RAPPORT FINAL

Projet SAMI

Nom du groupe : BAMI
Membres du groupe :

Pôle Mathématiques : - Mouad Monkade
- Mostafa Abdessadek
Pôle Automatique : - Omar Guennouni
- Ilyas Tazi
-Noufal Atman Zannouti Belkacemi

Pôle Informatique : - Amine Ait Elhaj
- Moundir Abouzid

Sommaire :

I) Introduction :

Description du Problème :

Le projet SAMI est une étude scientifique portant sur le fonctionnement autonome d'un robot. Nous avons pour objectif de résoudre la problématique suivante : Comment un robot peut-il atteindre un point X à un point Y en passant par plusieurs points, tout en parcourant une distance minimale ?
Pour aborder cette problématique, nous avons divisé le travail en trois parties distinctes : mathématiques, informatique et automatique.

Objectifs du projet :

-Gérer et mener un projet de manière autonome.
-Appliquer les connaissances acquises lors des semestres 5 et 6.
-Développer une approche scientifique pluridisciplinaire à travers l'ingénierie système, tant sur le plan théorique, opérationnel que technologique.
-Contrôler la conception d'un système complexe.
-Décomposer le système en hiérarchisant les sous-systèmes.
-Gérer l'hétérogénéité des systèmes grâce à une approche intégrative.

Organisation du travail :

Nous avons commencé par prendre en main le robot LEGO EV3 afin de comprendre son fonctionnement et d'adapter le problème du voyageur de commerce à celui-ci. Notre objectif était de faire parcourir au robot 15 points prédéfinis, le jour de l'évaluation, en un minimum de temps possible.
La partie Mathématique s'est concentrée sur la théorie du problème du voyageur de commerce. En se basant sur les positions des différents points, elle devait trouver le chemin le plus court possible sans toutefois obtenir la solution exacte.
La partie Automatique était responsable du mouvement du robot lui-même sur la plateforme, incluant la vitesse, la direction et le déplacement. Elle devait également définir les lois de commandes.
La partie Informatique servait de lien entre les deux parties précédentes. Elle était chargée de programmer le code à envoyer au robot afin qu'il puisse mener à bien sa mission.
La répartition des tâches nous a permis de prendre en compte les différents aspects nécessaires à la réalisation d'un projet, tels que l'organisation, le travail d'équipe et les échanges d'informations.

Cahier des Charges :

Données d'entrée :

Ensemble de points de coordonnées (x,y) avec un maximum de 15 points
Format : fichier de points et marquage des points sur le plateau
Besoins :
A partir d'une position de départ (position courante du robot), visiter tous les points au moins une fois en minimisant la distance parcourue.
Conditions opérationnelles : plateau horizontal, scène éclairée, autonomie pour un trajet.

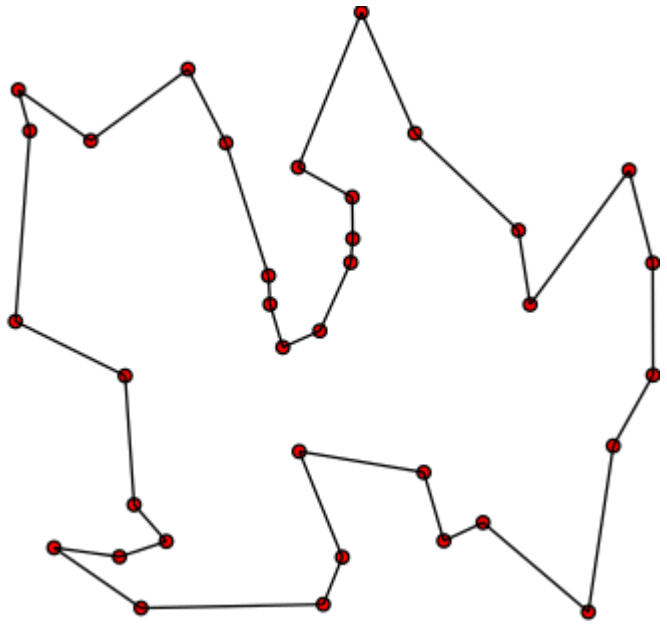
Matériels imposés :

Robot LEGO EV3 (motorisation, système de communications WIFI, calculateur, capteurs)
Système de repérage de la position et d'orientation du robot (Caméra et calculateur)
Poste utilisateur (saisie ou transfert du fichier de points)

