Algorithmique, robotique et systèmes temps réel (RO52)

TP N°5 – FILTRE DE KALMAN

Sébastien DOUTRELUINGNE, Samuel BERNARD

16/06/2023

Introduction

Dans le cadre de l'UV RO52, nous avons pu lors de différents travaux pratiques mettre en application les concepts abordés lors des cours magistraux. Ainsi, durant les précédentes séances, nous avions pu implémenter des programmes de suivi de ligne avec les algorithmes de PID, ou développer des fonctionnalités permettant à un robot de suivre efficacement un autre.

Pour cela, l'ensemble de ces notions ont pu être expérimentées grâce à l'utilisation des robots EV3 de LEGO, nous permettant de constater physiquement l'efficacité de nos programmes. Ces robots ont notamment la particularité d'être modulaire, par l'ajout de différents capteurs de couleur ou de profondeur.

Ainsi, pour cette dernière séance pratique, l'objectif était d'exploiter les résultats des capteurs tels qu'un gyroscope et du modèle de la cinématique du robot afin d'obtenir les coordonnées x et y de sa trajectoire en se déplaçant sur un circuit en 8. Par la suite, l'application du filtre de Kalman sur l'angle visait à permettre au robot d'obtenir une position plus précise.

Pour ce faire, ce TP se déroule en deux étapes. Une première étape consiste à effectuer les mesures nécessaires grâce à différentes méthodes pour modéliser la trajectoire du robot. La seconde vise à utiliser le filtre de Kalman pour améliorer la précision de cette trajectoire.

1. Réalisation des mesures

Tout d'abord la première étape a consisté à réaliser différentes mesures afin de déterminer la trajectoire du robot selon ses coordonnées x et y. Ainsi, nous avons exploité différentes méthodes pour calculer au cours du temps chacune de ces coordonnées, en faisant circuler le robot sur le circuit en 8 deux fois.

L'objectif de chacun de ces méthodes est d'exploiter les données calculées ou mesurées telles que la vitesse angulaire et la vitesse du robot, sa distance parcourue, ou son angle cumulé afin de calculer les coordonnées x et y du robot selon la formule suivante :

$$x(t+\Delta t) = x(t) + V.\Delta t.\cos\left(\frac{\varphi(t+\Delta t)+\varphi(t)}{2}\right)$$

$$y(t+\Delta t) = y(t) + V.\Delta t.\sin\left(\frac{\varphi(t+\Delta t)+\varphi(t)}{2}\right)$$

Avec :

- V la vitesse du robot
- Δt le pas de temps
- φ(t) l'angle du robot à l'instant t avec notamment $\varphi(t+\Delta t) = \varphi(t) + \omega.\Delta t$
- ω la vitesse angulaire du robot

Par ailleurs, pour l'ensemble de ces expérimentations, la vitesse V du robot est fixée sur 100 mm/s.

1.1. Première méthode

Ainsi, nous avons tout d'abord exploité les données issues de l'application de la commande PID (vitesse angulaire du robot) afin de déterminer les coordonnées du robot. En effet, les valeurs de coordonnées x et y sont ainsi calculés à chaque itération :

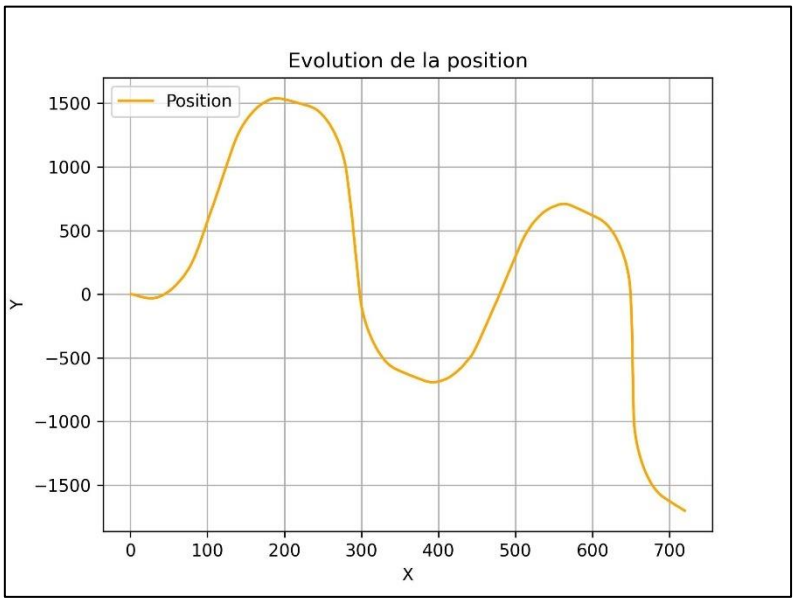
```
speed = 100
angular_speed = 0
angle = 0
x = 0
y = 0
timestamp = time.time()
elapsed_time = 0
clear_log()

while 1:
    # Calcul du PID
    […]
    elapsed_time = time.time() - timestamp
    timestamp = time.time()
    x = x + speed * elapsed_time * math.cos((math.pi / 180 * angular_speed
* elapsed_time + angle) / 2)
    y = y + speed * elapsed_time * math.sin((math.pi / 180 * angular_speed
* elapsed_time + angle) / 2)
    angle = angle + math.pi / 180 * angular_speed * elapsed_time
    write_log(x, y, angle, speed)
```

