Artificial Intelligence in Control Engineering exercise

Lecturer: Dr.Pham Viet Cuong

October 08th, 2018

Group 9:

Nguyen Chinh Thuy	1513372
Nguyen Tan Phu	1512489
Le Van Hoang Phuong	1512579
Do Tieu Thien	1513172
Nguyen Tan Sy	1512872
Nguyen Van Qui	1512702

1 Problem

2 Configuration

3 Implementation

3.1 Particle Filter

To solve the Paticle Filter problem, we implement follow below steps:

- Prediction

  Process model:

$$x_t = x_{t-1} + V_t \Delta t \cos(\theta_t + \varphi_{t-1}) \tag{1a}$$

$$y_t = y_{t-1} + V_t \Delta t \cos(\theta_t + \varphi_{t-1}) \tag{1b}$$

$$\varphi_t = \varphi_{t-1} + \frac{V_t \Delta t \sin \theta_t}{WB} \tag{1c}$$

  Measurement model:

$$r_t = \sqrt{(x_t - x_L)^2 + (y_t - y_L)^2} \tag{2a}$$

$$b_t = \arctan \frac{y_t - y_L}{x_t - x_L} + \varphi_t \tag{2b}$$

  - Measurement model
  - Implementing a loop with M step which is numbers of particles
  - Using process model and control signals u_t in **VG** which are affected by thermal noise to calculate coordinate $x_t^{[m]}$ of robot.
  - Combining coordinate of robot from above step and coordinate of landmarks in **lm** to calculate range r_t and bearing angle b_t.
  - Calculating importance factor $w_t^{[m]}$ depend on probability density function fomula with μ is matrix of expected range and bearing which are from **Z**.

$$f_x(x_1, x_2, ..., x_N) = \frac{1}{(2\pi)^{\frac{N}{2}} \|\Sigma\|^{\frac{1}{2}}} exp\left(\frac{-1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)\right) \tag{3}$$

- Selection
  - Implementing a loop with M step.
  - Choosing a index in range $[1, M]$ for $x_t^{[m]}$ with probabilities $w_t^{[m]}$.

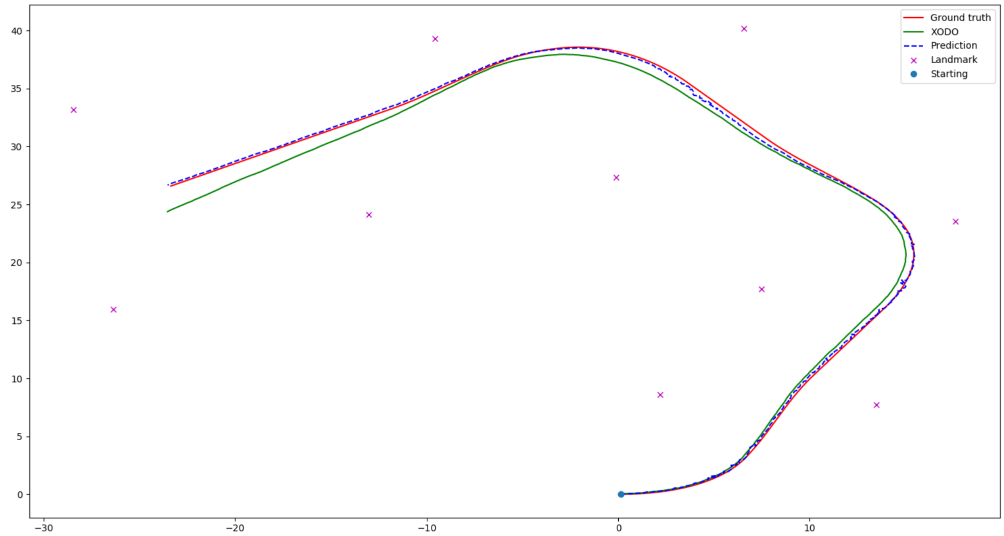
3.1.1 Python code

```python
#------------------------------------------------------------------------------
#    Libraries
#------------------------------------------------------------------------------
import numpy as np
from scipy.io import loadmat
from utils.particle_filter import ParticleFilter

#------------------------------------------------------------------------------
#    Parameters
#------------------------------------------------------------------------------
n_particles = 100
sigma_v, sigma_g = 0.5, 3/180*np.pi
sigma_r, sigma_b = 0.2, 2/180*np.pi
wb = 4
time_step = 0.025
particle = "max"

#------------------------------------------------------------------------------
#    Main execution
#------------------------------------------------------------------------------
# Load data
data = loadmat("data20171107.mat")
landmarks, X_gt, Z, U, X_ODO = data["lm"], data["XTRUE"], data["Z"], data["VG"], data["XODO"]
n_steps = X_gt.shape[1]
```