II) Partie Mathématiques :

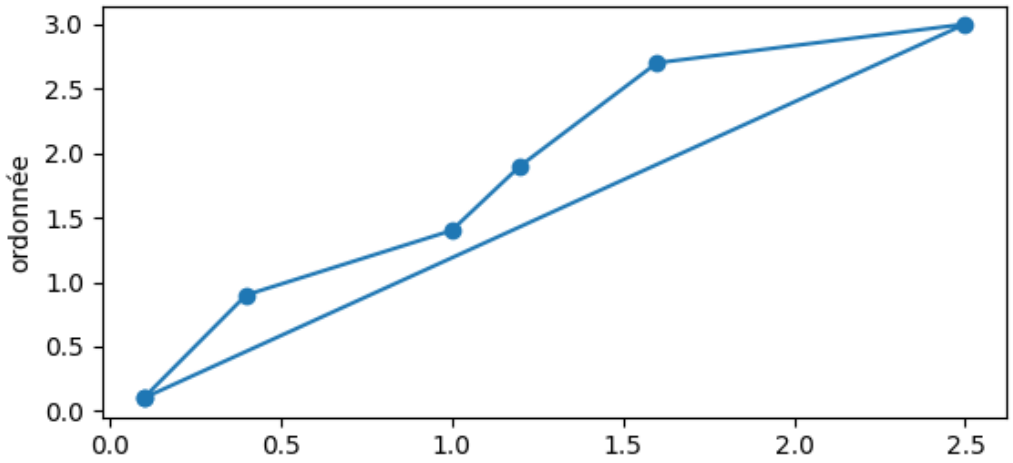
1. Positionnement du problème :

Le robot doit commencer à une position aléatoire sur une grille et ensuite parcourir des points qui ont été placés aléatoirement, de la manière la plus efficace possible. Cette situation est connue sous le nom de "problème du voyageur de commerce". Il existe une méthode optimale pour résoudre ce problème en examinant tous les chemins possibles de manière exhaustive afin de trouver le plus court. Cependant, cette méthode est extrêmement complexe avec une complexité en O(n !), où n représente le nombre de points à parcourir. Par conséquent, il est nécessaire d'utiliser une méthode alternative optimisée et moins complexe pour résoudre ce problème.
Dans notre cas, le nombre de points est relativement faible, ce qui favoriserait l'utilisation de la méthode optimale avec une analyse exhaustive, qui est simple à mettre en œuvre. Cependant, notre algorithme doit être capable de s'adapter à un nombre beaucoup plus élevé de points, nous devons donc utiliser une autre méthode.
Nos recherches nous ont conduits à deux méthodes alternatives différentes : la méthode des plus proches voisins et la méthode par insertion.



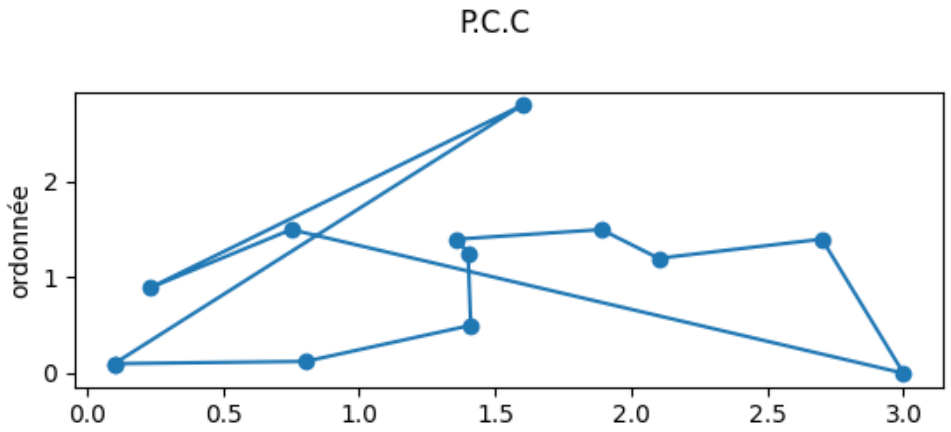
2) Méthode du plus proche voisin :

Nous avons choisi d'implémenter la méthode du plus proche voisin. Le principe est assez simple comme l'indique son nom, à partir d'un point initial, on cherche le point le plus proche dans la liste des points non explorés. Ainsi, on réitère pas à pas avec le point suivant jusqu'au dernier point. Une fois la liste des points non explorée est vide, on retourne au premier point pour fermer le graphe.