Dans ce programme, la variable *elapsed_time* permet de récupérer la valeur du temps actuel grâce à la fonction de la librairie *time*. On constate également que les coordonnées x et y sont calculées à partir des formules décrites ci-dessus. Cependant, on peut noter que la valeur d'angle *angle* est calculée grâce à la valeur de vitesse angulaire *angular_speed* retournée par la fonction de calcul du PID selon la formule présentée également précédemment $\varphi(t + \Delta t) = \varphi(t) + \omega . \Delta t$. De plus, cette valeur d'angle initialement en degrés est convertie en radians en la multipliant par $\frac{\pi}{180}$.

On peut également observer les deux fonctions *clear_log()* et *write_log()* issues de la librairie *log* développée par nous-mêmes, qui permettent d'initialiser un fichier log.txt et de l'ouvrir en écriture pour la première, et d'écrire dans celui-ci les données x, y, angle et speed à chaque itération.

Enfin, l'exécution du programme evcurves.py en prenant en paramètres le fichier log.txt ainsi qu'un dossier de destination permet de générer la courbe représentative de la trajectoire obtenue en représentant les coordonnées successives de y par rapport à celles en x.

Voici ainsi la courbe de la trajectoire du robot obtenue avec cette première méthode :



On peut en effet constater que la trajectoire obtenue ne correspond pas au circuit en 8 sur lequel le robot a réalisé son parcours. En effet, ce type de courbes sinusoïdales est obtenue lorsque la valeur de l'angle de rotation du robot mesurée est sous-estimée. Ainsi, pour corriger cette trajectoire, nous pouvons diminuer la valeur de l'empâtement des roues du robot. En effet, nous avons souhaité implémentée la stratégie de correction offline, qui consiste à corriger les valeurs des angles, en appliquant le modèle, jusqu'à l'obtention des meilleures courbes possibles. Cependant, nous n'avons pu obtenir de courbes d'avantage satisfaisantes par cette méthode.

1.2. Seconde Méthode

Ensuite, nous avons souhaité exploiter fonction *state()* qui renvoie la distance parcourue, la vitesse, l'angle et la vitesse angulaire du robot afin de déterminer les coordonnées du robot. Ainsi, le programme développé est similaire au précédent en intégrant cependant les données de la fonction *state()* dans le calcul des coordonnées :

