Département EEA - Faculté Sciences et Ingénierie

M2 - AURO

Projet Bloc 3

Année 2022-2023

Filtre Kalman Etendu sur robot

Rédigé par ABDESSALEM Younes et HAUSS Katell

30 avril 2023

1 Introduction

Notre troisième projet de cette année de M2 AURO concernait le déplacement de deux robots de manière autonome à l'aide de la perception de différents amers connus. Une première partie portait sur la simulation du comportement du robot, et une seconde partie sur la réalisation du filtrage de Kalman étendu (EKF). Une fois cette phase d'étude réalisée sous Python, il nous fallait commander deux robots différents équipés de caméra à l'aide d'une manette dans le but qu'ils puissent retrouver leur position. Pour cela, le ROS nous était nécessaire pour la phase de communiquation avec le robot. Dans ce rapport, nous présenterons notre travail ayant porté sur la partie simulation/filtrage en Python.

2 Filtre EKF

Dans notre cas, nous avons décidé de déplacer le robot entre deux rangés de quatre amers. Une fois la rangée parcourue, le robot effectue une rotation et se retrouve au dessus des amers les plus hauts, puis avance encore en ligne droite avant de tourner de nouveau jusqu'à retrouver sa position d'origine. En simulation, nous obtenons la figure 1 où nous pouvons observer les amers, le robot et son parcours.

Afin de calculer le déplacement du robot, dans les lignes droites qu'il doit parcourir, l'angle passé en commande est de 0° donc on utilise les formules trigonométriques de base.

$$\begin{aligned}
r_x(t+1) &= r_x(t) + u_d(t) * cos\Big(r_w(t)\Big) \\
r_y(t+1) &= r_y(t) + u_d(t) * sin\Big(r_w(t)\Big) \\
r_w(t+1) &= r_w(t) + u_w(t)
\end{aligned}$$

Si son orientation change, alors il faut prendre en compte ce changement d'orientation :

$$\begin{aligned}
r_x(t+1) &= r_x(t) + \frac{u_d(t)}{u_w(t)} * \Big[sin\big(r_w(t) + u_w(t)\big) - sin\big(r_w\big)\Big] \\
r_y(t+1) &= r_y(t) + \frac{u_d(t)}{u_w(t)} * \Big[cos\big(r_w\big) - cos\big(r_w(t) + u_w(t)\big)\Big] \\
r_w(t+1) &= r_w(t) + u_w(t)
\end{aligned}$$

Concernant les observations, celles-ci seront composées de la distance et de l'angle entre le robot et chaque amer.

$$z_d = \sqrt{(amer_x - r_x)^2 + (amer_y - r_y)^2} \tag{1}$$

$$z_w = atan2(amer_y - r_y, amer_x - r_x) - r_w \tag{2}$$

Pour simuler les limites des composants physiques, le robot ne sera plus capable d'observer les amers associés si la distance ou l'angle dépasse un certain seuil et on donnera une valeur NaN (Not a Number) afin de le signifier. Nous avons décidé arbitrairement d'imposer une limite de 3 pour la distance et de $\frac{3*\pi}{4}$ pour l'angle de la vision du robot.