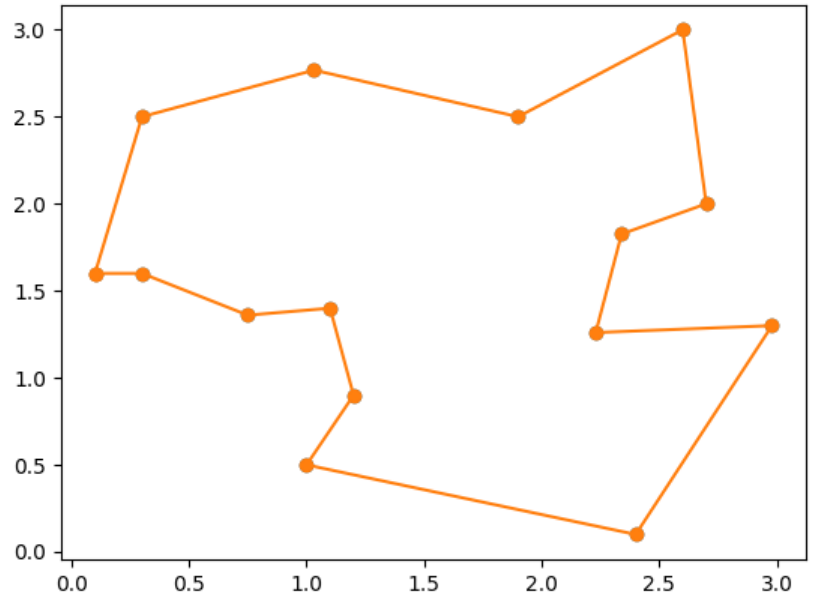
Malgré la rapidité et la complexité du programme (O(n^2)), le programme reste très dépendant au nombre des points et de leur positionnement aux uns par rapport aux

autres. Et donc ça peut aboutir à des chemins non optimaux comme le représente la figure ci-dessous.



3) Méthode par insertion :

Pour cela, nous recherchons l'enveloppe convexe de nos points, puis nous intégrons les autres points en fonction du coût minimal. Autrement dit, à chaque étape, nous insérons le sommet qui augmente le moins la longueur du cycle.

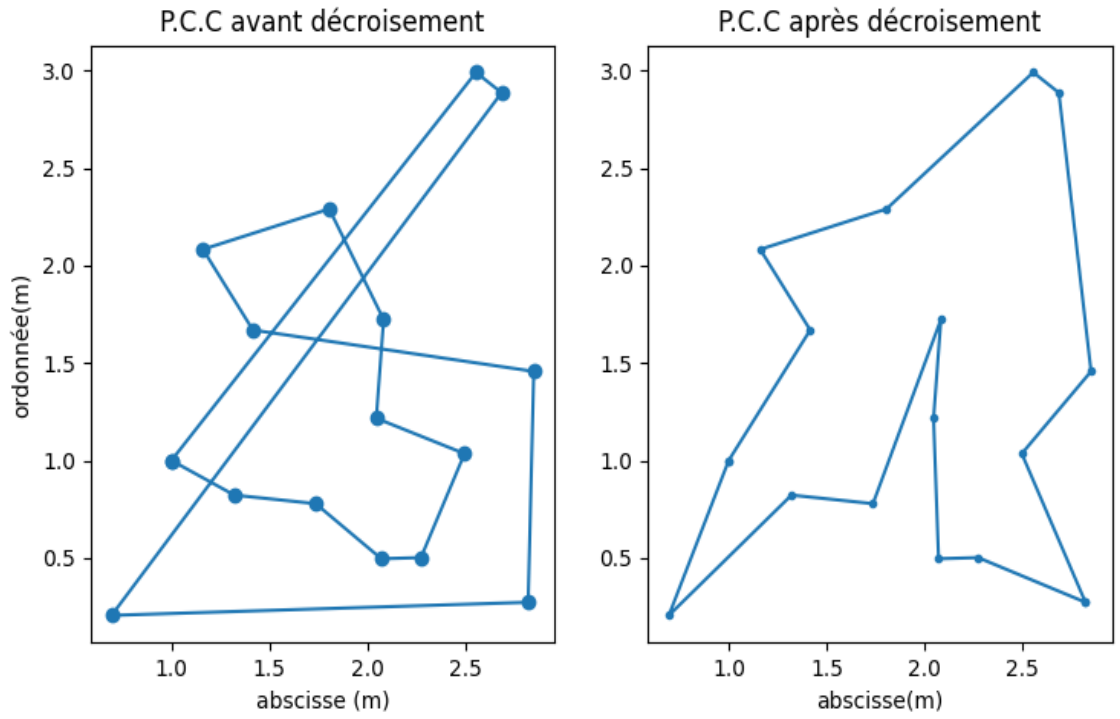


4) Méthode d'amélioration 2-opt :