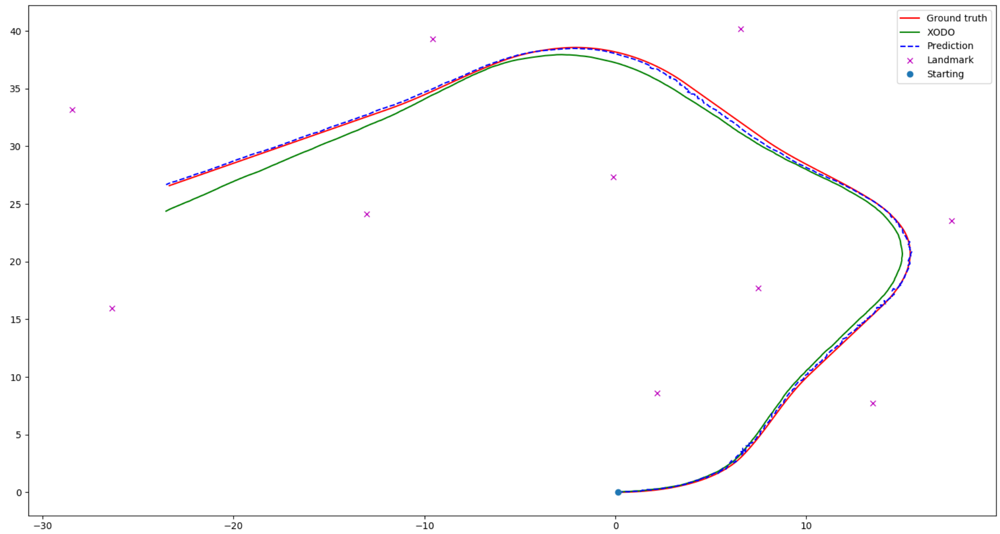
```

# Create a Particle Filter instance
particle_filt = ParticleFilter(
    n_particles=n_particles,
    n_steps=n_steps,
    landmarks=landmarks,
    sigma_v=sigma_v,
    sigma_g=sigma_g,
    sigma_r=sigma_r,
    sigma_b=sigma_b,
    wb=wb,
    time_step=time_step,
)

# Perform loops
x_start = X_gt[:, 0, np.newaxis]
X_record, W_record = particle_filt.loop_over_steps(x_start, U, Z)

# Visualize the result
mse = particle_filt.compute_MSE(X_gt, X_record, W_record, particle)
print("MSE: %.6f" % (mse))
particle_filt.visualize(X_gt, X_ODO, X_record, W_record, particle)
```