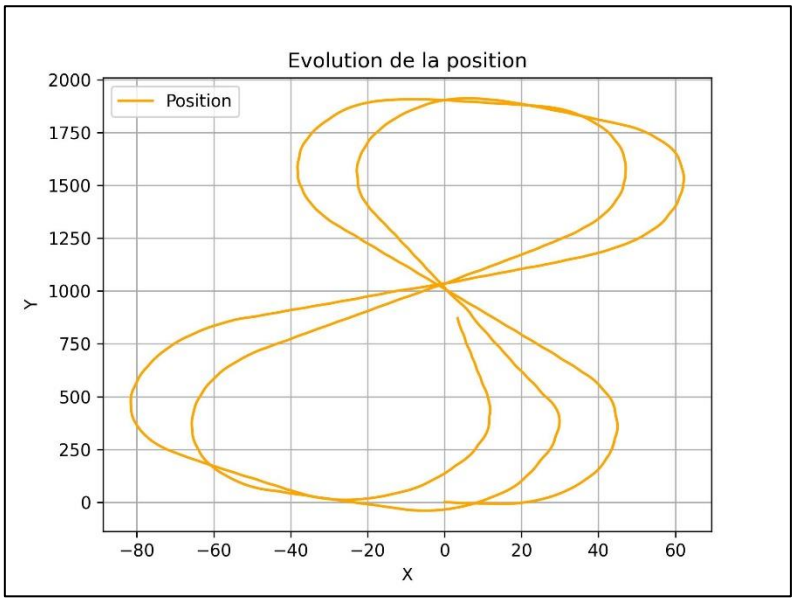
```
old_angle = 0
old_distance_state = 0
distance_state = 0
angular_speed_state = 0
angle = 0
x = 0
y = 0
timestamp = time.time()
elapsed_time = 0
clear_log()

while 1:
    state = robot.state()
    old_distance_state = distance_state
    distance_state = state[0]
    angular_speed_state = state[3]
    old_angle = angle
    angle = math.pi / 180 * state[2]

    # Calcul du PID
    […]

    x = x + (distance_state - old_distance_state) * math.cos((angle +
old_angle) / 2)
    y = y + (distance_state - old_distance_state) * math.sin((angle +
old_angle) / 2)

    write_log(x, y, angle, speed)
```

Ainsi, on peut tout d'abord constater qu'à chaque début d'itération, le programme va récupérer les données mesurées par le robot issues de la fonction *state()* de la classe *DriveBase*. En effet, les données retournées par cette fonction correspondent à un tableau dans lequel chaque élément correspond à une information sur l'état du robot :

- Le 1er élément correspond à la distance parcourue par le robot depuis le lancement du programme
- Le 3ème élément correspond à la valeur de l'angle
- Le 4ème élément correspond à la valeur de la vitesse angulaire du robot en degrés, que l'on convertit en radians

On peut notamment observer que la distance parcourue depuis la dernière itération précédemment calculée par $V.\Delta t$ est calculée ici en réalisant la différence entre la distance totale parcourue à l'itération actuelle *distance_state* et la distance totale parcourue à l'itération précédente *old_distance_state*.

Voici alors la courbe de la trajectoire du robot obtenue avec cette seconde méthode :



On peut alors constater la bonne interprétation des données mesurées par la fonction *state()* puisque la trajectoire obtenue s'apparente au circuit en 8 sur lequel le robot a évolué. Cependant, on peut constater des décalages d'un tour à l'autre dans la trajectoire empruntée par le robot. Ces erreurs seront par la suite corrigées par le filtre de Kalman.

Nous n'avons cependant pas implémenté de stratégies de correction afin de corriger l'allure de la trajectoire modélisée en raison car celle-ci est fidèle au circuit en 8.

1.3. Troisième méthode

Enfin, pour cette dernière méthode, nous avons intégré un gyroscope au robot afin d'exploiter les données que celui-ci peur retourner telles que la vitesse angulaire ou angle cumulé afin de déterminer les coordonnées x et y du robot au cours du temps.

Pour se faire, nous avons implémenté une classe Gyroscope avec pour attributs :

- La coordonnée x initialisée à 0
- La coordonnée y initialisée à 0
- La valeur de l'angle mesuré à l'itération précédente *prev_angle* initialisé à 0
- La valeur de l'angle mesuré à l'itération actuelle *angle* initialisé à 0
- Le temps actuel *timestamp* initialisé au temps actuel avec la librairie time
- Le temps écoulé depuis la dernière itération *elapsed_time* initialisé à 0

Ainsi que d'autres attributs initialisés dans le constructeur de la classe :

- Un gyroscope *sensor* correspondant à une instance de la classe *GyroSensor*
- La valeur de la vitesse *speed* initialisée à 100 mm/s

Le constructeur de la classe va également initialisé réinitialisé la valeur de l'angle de l'objet sensor à 0 grâce à la fonction resetAngle qui permet de définir la valeur de l'angle actuel du gyroscope à une valeur donnée.