Le principe de l'heuristique 2-opt est d'enlever tous les croisements existants dans le graphe. Cela revient à comparer la distance de la somme de chaque paires d'arêtes (i, i+1) et (j, j+1) non consécutives (j différents de i-1, i, i+1) avec les deux arêtes (i, j) et (i+1, j+1). Si la distance s'avère ne pas être minimale, on les permute alors tout en sachant que cela pourrait créer d'autres croisements. Et donc vérifier à nouveau, l'existence d'un croisement.

```
def decroisements(ordre):
    k=1
    while k!=0:
        k=0
        n=len(ordre)-1

        for i in range(n):
            for j in range(n):
                if (j!=i-1) and (j!=  i) and (j!=i+1):
                    if distance(ordre[i],ordre[i+1])+distance(ordre[j],ordre[j+1])>distance(ordre[i],ordre[j])
+distance(ordre[i+1],ordre[j+1]):
                        ordre[i+1],ordre[j]=ordre[j],ordre[i+1]
                        k=1
    if ordre[0]==ordre[-1]:
        return ordre
    else:
        ordre.append(ordre[0])
        return decroisements(ordre)
```

Application sur la méthode du Plus Court Chemin :



5) Conclusion (Partie Mathématiques) :

Etant donné un faible nombre de points, les deux algorithmes plus court chemin et par insertion reste les plus efficaces et rapides (O(n^2) /O(n^3)) en tenant compte de l'optimisation local (2-opt). Cependant, concernant la méthode par insertion il faut débuter par une enveloppe convexe englobant l'ensemble des points restants. Malheureusement, on était incapable d'implémenter un tel algorithme capable de retourner l'enveloppe convexe d'un ensemble de points. Ainsi, on a considéré que les deux premiers points de la liste des points. Le résultat reste totalement positif pour un nombre de points limités à 15 sans enveloppe convexe.

III) Partie Automatique :

1) Description problème

Le robot EV3 doit être programmé pour se déplacer de manière autonome le long d'un chemin préalablement calculé à l'aide de programmes mathématiques. Les dimensions du robot et sa vitesse maximale sont connues, ce qui permet de calculer les informations cinématiques du robot dans un plan donné :

