

```

1 import numpy as np
2
3 class EKF_SLAM():
4     def __init__(self, init_mu, init_P, dt, W, V, n
5 ):
6         """Initialize EKF SLAM
7
8         Create and initialize an EKF SLAM to
9         estimate the robot's pose and
10        the location of map features
11
12        Args:
13            init_mu: A numpy array of size (3+2*n
14            , ). Initial guess of the mean
15            of state.
16            init_P: A numpy array of size (3+2*n, 3
17            +2*n). Initial guess of
18            the covariance of state.
19            dt: A double. The time step.
20            W: A numpy array of size (3+2*n, 3+2*n
21            ). Process noise
22            V: A numpy array of size (2*n, 2*n).
23            Observation noise
24            n: A int. Number of map features
25
26        Returns:
27            An EKF SLAM object.
28        """
29        self.mu = init_mu # initial guess of state
30        mean
31        self.P = init_P # initial guess of state
32        covariance
33        self.dt = dt # time step
34        self.W = W # process noise
35        self.V = V # observation noise
36        self.n = n # number of map features
37
38        # TODO: complete the function below
39        def _f(self, x, u):
40            """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
71 ]), 2) + pow((x[2 * i + 4] - x[1]), 2))
72
73         # inter-landmark measurement
74         for i in range(self.n, 2*self.n):
75             # bearing
76             # y[self.n + 1] = self._wrap_to_pi(np.
77             arctan2(x[4 + (i - self.n) * 2] - x[1], x[3 + (i
78 - self.n) * 2] - x[0]) - x[2])
79             y[i] = self._wrap_to_pi(np.arctan2(x[4
80 + (i - self.n) * 2] - x[1], x[3 + (i - self.n) *
81 2] - x[0]) - x[2])
82         return y
83
84         # TODO: complete the function below
85         def _compute_F(self, x, u):
86             """Compute Jacobian of f
87
88             Args:
89                 x: A numpy array of size (3+2*n, ).
90                 The state vector.
91                 u: A numpy array of size (3, ). The
92                 control input [\dot{x}, \dot{y}, \dot{\psi}]
93
94             Returns:
95                 F: A numpy array of size (3+2*n, 3+2*n
96                 ). The Jacobian of f evaluated at x_k.
97             """
98
99             F = np.zeros((3+2*self.n, 3+2*self.n))
100
101             # identity matrix
102             for i in range(3+2*self.n):
103                 F[i,i] = 1

```

```

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         *****#
170         # compute H matrix
171         H = self._compute_H(self.mu)
172
173         # compute the Kalman gain
174         L = self.P @ H.T @ np.linalg.inv(H @ self.
175         P @ H.T + self.V)
176
177         # update estimation with new measurement
178         diff = y - self._h(self.mu)
179         diff[self.n:] = self._wrap_to_pi(diff[self
180         .n:])
181         self.mu = self.mu + L @ diff
182         self.mu[2] = self._wrap_to_pi(self.mu[2])
183
184         # update the error covariance
185         self.P = (np.eye(3+2*self.n) - L @ H) @
186         self.P
187
188         return self.mu, self.P
189
190
191     def _wrap_to_pi(self, angle):
192         angle = angle - 2*np.pi*np.floor((angle+np
193         .pi)/(2*np.pi))
194         return angle
195
196
197 if __name__ == '__main__':
198     import matplotlib.pyplot as plt
199
200     m = np.array([[0., 0.],
201                   [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
```