Ensuite la fonction *getCoordinate* va permettre de calculer les coordonnées x et y actuelle du robot grâce aux données renvoyées par le gyroscope :

```
def getCoordinate(self):
        self.elapsed_time = time.time() - self.timestamp
        self.timestamp = time.time()
        self.setAngle()
        self.X =
self.X+self.speed*self.elapsed_time*math.cos((self.angle+self.prev_angle)/2)
        self.Y =
self.Y+self.speed*self.elapsed_time*math.sin((self.angle+self.prev_angle)/2)
        self.prev_angle = self.angle

        return self.X, self.Y
```

Pour cela, après avoir mis à jour le temps écoulé depuis la dernière itération elapsed_time, la fonction met également à jour la valeur de la variable *angle* grâce à la fonction *setAngle* définie de la façon suivante :

```
def setAngle(self):
        self.angle = (math.pi*self.sensor.angle())/180
```

Cette fonction récupère la valeur de l'angle actuel mesuré par le gyroscope qui correspond à l'angle accumulé par le gyroscope depuis le lancement du programme.

Ensuite, les coordonnées x et y sont déterminées de façon similaire à la première méthode développée, avant d'être retournées.

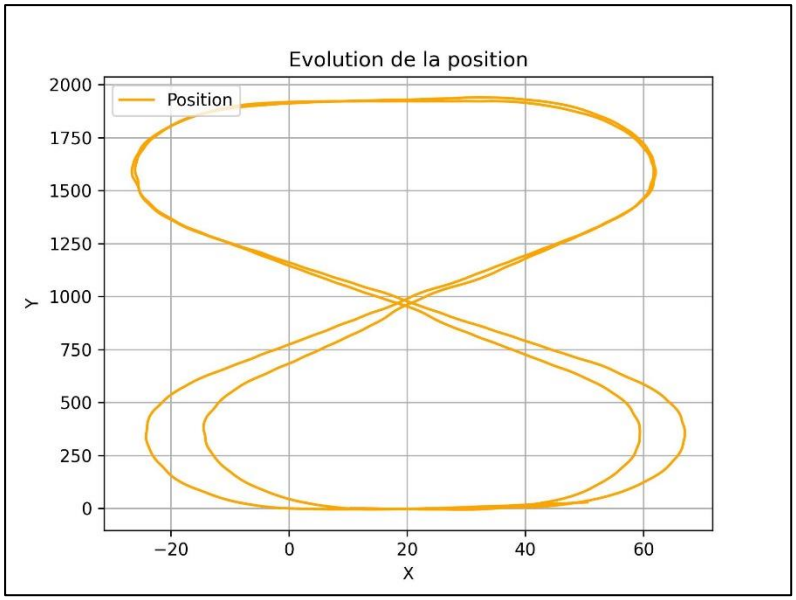
Ainsi, voici les différentes étapes dans le programme principal du fichier main afin d'exploiter les données du gyroscope pour calculer les coordonnées du robot :

```
speed = 100
angular_speed = 0
angle = 0
x = 0
y = 0

gyroscope = Gyroscope(Port.S4, speed)

timestamp = time.time()
elapsed_time = 0

clear_log()

while 1:
    # Calcul du PID
    […]
    x, y = gyroscope.getCoordinate()
    write_log(x, y, gyroscope.angle, speed)
```

Ainsi, on peut constater qu'une instance de la classe *Gyroscope* est créée en indiquant que le capteur est sur le port 4 du robot, et que sa vitesse est de 100 mm/s.

Ensuite, les valeurs des coordonnées sont calculées grâce à la fonction *getCoordinate* de la classe Gyroscope.

Voici enfin la courbe de la trajectoire du robot obtenue avec cette dernière méthode :