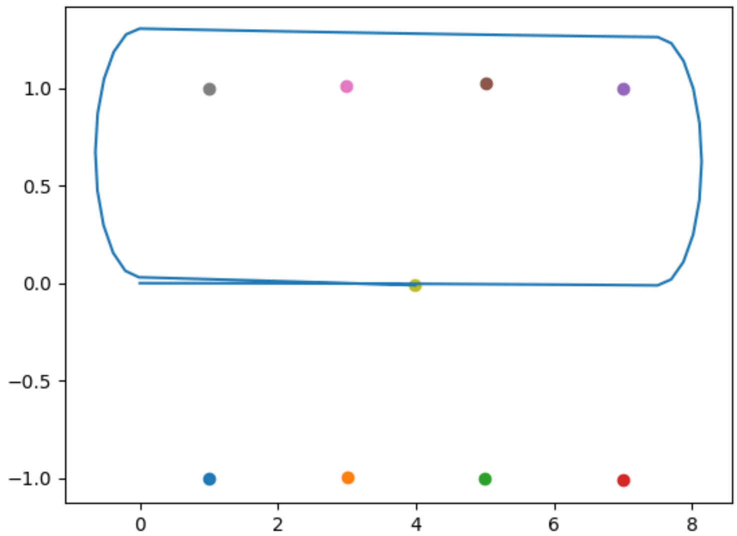

FIGURE 1 – Trajectoire du robot

Etant donné que la trajectoire est générée par une équation non-linéaire, nous ne pouvons pas appliquer le filtre de Kalman classique mais celui étendu. En effet, celui-ci utilise une approximation linéaire du modèle non-linéaire pour prédire l'état du système et la covariance associée. À l'aide de la matrice Jacobienne, le filtre de Kalman étendu linéarise le modèle mathématique du système autour de l'estimation courante de l'état. Afin de construire un filtre de Kalman étendu, nous devons procéder à deux phases distinctes qui sont celles de "Prédiction" et de "Mise à jour". Ces deux phases sont composées respectivement de deux et trois équations.

Prédiction :

$$x_{est}(t) = f\Big(x_{maj}(t-1)\Big) \tag{3}$$

$$P_{est}(t) = F * P_{maj}(t-1) * F^T + Q \tag{4}$$

Mise à jour :

$$K = P_{est} * H^T * S^{-1} \tag{5}$$

$$P_{maj} = P_{est} - K * H * P_{est} \tag{6}$$

$$x_{maj} = x_{est} + K * (z - z_{est}) \tag{7}$$

Après avoir réalisé le filtre avec la simulation, nous avons obtenu le résultat suivant (figure 2) qui démontre son bon fonctionnement.

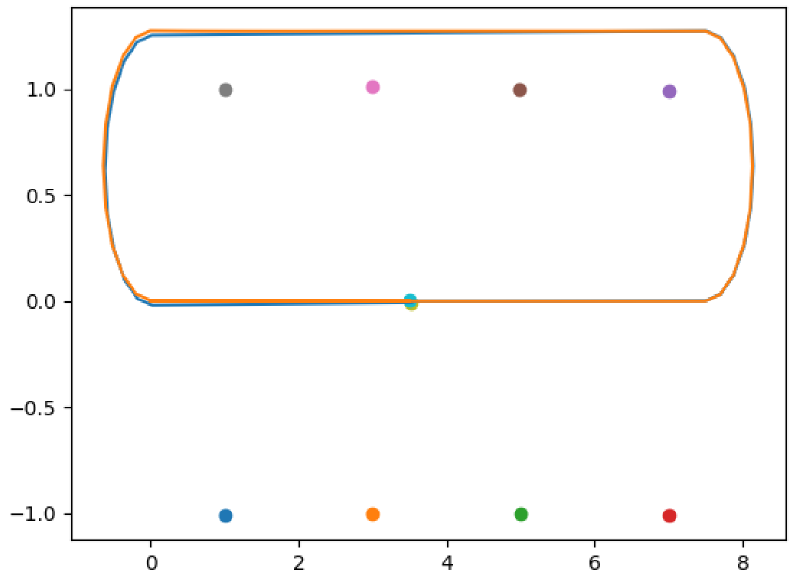

FIGURE 2 – Résultat

3 Conclusion

En somme, ce projet nous a permis d'explorer davantage le filtrage de Kalman étendu ainsi que de plus pratiquer le Python et le ROS. En comparaison avec notre premier projet portant sur le filtrage de Kalman simple, ici nous avons appliqué la version EKF qui est adaptée à la non-linéarité de notre problème. L'utilité de l'EKF étant primordiale dans la navigation de robots autonomes, sa compréhension pourra nous être nécessaire pour le futur.

