```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.
```

```

        Compute the state at next time step according to the nonlinear dynamics f.

        Args:
            x: A numpy array of size (3+2*n, ). State at current time step.
            u: A numpy array of size (3, ). The control input [\dot{x}, \dot{y}, \dot{\psi}]

        Returns:
            x_next: A numpy array of size (3+2*n, ). The state at next time step
        """
        angle = self._wrap_to_pi(x[2])  # wrap the angle between [-pi, pi]
        x_next = np.zeros(3+2*self.n)
        x_next[0] = x[0] + self.dt * (u[0] * np.cos(angle) - u[1] * np.sin(angle))
        x_next[1] = x[1] + self.dt * (u[0] * np.sin(angle) + u[1] * np.cos(angle))
        x_next[2] = self._wrap_to_pi(x[2] + self.dt * u[2])
        x_next[3:] = x[3:]

        return x_next

    # TODO: complete the function below
    def _h(self, x):
        """Non-linear measurement function.

        Compute the sensor measurement according to the nonlinear function h.

        Args:
            x: A numpy array of size (3+2*n, ). State at current time step.

        Returns:
            y: A numpy array of size (2*n, ). The sensor measurement.
```

```
        """
        y = np.zeros(2*self.n)

        # inter-landmark measurement
        for i in range(self.n):
            # distance
            y[i] = np.sqrt(pow((x[2 * i + 3] - x[0]), 2) + pow((x[2 * i + 4] - x[1]), 2))

        # inter-landmark measurement
        for i in range(self.n, 2*self.n):
            # bearing
            # 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])
            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])
        return y

    # TODO: complete the function below
    def _compute_F(self, x, u):
        """Compute Jacobian of f

        Args:
            x: A numpy array of size (3+2*n, ). The state vector.
            u: A numpy array of size (3, ). The control input [\dot{x}, \dot{y}, \dot{\psi}]

        Returns:
            F: A numpy array of size (3+2*n, 3+2*n). The Jacobian of f evaluated at x_k.
        """

        F = np.zeros((3+2*self.n, 3+2*self.n))

        # identity matrix
        for i in range(3+2*self.n):
            F[i,i] = 1