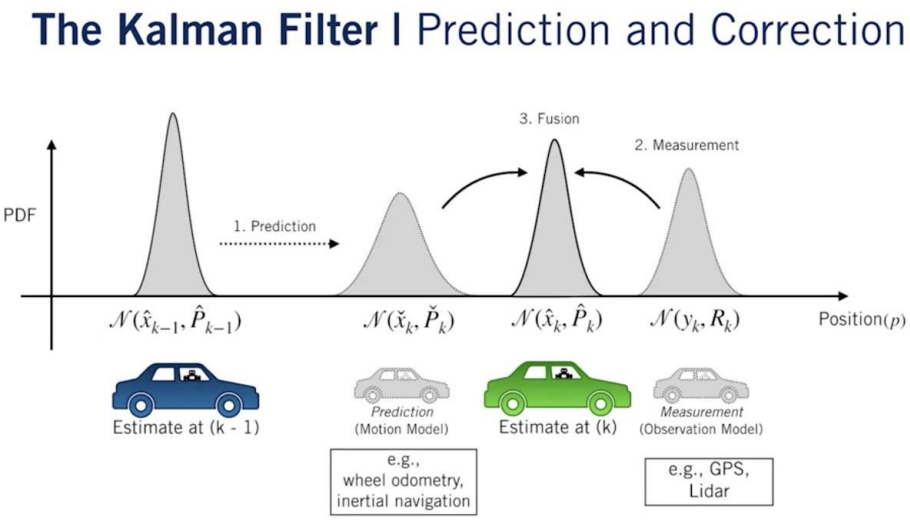
On constate alors que la trajectoire modélisée grâce aux données du gyroscope est fidèle à la trajectoire réelle du robot. On peut ainsi observer davantage de précision dans la trajectoire du robot notamment dans la boucle supérieure du 8 qui est quasi-identique sur les deux tours du robot. Cependant, comme pour la seconde méthode, nous n'avons implémentée de stratégie de correction offline du robot afin de corriger les valeurs d'angle ou de distance obtenus en raison de la précision de la courbe.

2. Utilisation du filtre de Kalman

Nous pouvons remarquer que lors que nous avons effectué nos mesures, les coordonnées calculées n'étaient pas les mêmes lorsque le robot faisait un deuxième tour sur un même circuit. En effet, nous pouvons rencontrés deux types de problème sur les instruments de mesure : la précision et l'exactitude ce qui provoque l'oscillation du système lorsqu'il est couplé avec la commande.

Nous avons utilisé le filtre de Kalman qui est une technique d'estimation et de prédiction permettant d'estimer l'état d'un système dynamique en combinant les mesures provenant de capteurs avec un modèle mathématique de ce système. Notre capteur correspond à un gyroscope qui nous permet d'avoir l'angle cumulé au cours du temps ce qui nous a permit pas la suite de calculer les coordonnées.

Etant donné que nous avons des problèmes de mesures, ce qui est souvent le cas dans les systèmes du monde réel, le filtre de Kalman nous sera très utile pour avoir une estimation plus précise de la trajectoire du robot. Nous allons l'expérimenter en robotique en utilisant un robot EV3 afin d'obtenir un angle qui correspond au plus proche de la réalité en se basant non plus exclusivement sur la mesure mais aussi sur l'estimation de l'angle. Il fonctionne en combinant les informations provenant des mesures actuelles avec les estimations précédentes de l'état du système pour obtenir une estimation plus précise de l'état actuel. Il prend en compte à la fois l'incertitude des mesures et les erreurs de modélisation pour calculer une estimation optimale de l'état du système.



Nous avons créé une classe qui s'appelle Kalman. Elle dispose de de 2 fonctions, l'initialisation et l'application du filtre dans notre système. L'implémentation de ce système fait intervenir plusieurs formules que nous allons expliquer plus en détail dans ce rapport. Mais avant de commencer, nous allons utiliser un ensemble de variables que vous pouvez voir juste en dessous :

- $\hat{\alpha}_{n,n}$: estimation de l'angle après n mesures
- $\hat{\alpha}_{n,n-1}$: estimation de l'angle à l'étape $n-1$
- K_n : Gain du filtre de Kalman
- $\hat{y}_n$: estimation de l'erreur
- z_n : mesure de l'angle à l'étape n
- $V_{\alpha_{n,n-1}}$: incertitude réalisée de l'estimation de l'angle réalisée à l'étape $n-1$
- R_n : erreur (incertitude) de mesure initialisée à 10°.
- $V_{\alpha_{n-1,n-1}}$: incertitude réalisée à l'étape $n-1$ de l'estimation de l'angle réalisée à l'étape $n-1$
- ω : Bruit du processus : composante du filtre qui représente les variations et les erreurs imprévisibles dans le système dynamique. Après plusieurs tests, il a été estimé que l'erreur est de 10°
- C_{n-1} : Commande du système qui correspond à l'angle calculée après correction à l'étape $n-1$
- B : matrice de gain (ou de transition) du filtre de Kalman
- $\hat{\alpha}_{n-1,n-1}$: estimation de l'angle précédent réalisée à l'étape $n-1$
- Q_n : covariance du bruit processus : constante
- $V_{\alpha_{n,n}}$: erreur (incertitude) réalisée de l'estimation de l'angle à l'étape actuelle