4 Annexe

```python
## Projet BLOC 3
# ABDESSALEM Younes, HAUSS Katell

import matplotlib.pyplot as plt
import numpy as np
import math


### Fonction

# Affichage
def affichage(pos_a: float, r: float, x_maj: float, t: int):
    # Affichage de la map
    for i in range(0, pos_a.shape[0], 2):
        plt.scatter(pos_a[i], pos_a[i + 1])

    # Affichage de la trajectoire du robot dans la map
    plt.plot(r[:t+1, 0], r[:t+1, 1])
    # Affichage du robot dans son dernier etat
    plt.scatter(r[t, 0], r[t, 1])
    # Affichage des  tats  robot
```

```python
    plt.plot(x_maj[:t+1, 0], x_maj[:t+1, 1])
    # Affichage du robot dans son dernier etat predit
    plt.scatter(x_maj[t, 0], x_maj[t, 1])

    plt.show()


# Generation amers
def generation_amers(Nb_amer : int, x : float, y : float, incertitude_amer : float) :
    m = np.ndarray((Nb_amer*2))

    for i in range(int(Nb_amer / 2)):
        m[i*2] = x
        m[i*2+1] = y
        x = x + 2
    i = i + 1
    x = x - 2
    y = y + 2
    while i < Nb_amer:
        m[i * 2] = x
        m[i * 2 + 1] = y
        x = x - 2
        i = i + 1

    m = m + (np.linalg.cholesky(incertitude_amer))@(np.random.normal(
        size=(16)))

    return m

# Calcul position robot
def avance_robot(r : float, u : float, w : float, position_amer : float, t : float):
    if (u[t, 1] == 0):
        r[t+1, 0] = r[t, 0] + (u[t, 0]) * math.cos(r[t, 2]) + w[0,0]
        r[t+1, 1] = r[t, 1] + (u[t, 0]) * math.sin(r[t, 2]) + w[0,1]
        r[t+1, 2] = r[t, 2] + u[t, 1] + w[0,2]
        r[t+1, 3:] = position_amer + w[0,3:]
    else:
        r[t+1, 0] = r[t, 0] + (u[t, 0] / u[t, 1]) * (math.sin(r[t, 2] +
                u[t, 1]) - math.sin(r[t, 2])) + w[0,0]
        r[t+1, 1] = r[t, 1] + (u[t, 0] / u[t, 1]) * (math.cos(r[t, 2]) -
                math.cos(r[t, 2] + u[t, 1])) + w[0,1]
        r[t+1, 2] = r[t, 2] + u[t, 1] + w[0,2]
        r[t+1, 3:] = position_amer + w[0,3:]
    return r

# Generation trajectoire
def generation_trajectoire(nb_amer : float, x : float, position_amers : float, u : float, w : float, x_r : float, y_r : float, theta_r : float, t : float):
    i = 0
    x[0, :3] = [x_r, y_r, theta_r]
    dist = x[0, 0] - position_amers[nb_amer]  # x_robot - x_amer_(4)

    # Aller en x
    while(dist < 0):
        u[t] = np.array([0.5, 0])
        x = avance_robot(x, u, w, position_amers, t)
        dist = x[i, 0] - position_amers[nb_amer]
        i += 1
        t += 1

