velocity_derivative_error = (
    velocity_error - self.velocity_previous_step_error
    ) / delT
195 
196         # F with PID feedback control
197         F = (velocity_error * Kp_velocity) + (self
    .velocity_integral_error * Ki_velocity) + (
198                 velocity_derivative_error *
    Kd_velocity)
199 
200         # Return all states and calculated control
     inputs (F, delta)
201         return X, Y, xdot, ydot, psi, psidot, F,
    delta
202 
```