3.1.2 Result

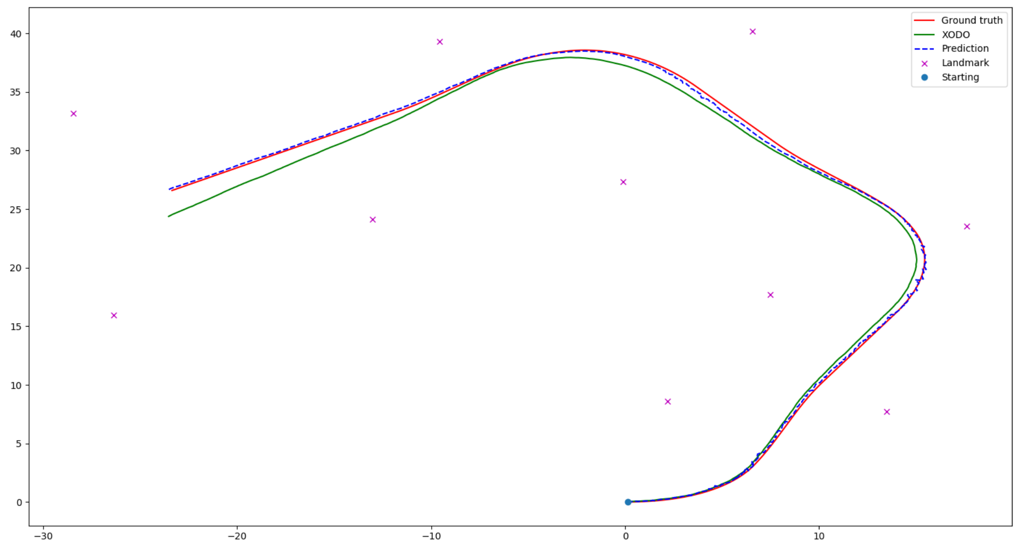
- Comment:
  - Overall, three trajectories of Prediction fits Ground truth with the same patterns. However, the best fit is belong to trajectoty with choosing max w_t, so the root mean square is smallest.
  - Line of XODO which is calculated from process model is different from Ground truth because of effect on range and bearing angle from thermal noise, while Prediction is calculated and chosen with importance factor w_t. That is the reason why line of Prediction is better than XODO.



(a) Trajectories with max w_t. $RMS = 11.289769$



(b) Trajectories with median w_t. $RMS = 11.428931$



(c) Trajectories with min w_t. $RMS = 11.475249$

Figure 1: Trajectories of Ground truth (XTRUE), XODO, Prediction and root mean square (RMS) of Prediction compared to Ground truth.

3.2 Extended Kalman Filter

3.2.1 Algorithm

- Extended Kalman Filter Overview:
  The extended Kalman filter (EKF) calculates an approximation to the true belief.It represents this approximation by a Gaussian. The belief is only approximate, not exact as the case in Kalman filters. Next state probability and the measurement probabilities are governed by nonlinear functions.

Algorithm Extended_Kalman_filter($\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$):

$\bar{\mu}_t = g(u_t, \mu_{t-1})$

$\bar{\Sigma}_t = G_t\, \Sigma_{t-1}\, G_t^T + R_t$

$K_t = \bar{\Sigma}_t\, H_t^T (H_t\, \bar{\Sigma}_t\, H_t^T + Q_t)^{-1}$

$\mu_t = \bar{\mu}_t + K_t(z_t - h(\bar{\mu}_t))$

$\Sigma_t = (I - K_t\, H_t)\, \bar{\Sigma}_t$

return μ_t, Σ_t

Figure 2: Extended Kalman Filter Algorithm

- Design the system model:
  Process model:

$$x_t = x_{t-1} + V_t \Delta t \cos(\theta_t + \varphi_{t-1}) \tag{4a}$$

$$y_t = y_{t-1} + V_t \Delta t \cos(\theta_t + \varphi_{t-1}) \tag{4b}$$

$$\varphi_t = \varphi_{t-1} + \frac{V_t \Delta t \sin \theta_t}{WB} \tag{4c}$$

  We model system as a nonlinear model plus noise:

$$x_t = g(u_t, x_{t-1}) + \epsilon_t \tag{5a}$$

  Calculate G by taking the Jacobian of g (nonlinear function):

$$G = \begin{bmatrix} 1 & 0 & -R\cos(\theta) + R\cos(\theta + \varphi) \\ 0 & 1 & -R\sin(\theta) + R\sin(\theta + \varphi) \\ 0 & 0 & 1 \end{bmatrix}$$