```

```python
    # Rotation
    while (x[i, 2] < np.pi * 0.97):
        u[t] = np.array([0.2, np.pi/10])
        x = avance_robot(x, u, w, position_amers, t)
        i += 1
        t += 1

    # Retour en x
    dist = x[i, 0] - position_amers[0]
    while (dist > 0):
        u[t] = np.array([0.5, 0])
        x = avance_robot(x, u, w, position_amers, t)
        dist = x[i, 0] - position_amers[0]
        i += 1
        t += 1

    # Rotation
    while (x[i, 2] < np.pi * 2 * 0.97):
        u[t] = np.array([0.2, np.pi/10])
        x = avance_robot(x, u, w, position_amers, t)
        i += 1
        t += 1

    # Aller 2 en x
    dist = x[i, 0] - position_amers[2]
    while (dist < 0 ):
        u[t] = np.array([0.5, 0])
        x = avance_robot(x, u, w, position_amers, t)
        dist = x[i, 0] - position_amers[2]
        i += 1
        t += 1
    return t, x, u



# Generation mesures
def generation_mesure(nb_amer : float, z1:float, r1:float, pos_a:float, v:
    float, t : float):
    for i in range(t):
        for e in range(nb_amer):
            z1[i, e*2] = math.sqrt((pos_a[2 * e] - r1[i, 0]) ** 2 +
                        (pos_a[2 * e + 1] - r1[i, 1]) ** 2) + v[i,2*e]
            z1[i, 2*e+1] = (math.atan2(pos_a[2*e+1] - r1[i,1], pos_a[2*e] -
                        r1[i,0]) - r1[i,2] + v[i,e+1]) %(np.pi)

            if (z1[i, e * 2] > 3 or abs(z1[i, e * 2 + 1]) > 3*np.pi/4 ):
                z1[i, e * 2] = np.nan
                z1[i, e * 2 + 1] = np.nan
    return z1

# Filtre
def filtre(x_pred : float, P_pred : float, x_maj : float, P_maj : float,
    K : float, z: float, u : float, position_amer : float, nb_amer : float,
     t : float):
    Qw_pred = np.diag(0.000000001 * np.ones(3 + 2 * nb_amer))
    for i in range(t):
        print("\nInstant n  ", i)

        # Recup obs sans 'nan'
        amers_visibles = []
        for j in range(nb_amer):
            if (np.isnan(z[i,2 * j]) == False):
                amers_visibles = amers_visibles + [j]
```

```
        print("amers_visibles : ", amers_visibles)

        # Prediction
        if (u[i, 1] == 0):
            x_pred[i+1, 0] = x_maj[i, 0] + u[i, 0] * math.cos(x_maj[i, 2])
            x_pred[i+1, 1] = x_maj[i, 1] + u[i, 0] * math.sin(x_maj[i, 2])
            x_pred[i+1, 2] = x_maj[i, 2] + u[i, 1]
            x_pred[i+1, 3:] = position_amer
        else:
            x_pred[i+1, 0] = x_maj[i, 0] + (u[i, 0] / u[i, 1]) * (math.sin(
                        x_maj[i, 2] + u[i, 1]) - math.sin(x_maj[i,2]))
            x_pred[i+1, 1] = x_maj[i, 1] + (u[i, 0] / u[i, 1]) * (math.cos(
                        x_maj[i, 2]) - math.cos(x_maj[i, 2] + u[i, 1]))
            x_pred[i+1, 2] = x_maj[i, 2] + u[i, 1]
            x_pred[i+1, 3:] = position_amer

        F = jacF(x_pred, u, i)

        P_pred = F @ P_maj @ F.T + Qw_pred

        z_visible = np.zeros((1, len(amers_visibles * 2)))
        z_pred = np.zeros((1, len(amers_visibles * 2)))
        H = np.zeros((len(amers_visibles * 2), 19))
        Rv = np.zeros((len(amers_visibles * 2), len(amers_visibles * 2)))
        for o in range(len(amers_visibles)):
            Rv[2 * o][2 * o] = 0.1
            Rv[2 * o + 1][2 * o + 1] = np.pi / 10

        for e in range(len(amers_visibles)):
            z_visible[0, e * 2] = z[i,2 * amers_visibles[e]]
            z_visible[0, 2 * e + 1] = z[i,2 * amers_visibles[e] + 1]
            z_pred[0, e * 2] = math.sqrt((x_pred[i, 0] - position_amer[
                        amers_visibles[e] - 1]) ** 2 + (x_pred[i, 1] -
                        position_amer[amers_visibles[e]]) ** 2)
            z_pred[0, 2 * e + 1] = math.atan2(position_amer[
                    amers_visibles[e]] - x_pred[i, 1], position_amer[
                    amers_visibles[e] - 1] - x_pred[i, 0]) - x_pred[i, 2]


            H[2*e, 0] = -2 * (position_amer[amers_visibles[e]] -
                x_pred[i, 0]) * 1 / (2 * math.sqrt((x_pred[i, 0] -