L'implémentation de ce filtre est réalisée en plusieurs étapes :

1. **Initialisation des variables du filtre de Kalman ce qui comprend :**
   1.1. ω, R_n -> Bruit de processus et incertitude de mesure qui est initialisé à 10° et converti en radiant.
   1.2. B -> La matrice de transition initialisé à 5.
   1.3. $V_{\alpha_{0,0}}$ -> Incertitude de l'estimation de l'angle initiale. Il est obligatoire de l'initialiser afin d'éviter d'avoir une valeur à 0 au début.
   1.4. $\hat{\alpha}_{0,0}$ -> estimation de l'angle à l'état initiale.
   1.5. Q_n -> matrice de covariance de l'état initialisé à 5.

   Avec l'initialisation de la classe :

```
kalman = Kalman((math.pi*10/180), 5, 1.5, 0, 5)
def __init__(self, processN, kPID, V0, angle0, O):
  self.processN = processN
  self.kPID = kPID
  self.VEstPrevious = V0
  self.angleEstPrevious = angle0
  self.O = O
```

2. **Prédiction de l'état**
   2.1. Mise à jour de la variance de l'estimation de l'angle à l'étape n-1 ($V_{\alpha_{n,n-1}}$). L'incertitude de l'estimation de l'angle évolue au fil du temps en tenant compte des perturbations aléatoires du processus.

$$V_{\alpha_{n,n-1}} = V_{\alpha_{n-1,n-1}} + Q_n$$

   2.2. Prédiction du filtre de Kalman, où l'estimation précédente de l'angle, la commande et le bruit du processus sont combinés pour estimer l'angle actuel.

$$\hat{\alpha}_{n,n-1} = \hat{\alpha}_{n-1,n-1} + B \times C_{n-1} + \omega_n$$

3. **Mise à jour de l'état**

   3.1. Mesure de l'angle à l'aide du gyroscope. Dans notre classe, nous disposons d'une fonction qui permet de récupérer l'angle accumulé à l'aide du gyroscope :

```
gyroscope.getAngle()
```

   3.2. Calcule de la différence entre l'angle mesuré et l'estimation de l'angle prédit:

$$\hat{y}_n = z_n - \hat{\alpha}_{n,n-1}$$

3.3. Calcule du gain de Kalman en utilisant l'incertitude de l'estimation l'erreur de mesure . Le gain de Kalman doit être conçu pour minimiser l'erreur d'estimation globale et ainsi obtenir une meilleure estimation de l'angle actuel :

$$K_n = \frac{V_{\alpha_{n,n-1}}}{V_{\alpha_{n,n-1}} + R_n}$$

Lorsque K tend vers 1, cela signifie de l'incertitude de mesure est faible et que l'incertitude de l'estimation converge rapidement vers 0, il faudrait alors plutôt utiliser la mesure de l'angle et non l'estimation. Alors que lorsque K tend vers 0, l'incertitude de mesure est élevée et la convergence de l'incertitude est lente. Il faudrait donc utiliser l'estimation.

3.4. Mise à jour de l'estimation de l'angle en utilisant le gain de Kalman et la différence entre l'angle mesuré et l'estimation de l'angle prédit :

$$\hat{\alpha}_{n,n} = \hat{\alpha}_{n,n-1} + K_n \hat{y}_n$$

3.5. Mise à jour de la variance l'estimation de l'angle en faisant converger l'erreur vers 0 afin de réduire l'erreur d'estimation et d'améliorer la précision de l'estimation de l'état du système.

$$V_{\alpha_{n,n}} = (1 - K_n) V_{\alpha_{n,n-1}}$$

Un gain de Kalman élevé (proche de 1) réduit la variance de l'estimation, ce qui diminue l'erreur d'estimation et améliore la convergence vers la véritable valeur de l'angle.