- Design the measurement model:
  Measurement model: r(t) is range between state of robot and position of landmark. The sensor provides bearing relative to the orientation of the robot, we subtract the robot's orientation from the bearing to get the sensor reading b(t)

$$z_t = h(x_t) + \delta_t \tag{6a}$$

$$r_t = \sqrt{(x_t - x_L)^2 + (y_t - y_L)^2} \tag{6b}$$

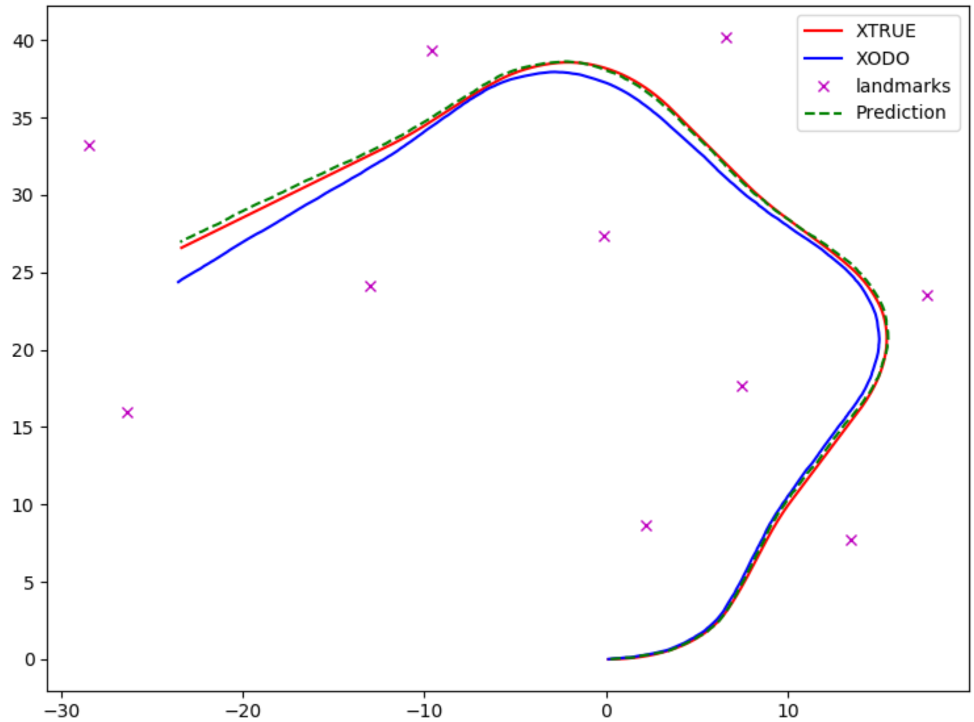
$$b_t = \arctan \frac{y_t - y_L}{x_t - x_L} + \varphi_t \tag{6c}$$

