```
import numpy as np
import matplotlib.pyplot as plt
from plotcov2 import plotcov2

# EN530.603 Extended Kalman filtering of the unicycle with bearing and range
# measurements
# M. Kobilarov , marin(at)jhu.edu

np.random.seed(0)


def fangle(a):
    # make sure angle is between -pi and pi
    a = np.mod(a, 2 * np.pi)
    if a < -np.pi:
        a = a + 2 * np.pi
    else:
        if a > np.pi:
            a = a - 2 * np.pi
    return a


class Problem:

    def __init__(self):

        np.random.seed(10212)

        self.f = self.uni_f   # mobile-robot dynamics
        self.h = self.br_h  # bearing-reange sensing
        self.n = 4  # state dimension
        self.r = 2  # measurement dimension

        self.p0 = np.array([0, 2])    # beacon positions

        # timing
        dt = .1
        self.N = 50
        self.T = dt * self.N
        self.dt = dt

        # noise models
        self.Q = .1 * dt * dt * np.diag([.1, .1, .1, .001])
        self.R = .01 * np.diag([0.5, 1.0])

    def br_h(self, x):

        p = x[:2]
        px = p[0]
        py = p[1]

        d = self.p0 - p
        r = np.linalg.norm(d)

        th = fangle(np.arctan2(d[1], d[0]) - x[2])

        y = np.array([th, r])
        H = np.array([[d[1] / r**2, -d[0] / r**2, -1, 0],
                      [-d[0] / r, -d[1] / r, 0, 0]])

        return y, H

    def fix_state(self, x):
        x[2] = fangle(x[2])
        return x

    def uni_f(self, x, u):
        # dynamical model of the unicycle
        c = np.cos(x[2])
        s = np.sin(x[2])
```

```
        x = x + np.array([c * u[0] * x[3],
                          s * u[0] * x[3],
                          u[1],
                          0])
        x = self.fix_state(x)
        A = np.array([[1, 0, -s * u[0] * x[3], c * u[0]],
                      [0, 1, c * u[0] * x[3], s * u[0]],
                      [0, 0, 1, 0],
                      [0, 0, 0, 1]])

        return x, A


def ekf_predict(x, P, u, prob):

    x, F = prob.f(x, u)
    x = prob.fix_state(x)
    P = F @ P @ np.transpose(F) + prob.Q

    return x, P


def ekf_correct(x, P, z, prob):

    y, H = prob.h(x)
    K = P @ np.transpose(H) @ np.linalg.inv(H @ P @ np.transpose(H) + prob.R)
    P = (np.eye(prob.n) - K @ H) @ P

    e = z - y
    e[0] = fangle(e[0])
    x = x + K @ e
    return x, P


prob = Problem()

# initial mean and covariance
xt = np.array([0, 0, 0, 1])  # true state

P = .1 * np.diag([.1, .1, .1, .4])  # covariance
# initial estimate with added noise
x = xt + np.sqrt(P) @ np.random.randn(prob.n)

xts = np.zeros((prob.n, prob.N + 1))  # true states
xs = np.zeros((prob.n, prob.N + 1))  # estimated states
Ps = np.zeros((prob.n, prob.n, prob.N + 1))  # estimated covariances
ts = np.zeros((prob.N + 1, 1))  # times

zs = np.zeros((prob.r, prob.N))  # measurements

xts[:, 0] = xt
xs[:, 0] = x
Ps[:, :, 0] = P

ds = np.zeros((prob.n, prob.N + 1))  # errors
ds[:, 0] = x - xt

for k in range(prob.N):
    u = prob.dt * np.array([2, 1])  # known controls

    # true state
    x, _ = prob.f(xts[:, k], u)
    xts[:, k + 1] = x + np.sqrt(prob.Q) @ np.random.randn(4)

    x, P = ekf_predict(x, P, u, prob)  # predict

    # generate measurement
    y, H = prob.h(xts[:, k + 1])
    z = y + np.sqrt(prob.R) @ np.random.randn(prob.r)
```

```
    z[0] = fangle(z[0])
    [x, P] = ekf_correct(x, P, z, prob)  # correct

    xs[:, k + 1] = x
    Ps[:, :, k + 1] = P

    zs[:, k] = z
    ds[:, k + 1] = x - xts[:, k + 1]  # actual estimate error
    ds[2, k + 1] = fangle(ds[2, k + 1])

fig1, ax1 = plt.subplots(1, 1)
fig2, ax2 = plt.subplots(1, 1)

ax1.plot(xts[0, :], xts[1, :], '--g', linewidth=3)
ax1.plot(xs[0, :], xs[1, :], '-b', linewidth=3)
ax1.legend({'true', 'estimated'})

# beacon
ax1.plot(prob.p0[0], prob.p0[1], '*r')

ax1.set_xlabel('x')
ax1.set_ylabel('y')
ax1.set_aspect('equal')

ax2.plot(ds[0, :], label='e_x')
ax2.plot(ds[1, :], label='e_y')
ax2.plot(ds[2, :], label='e_theta')
ax2.plot(ds[3, :], label='e_r')

ax2.set_xlabel('k')
ax2.set_ylabel('meters or radians')
handles, labels = ax2.get_legend_handles_labels()
labels, handles = zip(*sorted(zip(labels, handles), key=lambda t: t[0]))
ax2.legend(handles, labels)

plt.show()
```