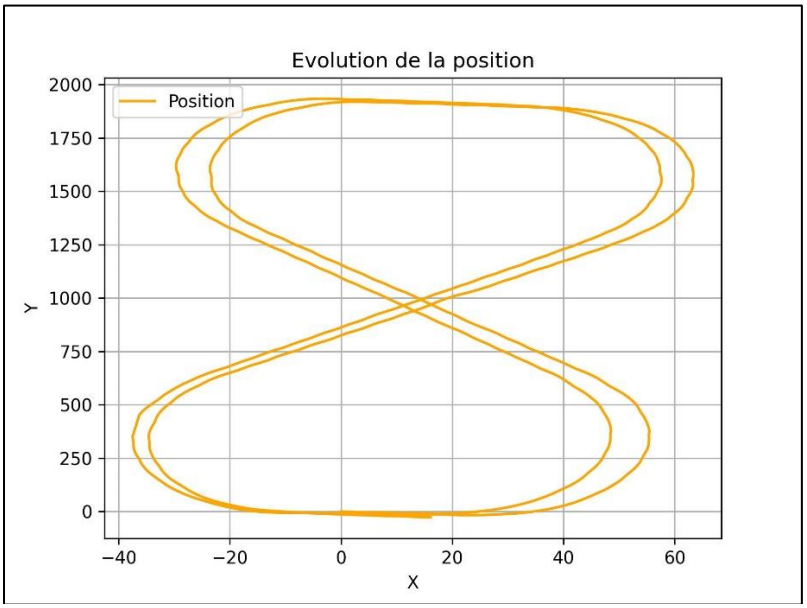
4. Utilisation de l'estimation de l'angle pour recalculer les coordonnées en fonction du filtre de Kalman

La fonction filtrer reçoit en paramètres d'entrées 2 valeurs : *l'angleN* qui correspond à l'angle obtenue par la correction précédente et *angleGyro* qui correspond à l'angle obtenue à l'aide du gyroscope. Vous pouvoir voir ci-dessous la fonction qui décrit dans l'ordre ce que nous indiqué juste au-dessus.

```
def filter(self, angleN, angleGyro):
        self.VEstPrevious = self.VEstPrevious + self.O
        self.angleEstPrevious = self.angleEstPrevious + self.kPID*angleN
+ self.processN
        Y = angleGyro - self.angleEstPrevious
       K = self.VEstPrevious/(self.VEstPrevious+self.processN)
        self.angleEstPrevious = self.angleEstPrevious + K*Y
        self.VEstPrevious = (1-K)*self.VEstPrevious
        return self.angleEstPrevious
```

Cette fonction est appelée avant le calcul de la commande de l'angle en utilisant un PID. Nous utilisons la deuxième méthode pour recalculer les coordonnées du robot (voir 1.2.) à l'aide de l'estimation de l'angle que la fonction retourne.

Résultat : Etant donné que nous avons utilisé la méthode 2 pour le calcul des coordonnées, nous pouvons remarquer une nette amélioration de la trajectoire du robot. L'estimation des coordonnées s'est améliorée autant dans les virages que en ligne droite. Toutefois, il reste tout de même des écarts entre le premier et le deuxième tour du circuit et cela est surement dû à l'incertitude de mesure que nous avons déclarée comme une constante. De plus, cela demande d'adapter certains paramètres comme le bruit du processus où la matrice de gain.



Conclusion

En conclusion, ce TP nous a donné l'opportunité de mettre en pratique nos connaissances sur le filtre de Kalman et d'approfondir notre compréhension de la prédiction et la correction des valeurs mesurées à l'aide de capteurs.

C'est un algorithme puissant pour estimer l'état d'un système en combinant des mesures et des estimations préalables afin d'obtenir une approche optimale pour estimer l'état réel en tenant compte de l'incertitude de mesure et du processus dans l'objectif d'améliorer la précision de l'estimation.

Nous avons pu observer l'importance de l'utilisation du filtre de Kalman dans les domaines de la robotique mais il est aussi utilisé dans l'aérospatial où il joue un rôle crucial dans la navigation des avions ou le suivi d'objet spatiaux par exemple.

Cependant, la mise en œuvre du filtre de Kalman reste complexe et nécessite une connaissance importante du modèle du système, des caractéristiques des mesures et des bruits du processus. Des approximations et des adaptations peuvent être nécessaires pour obtenir une estimation qui s'approche de manière précise et exacte du réel.