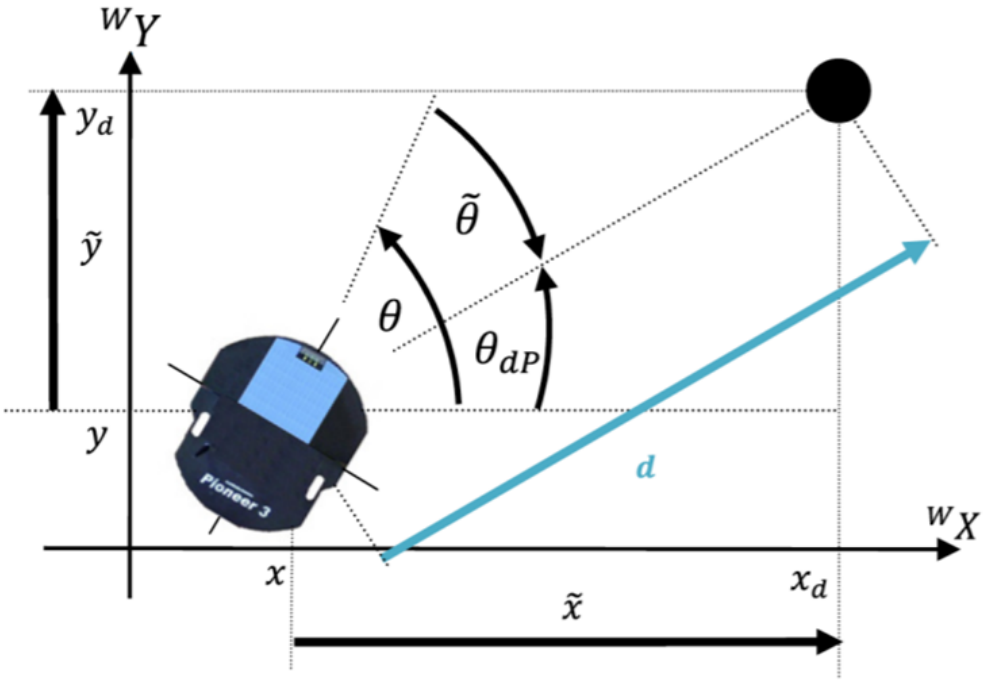


FIGURE 1 – Description du véhicule

Si l'on considère l'erreur de position et l'erreur d'orientation :

$$\tilde{x} = x_d - x$$
$$\tilde{y} = y_d - y$$
$$\tilde{\theta} = \theta_{dP} - \theta = atan(\frac{\tilde{y}}{\tilde{x}}) - \theta$$

Et la distance à l'objectif :

$$d = \sqrt{\tilde{x}^2 + \tilde{y}^2}$$

Par dérivation, on trouve que la dynamique de ces variables est déterminée par :

$$\dot{d} = -v\cos(\tilde{\theta})$$
$$\dot{\tilde{\theta}} = \frac{v}{d}\sin(\tilde{\theta}) - \omega$$

Et si l'on paramétrise le problème à l'aide des constantes k1 et k2 :

$$\dot{d} = -k_1\, d$$
$$\dot{\tilde{\theta}} = -k_2\tilde{\theta}$$

On peut alors calculer v et oméga du robot :

$$v = \frac{(k1 * d)}{cos(\tilde{\theta})} \qquad \omega = (k2 * \tilde{\theta} - (v/d) * sin(\tilde{\theta}))$$

Puis en utilisant:

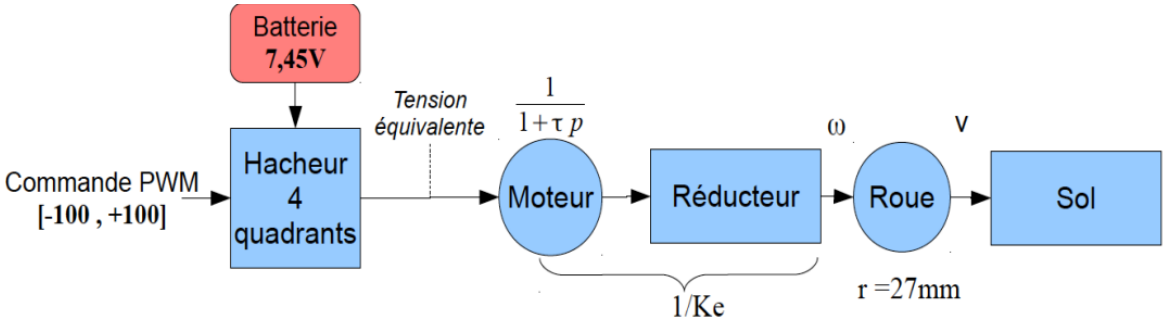
$$\begin{bmatrix} \omega \\ v \end{bmatrix} = \begin{pmatrix} \frac{-1}{\Delta} & \frac{1}{\Delta} \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix} \begin{bmatrix} v_g \\ v_d \end{bmatrix}$$

On retrouve les expressions de Vg et Vd:

$$v_g = (v - \frac{\Delta}{2} * \omega) \quad v_d = (\frac{\Delta}{2} * \omega + v)$$

L'objectif est donc de développer un programme en Python qui peut être enregistré sur le robot. Cependant, pour cela, il est nécessaire de trouver certaines constantes dans les équations précédemment identifiées. Pour ce faire, il est nécessaire de construire un modèle Simulink qui permettra de simuler les mouvements du robot et de trouver les valeurs optimales de K1 et K2 afin d'atteindre le point souhaité.

Cependant, pour obtenir une simulation réaliste du fonctionnement du robot, il est important de prendre en compte la simulation des moteurs gauche et droit du robot. Nous considérons que les deux moteurs seront simulés de la même manière. Dans un premier temps, nous savons que chaque moteur se comporte comme un système du premier ordre :



Le transfert entre PWM et v_{sol} est déterminé par:

$$v_{sol} = \frac{K_s}{1 + \tau p} PWM$$

En résumé, nous disposons de toutes les valeurs nécessaires pour construire le modèle robot et retour d'état sur Simulink, à l'exception de deux constantes.

