```
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


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.bearing_only = 1

        # single beacon at (-2,2) : system is unobservable
        self.pbs = np.array([[0], [2]])    # beacon positions

        # two beacons at (-2,2) and (2,2) : system is observable (two or more)
        # self.pbs = np.array([[-2, 2], [2, 2]])    # beacon positions

        self.nb = self.pbs.shape[1]  # number of beacons

        if self.bearing_only:
            self.h = self.b_h         # bearing sensing
            self.r = self.nb          # measurement dimension
            # self.R = .4 * np.diag(np.ones(self.nb) * 0.1)
            self.R = .01 * np.diag(np.ones(self.nb) * 0.5)
        else:
            self.h = self.br_h        # bearing-reange sensing
            self.r = 2 * self.nb      # measurement dimension
            # self.R = .4 * np.diag(np.tile(np.array([.1, .01]), self.nb))
            self.R = .01 * np.diag(np.tile(np.array([0.5, 1.0]), self.nb))

        self.n = 4            # state dimension
        self.f = self.uni_f   # mobile-robot dynamics

        # timing
        dt = .1
        # N = 2580
        self.N = 50
        self.T = dt * self.N
        self.dt = dt

        # noise models
        # self.Q = .3 * dt * np.diag([.1, .1, .01])
        self.Q = .1 * dt * dt * np.diag([.1, .1, .1, .001])
        # self.R = .01 * np.diag([0.5, 1.0])

    def b_h(self, x):

        p = x[:2]
        y = np.zeros(self.nb)
        H = np.zeros((self.nb, 4))

        for i in range(self.nb):
            pb = self.pbs[:, i]  # i-th beacon
```

```
            d = pb - p
            r = np.linalg.norm(d)

            th = fangle(np.arctan2(d[1], d[0]) - x[2])
            y[i] = th
            H[i, :] = np.array([d[1] / r**2, -d[0] / r**2, -1, 0])

        return y, H

    def br_h(self, x):

        p = x[:2]
        y = np.zeros(self.nb * 2)
        H = np.zeros((self.nb * 2, 4))

        for i in range(self.nb):
            pb = self.pbs[:, i]  # i-th beacon
            d = pb - p
            r = np.linalg.norm(d)

            th = fangle(np.arctan2(d[1], d[0]) - x[2])
            y[i * 2:i * 2 + 2] = np.array([th, r])
            H[i * 2:i * 2 + 2, :] = 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 fix_meas(self, z):
        for i in range(prob.nb):
            if prob.bearing_only:
                z[i] = fangle(z[i])
            else:
                z[2 * i] = fangle(z[2 * i])
        return z

    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 = x + np.array([c * u[0], s * u[0], u[1]])
        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]])
        # A = np.array([[1, 0, -s * u[0]], [0, 1, c * u[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)
    P = P - P @ np.transpose(H) @ np.linalg.inv(H @ P @
                                                np.transpose(H) + prob.R) @ H @ P
    K = P @ np.transpose(H) @ np.linalg.inv(prob.R)

    e = z - y
    e = prob.fix_meas(e)
    x = x + K @ e
    return x, P


prob = Problem()

# initial mean and covariance
# xt = np.array([0, 0, np.pi / 4, 0])  # true state
xt = np.array([0, 0, 0, 1])  # true state
P = .1 * np.diag([.1, .1, .1, .4])  # covariance

# P = .2 * np.diag([1, 1, .1])  # 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
ts[0] = 0

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
    ts[k + 1] = k * prob.dt

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

fig1, ax1 = plt.subplots(1, 1)
fig2, ax2 = plt.subplots(1, 2)

for k in range(prob.N):

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

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

```
    plotcov2(xs[0:2, k], Ps[:2, :2, k], ax1)

    ax1.set_xlabel('x')
    ax1.set_ylabel('y')
    ax1.set_aspect('equal')
    ax1.set_xlim([-4, 3])
    ax1.set_ylim([-4, 4])

    # fig1.savefig("./figures/frame%03d" % k + ".jpg")

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

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

ax2[1].plot(ts, np.reshape(np.sqrt(Ps[0, 0, :]), Ps.shape[2]), label="sigma_x")
ax2[1].plot(ts, np.reshape(np.sqrt(Ps[1, 1, :]), Ps.shape[2]), label="sigma_y")
ax2[1].plot(ts, np.reshape(np.sqrt(Ps[2, 2, :]),
            Ps.shape[2]), label="sigma_theta")
ax2[1].plot(ts, np.reshape(np.sqrt(Ps[3, 3, :]),
            Ps.shape[2]), label="sigma_r")

ax2[1].set_xlabel('t')
ax2[1].set_ylabel('meters or radians')
handles, labels = ax2[1].get_legend_handles_labels()
labels, handles = zip(*sorted(zip(labels, handles), key=lambda t: t[0]))
ax2[1].legend(handles, labels)

plt.show()
```