```

```
        # variables
        x, y, psi = x[0], x[1], x[2]
        u1, u2, u3 = u[0], u[1], u[2]

        # Jacobian calculation for F
        angle = self._wrap_to_pi(self.mu[2])  # wrap the angle between [-pi, pi]
        F[0, 2] = -self.dt * (u[0] * np.sin(angle) + u[1] * np.cos(angle))
        F[1, 2] = self.dt * (u[0] * np.cos(angle) - u[1] * np.sin(angle))

        return F

    # TODO: complete the function below
    def _compute_H(self, x):
        """Compute Jacobian of h

        Args:
            x: A numpy array of size (3+2*n, ). The state vector.

        Returns:
            H: A numpy array of size (2*n, 3+2*n). The jacobian of h evaluated at x_k.
        """

        H = np.zeros((2*self.n, 3+2*self.n))
        X = self.mu[0]
        Y = self.mu[1]

        for i in range(self.n):
            x = self.mu[3 + i * 2]
            y = self.mu[4 + i * 2]
            # distance
            H[i, 0] = (X - x) / np.sqrt((X - x)**2 + (Y - y)**2)
            H[i, 1] = (Y - y) / np.sqrt((X - x)**2 + (Y - y)**2)
            H[i, 2] = 0
            H[i, 3 + i * 2] = -(X - x) / np.sqrt((
```

```python
X - x)**2 + (Y - y)**2)
            H[i, 4 + i * 2] = -(Y - y) / np.sqrt((X - x)**2 + (Y - y)**2)

        for i in range(self.n, 2*self.n):
            x = self.mu[3+(i-self.n)*2]
            y = self.mu[4+(i-self.n)*2]
            # bearing
            H[i, 0] = -(Y - y) / ((X - x)**2 + (Y - y)**2)
            H[i, 1] = (X - x) / ((X - x)**2 + (Y - y)**2)
            H[i, 2] = -1
            H[i, 3+(i-self.n)*2] = (Y - y) / ((X - x)**2 + (Y - y)**2)
            H[i, 4+(i-self.n)*2] = -(X - x) / ((X - x)**2 + (Y - y)**2)

        return H


    def predict_and_correct(self, y, u):
        """Predice and correct step of EKF

        Args:
            y: A numpy array of size (2*n, ). The measurements according to the project description.
            u: A numpy array of size (3, ). The control input [\dot{x}, \dot{y}, \dot{\psi}]

        Returns:
            self.mu: A numpy array of size (3+2*n, ). The corrected state estimation
            self.P: A numpy array of size (3+2*n, 3+2*n). The corrected state covariance
        """

        # compute F
        F = self._compute_F(self.mu, u)

        #***************** Predict step
```

```
*****************#
        # predict the state
        self.mu = self._f(self.mu, u)
        self.mu[2] =  self._wrap_to_pi(self.mu[2])
        # predict the error covariance
        self.P = F @ self.P @ F.T + self.W

        #***************** Correct step *****************#
        # compute H matrix
        H = self._compute_H(self.mu)

        # compute the Kalman gain
        L = self.P @ H.T @ np.linalg.inv(H @ self.P @ H.T + self.V)

        # update estimation with new measurement
        diff = y - self._h(self.mu)
        diff[self.n:] = self._wrap_to_pi(diff[self.n:])
        self.mu = self.mu + L @ diff
        self.mu[2] =  self._wrap_to_pi(self.mu[2])

        # update the error covariance
        self.P = (np.eye(3+2*self.n) - L @ H) @ self.P

        return self.mu, self.P


    def _wrap_to_pi(self, angle):
        angle = angle - 2*np.pi*np.floor((angle+np.pi )/(2*np.pi))
        return angle


if __name__ == '__main__':
    import matplotlib.pyplot as plt

    m = np.array([[0.,  0.],
                  [0.,  20.],
```

```
                  [20., 0.],
                  [20., 20.],
                  [0,  -20],
                  [-20, 0],
                  [-20, -20],
                  [-50, -50]]).reshape(-1)

    dt = 0.01
    T = np.arange(0, 20, dt)
    n = int(len(m)/2)
    W = np.zeros((3+2*n, 3+2*n))
    W[0:3, 0:3] = dt**2 * 1 * np.eye(3)
    V = 0.1*np.eye(2*n)
    V[n:,n:] = 0.01*np.eye(n)

    # EKF estimation
    mu_ekf = np.zeros((3+2*n, len(T)))
    mu_ekf[0:3,0] = np.array([2.2, 1.8, 0.])
    # mu_ekf[3:,0] = m + 0.1
    mu_ekf[3:,0] = m + np.random.multivariate_normal(np.zeros(2*n), 0.5*np.eye(2*n))
    init_P = 1*np.eye(3+2*n)

    # initialize EKF SLAM
    slam = EKF_SLAM(mu_ekf[:,0], init_P, dt, W, V, n)

    # real state
    mu = np.zeros((3+2*n, len(T)))
    mu[0:3,0] = np.array([2, 2, 0.])
    mu[3:,0] = m

    y_hist = np.zeros((2*n, len(T)))
    for i, t in enumerate(T):
        if i > 0:
            # real dynamics
            u = [-5, 2*np.sin(t*0.5), 1*np.sin(t*3)]
            # u = [0.5, 0.5*np.sin(t*0.5), 0]
            # u = [0.5, 0.5, 0]
```

```
            mu[:,i] = slam._f(mu[:,i-1], u) + \
                np.random.multivariate_normal(np.zeros(3+2*n), W)

            # measurements
            y = slam._h(mu[:,i]) + np.random.multivariate_normal(np.zeros(2*n), V)
            y_hist[:,i] = (y-slam._h(slam.mu))
            # apply EKF SLAM
            mu_est, _ = slam.predict_and_correct(y, u)
            mu_ekf[:,i] = mu_est


    plt.figure(1, figsize=(10,6))
    ax1 = plt.subplot(121, aspect='equal')
    ax1.plot(mu[0,:], mu[1,:], 'b')
    ax1.plot(mu_ekf[0,:], mu_ekf[1,:], 'r--')
    mf = m.reshape((-1,2))
    ax1.scatter(mf[:,0], mf[:,1])
    ax1.set_xlabel('X')
    ax1.set_ylabel('Y')

    ax2 = plt.subplot(322)
    ax2.plot(T, mu[0,:], 'b')
    ax2.plot(T, mu_ekf[0,:], 'r--')
    ax2.set_xlabel('t')
    ax2.set_ylabel('X')

    ax3 = plt.subplot(324)
    ax3.plot(T, mu[1,:], 'b')
    ax3.plot(T, mu_ekf[1,:], 'r--')
    ax3.set_xlabel('t')
    ax3.set_ylabel('Y')

    ax4 = plt.subplot(326)
    ax4.plot(T, mu[2,:], 'b')
    ax4.plot(T, mu_ekf[2,:], 'r--')
    ax4.set_xlabel('t')
    ax4.set_ylabel('psi')

```

```
    plt.figure(2)
    ax1 = plt.subplot(211)
    ax1.plot(T, y_hist[0:n, :].T)
    ax2 = plt.subplot(212)
    ax2.plot(T, y_hist[n:, :].T)

    plt.show()

```