Nous avons besoin de la constante de temps et du gain statique de la fonction de transfert du robot Ks

2) Solutions élaborées :

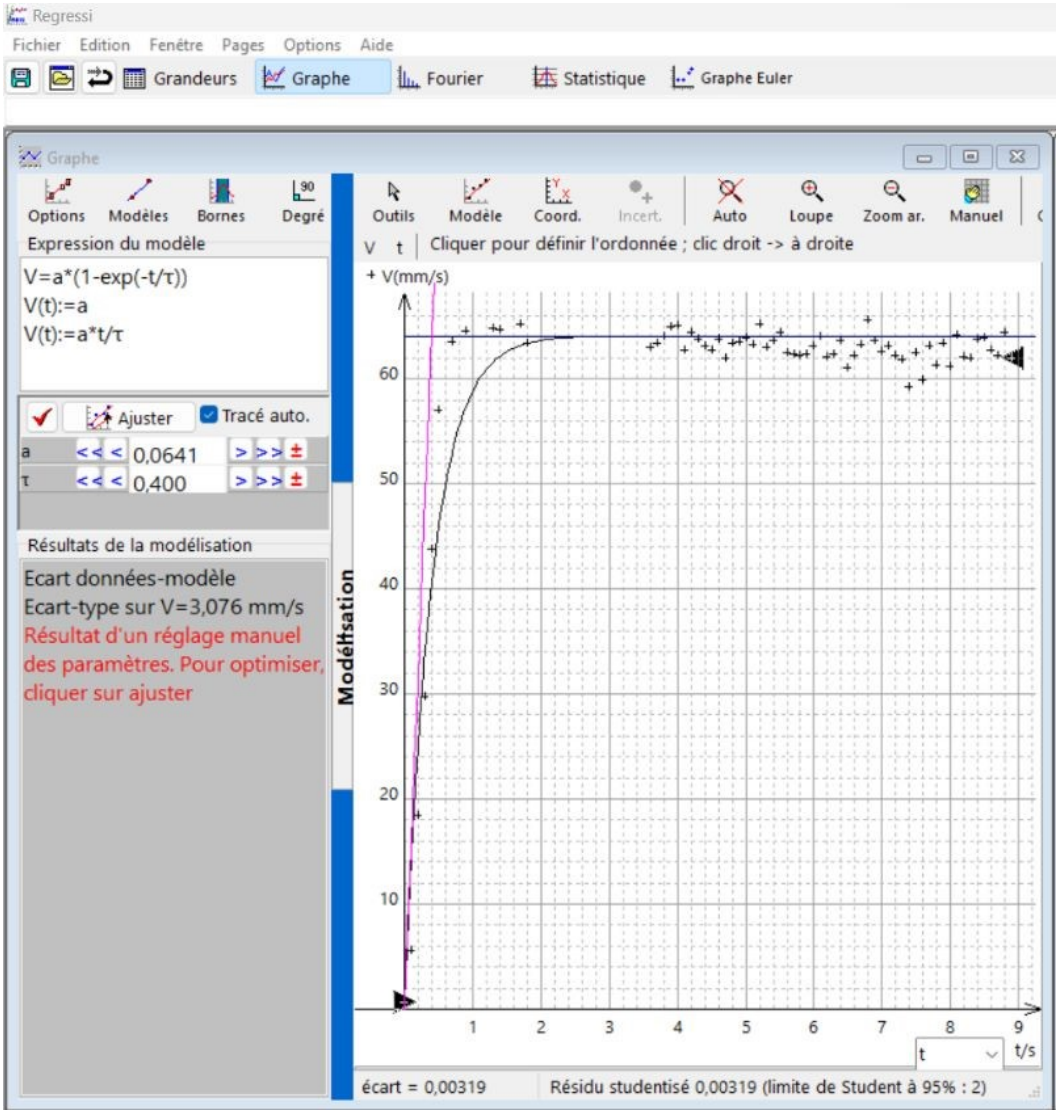
a) Identification de Ks et T :

Pour identifier le gain statique Ks ainsi que la constante du temps τ du robot, on demande au robot d'effectuer une trajectoire rectiligne à une vitesse donnée, on récupère ainsi un tableau de la position x du robot en fonction du temps .