3.2.2 Python code

```python
#   Libraries
#------------------------------------------------------------------------------
from scipy.io import loadmat
from math import cos, sin, sqrt, atan2, tan
import matplotlib.pyplot as plt
import numpy as np
from numpy import array, dot
from numpy.random import randn
from EKF import ExtendedKalmanFilter as EKF

np.random.seed(2)
#------------------------------------------------------------------------------
#   Parameters
#------------------------------------------------------------------------------
sigma_v, sigma_g = 0.5, 3/180*np.pi
sigma_r, sigma_b = 0.2, 2/180*np.pi
wb = 4
time_step = 0.025

#------------------------------------------------------------------------------
#   Main execution
#------------------------------------------------------------------------------
# Load data
data = loadmat("data20171107.mat")
landmarks, X_gt, Z, U, XODO = data["lm"], data["XTRUE"], data["Z"], data["VG"], data["XODO"]
n_steps = X_gt.shape[1]

x_true, y_true, phi_true = X_gt[0,:], X_gt[1,:], X_gt[2,:]
x_odo, y_odo, phi_odo = XODO[0,:], XODO[1,:], XODO[2,:]
x_lm, y_lm = landmarks[0,:], landmarks[1,:]


def residual(a,b):
    # Bearing angle is normalized to [-pi, pi)
    if a[1] >= b[1]:
        y = a - b
    else:
        y = b - a
    y[1] = y[1] % (2*np.pi)
    if y[1] > np.pi:
        y[1] -= 2*np.pi
    return y


class RobotEKF(EKF):
    def __init__(self, dt, sigma_v, sigma_g, wheelbase = 4):
        EKF.__init__(self, 3, 2, 2)
        self.dt = dt
        self.wheelbase = wheelbase
        self.sigma_v = sigma_v
        self.sigma_g = sigma_g

    def predict(self, x, u, dt):
        v = u[0] + self.sigma_v * np.random.randn()
        theta = u[1] + self.sigma_g * np.random.randn()

        dist = v*dt
        phi = x[2]
        b = dist / self.wheelbase * sin(theta)

        sinhb = sin(theta + phi + b)
        coshb = cos(theta + phi + b)

        return x + array([[dist*coshb],
                          [dist*sinhb],
                          [b]])

def H_of(x, p):
    ''' Compute Jacobian of H matrix where h(x) computes the range and
    bearing to a landmark for state x '''

    px = p[0]
    py = p[1]
    hyp = (px - x[0, 0])**2 + (py - x[1, 0])**2
```

```python
    dist = np.sqrt(hyp)

    H = array(
        [[-(px - x[0, 0]) / dist, -(py - x[1, 0]) / dist, 0],
         [ (py - x[1, 0]) / hyp,  -(px - x[0, 0]) / hyp,  -1]])
    return H


def Hx(x, p):
    ''' Takes a state variable and returns the measurement that would
    correspond to that state.
    '''
    px = p[0]
    py = p[1]
    dist = np.sqrt((px - x[0, 0])**2 + (py - x[1, 0])**2)

    Hx = array([[dist],
                [atan2(py - x[1, 0], px - x[0, 0]) - x[2, 0]]])
    return Hx

def calculate_MSE(x_TRUE, x_predict, y_predict):
    x_true, y_true = x_TRUE[0,:], x_TRUE[1,:]
    MSE= 0.5 * np.sum((x_predict-x_true)**2 + (y_predict-y_true)**2)
    return MSE


dt = 0.025
ekf = RobotEKF(dt, wheelbase=4, sigma_v=sigma_v, sigma_g=sigma_g)
ekf.x = X_gt[:, 0, np.newaxis]

sigma_steer =  np.radians(2)

ekf.P = np.diag([1., 1., 1.])
ekf.R = np.diag([sigma_r**2, sigma_b**2])

xp = ekf.x.copy()
xp_0= np.array([])
xp_1= np.array([])

for k in range(len(U[0])):
    xp = ekf.predict(xp, U[:,k], dt)        # Simulate robot
    xp_0= np.append(xp_0,xp[0])
    xp_1= np.append(xp_1,xp[1])

    if k % 2 == 0:
        for x, y in zip(landmarks[0,:],landmarks[1,:]):
            d = sqrt((x - xp[0, 0])**2 + (y - xp[1, 0])**2)  + randn()*sigma_r
            a = atan2(y - xp[1, 0], x - xp[0, 0]) - xp[2, 0] + randn()*sigma_b
            z = np.array([[d], [a]])

            ekf.update(z, HJacobian=H_of, Hx=Hx, residual=residual,
                    args=([x,y]), hx_args=([x,y]))

# Visualize the result

MSE= calculate_MSE(X_gt, xp_0, xp_1)
print("MSE= %.3f" % (MSE))

plt.plot(x_true, y_true, "-r")
plt.plot(x_odo, y_odo, "b")
plt.plot(x_lm, y_lm, "xm")
plt.plot(xp_0, xp_1, "--g")
plt.legend(["XTRUE","XODO","landmarks","Prediction"])
plt.show()
```

3.2.3 Result



(a) Root Mean Square (RMS) = 26.201

Figure 3: Position of lanmarks, path of Ground truth (XTRUE), XODO, Prediction(using EKF) and RMS.