                position_amer[2 * amers_visibles[e]]) **2 + (x_pred[i, 1] -
                position_amer[2 * amers_visibles[e] + 1]) ** 2))
            H[2*e, 1] = -2 * (position_amer[amers_visibles[e] + 1] -
                x_pred[i, 1]) / (2 * math.sqrt((x_pred[i, 0] -
                position_amer[2 * amers_visibles[e]]) **2 + (x_pred[i, 1] -
                position_amer[2 * amers_visibles[e] + 1]) ** 2))
            H[2*e, 3+2*amers_visibles[e]] = 2 * (position_amer[
                amers_visibles[e]] - x_pred[i, 0]) / (2 * math.sqrt((
                x_pred[i, 0] - position_amer[2 * amers_visibles[e]]) **2 +
                (x_pred[i, 1] - position_amer[2 * amers_visibles[e] + 1])
                ** 2))
            H[2*e, 4+2*amers_visibles[e]] = 2 * (position_amer[
                amers_visibles[e] + 1] - x_pred[i, 1]) / (2 * math.sqrt((
                x_pred[i, 0] - position_amer[2 * amers_visibles[e]]) ** 2 +
                (x_pred[i, 1] - position_amer[2 * amers_visibles[e] + 1])
                ** 2))
            #
            H[2*e+1, 0] = (position_amer[amers_visibles[e] + 1] - x_pred[i,
                 2]) / ((position_amer[amers_visibles[e]] - x_pred[i, 1])
                 ** 2 + (position_amer[amers_visibles[e] + 1] - x_pred[i,
                 2]) ** 2)
```

```
            H[2*e+1, 1] = -(position_amer[amers_visibles[e]] - x_pred[i, 1]) / ((position_amer[amers_visibles[e]] - x_pred[i, 1]) ** 2 + (position_amer[amers_visibles[e] + 1] - x_pred[i, 2]) ** 2)
            H[2*e+1, 2] = -1
            H[2*e+1, 3+2*amers_visibles[e]] = -(position_amer[amers_visibles[e] + 1] - x_pred[i, 2]) / ((position_amer[amers_visibles[e]] - x_pred[i, 1]) ** 2 + (position_amer[amers_visibles[e] + 1] - x_pred[i, 2]) ** 2)
            H[2*e+1, 4+2*amers_visibles[e]] = (position_amer[amers_visibles[e]] - x_pred[i, 1]) / ((position_amer[amers_visibles[e]] - x_pred[i, 1]) ** 2 + (position_amer[amers_visibles[e] + 1] - x_pred[i, 2]) ** 2)

        S = Rv + H @ P_pred @ np.transpose(H)

        # Mise a jour
        K = P_pred @ H.T @ np.linalg.inv(S)

        x_maj = x_pred + K @ (z_visible[0] - z_pred[0])

        P_maj = P_pred - K @ H @ P_pred
    return  x_maj


# Calcul de F
def jacF(x_pred : float, u : float, i : int):
    F_sa = np.zeros((19, 19))
    if (u[i, 1] == 0):
        F_sa[0, 0] = 1
        F_sa[0, 2] = -u[i, 0] * math.sin(x_pred[i, 2])
        F_sa[1, 1] = 1
        F_sa[1, 2] = u[i, 0] * math.cos(x_pred[i, 2])
        F_sa[2, 2] = 1

    else:
        F_sa[0, 0] = 1
        F_sa[0, 2] = (u[i, 0] / u[i, 1]) * (math.cos(x_pred[i, 2] + u[i, 1]) - math.cos(x_pred[i, 2]))
        F_sa[1, 1] = 1
        F_sa[1, 2] = (u[i, 0] / u[i, 1]) * (math.sin(x_pred[i, 2] + u[i, 1]) - math.sin(x_pred[i, 2]))
        F_sa[2, 2] = 1

    return F_sa





### Main
def main():
    # Initialisation
    t = 0
    x_r = 0
    y_r = 0
    theta_r = 0
    nb_amer = 8
    x = np.zeros((100,19))
    u = np.zeros((100,2))
    z = np.zeros((100,16))
    Qw = np.diag(np.ones(19) * 0.0000001)
```

```python
    w = np.transpose((np.linalg.cholesky(Qw)) @ (np.random.normal(size=(19,
         100))))
    Rv = np.diag(0.000001 * np.ones(2 * nb_amer))
    v = np.transpose((np.linalg.cholesky(Rv)) @ (np.random.normal(size=(2 *
         nb_amer, 100))))

    # Init Filtre
    x_pred = np.zeros((100, 19))
    P_pred = np.zeros((19, 19))
    x_maj = np.zeros((100, 19))
    P_maj = np.zeros((19, 19))
    K = np.zeros((19, 16))

    incertitude_amer = np.diag(0.0001 * np.ones(2 * nb_amer))

    position_amer = generation_amers(nb_amer, 1, -1, incertitude_amer)
    t, x, u = generation_trajectoire(nb_amer, x, position_amer, u, w, x_r,
             y_r, theta_r, t)
    z = generation_mesure(nb_amer, z, x, position_amer, v, t)

    x_maj = filtre(x_pred, P_pred, x_maj, P_maj, K, z, u, position_amer,
             nb_amer, t)

    affichage(position_amer, x, x_maj, t)



if __name__ == "__main__":
    main()
```