Temps	déplacement
0	0.515691637992859
0,01	0.5638631582260133
0,02	0.612378478050232
0,03	0.662533462047577
0,04	0.7016952633857728
0,05	0.7591240406036378
0,06	0.8128058910369874
0,07	0.869361698627472
0,08	0.9073948860168458
0,09	0.966634750366211
0,1	1.0157724618911743
0,11	1.0479214191436768
0,12	1.1060385704040527
0,13	1.3819307088851929
0,14	1.422946572303772
0,15	1.4753453731536865
0,16	1.5225541591644287
0,17	1.5716782808303833
0,18	1.6082763671875
0,19	1.6533892154693604
0,2	1.7129713296890259
0,21	1.7553215026855469
0,22	1.8041578531265259
0,23	1.8565517663955688
0,24	1.9076515436172485
0,25	1.9462056159973145
0,26	1.9920870065689087
0,27	2.055901288986206
0,28	2.074219226837158
0,29	2.1277098655700684
0,3	2.1977083683013916
0,31	2.245180368423462
0,32	2.281804084777832
0,33	2.337656259536743

On trace ensuite le graphique de la vitesse en fonction du temps :

-V=f(t)



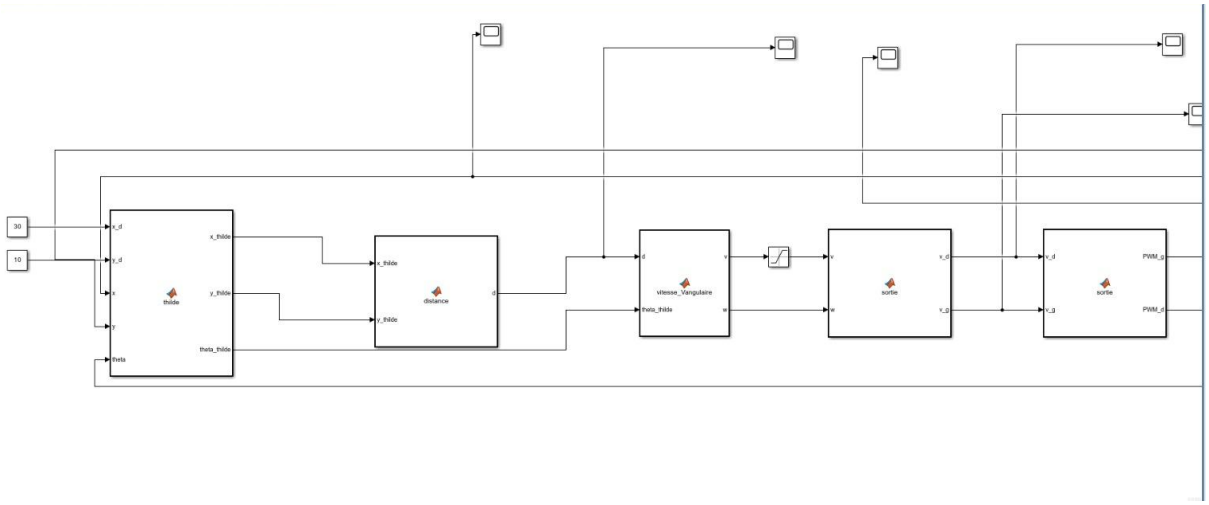
La pente obtenue représente la valeur de Ks, et le temps nécessaire pour atteindre 63% de la valeur finale de la vitesse représente la valeur de τ .

Autre manière, le logiciel Regressi nous donne directement les valeurs des constantes recherchées : On trouve donc Ks= 0.06 , et τ = 0.4 .

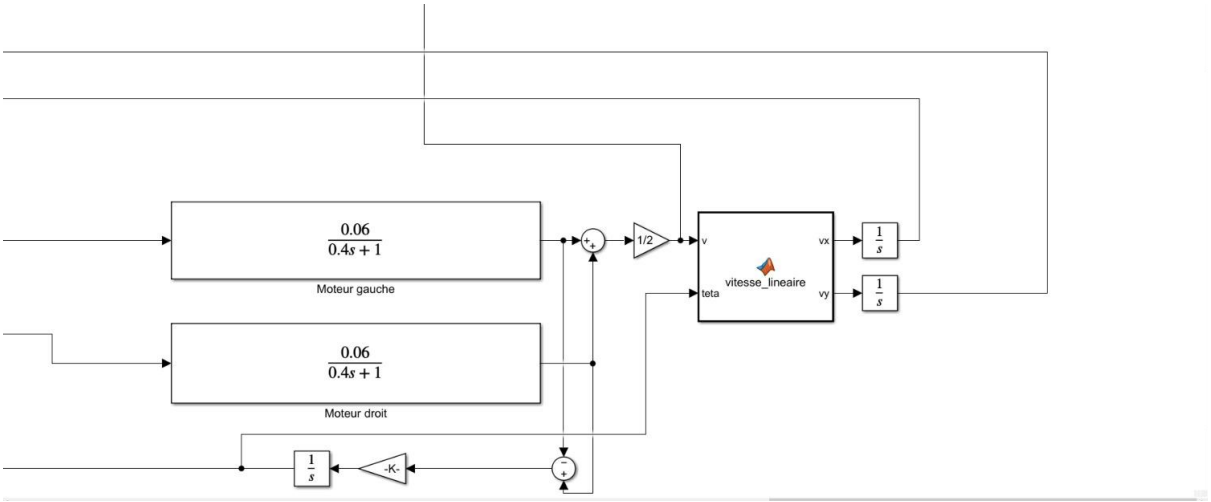
b) Simulation du modèle dynamique sur Matlab/Simulink :

Partie Contrôleur :

Le script de toutes les fonctions sont disponibles en annexes

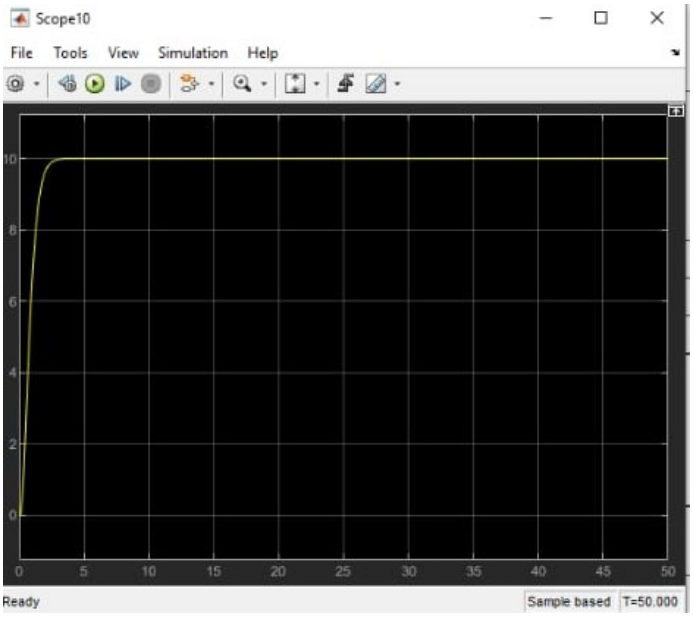


Partie Robot :



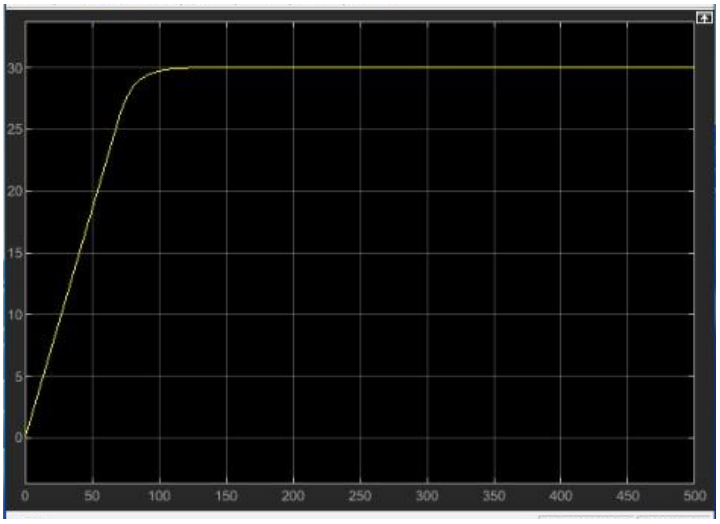
On réalise ensuite différents tests pour trouver les valeurs optimales de K1 et K2, pour déterminer la valeur de K1 en fonction des exigences, nous fixons arbitrairement la valeur de K2 , puis nous effectuons des simulations des déplacements du robot.
On obtient K2 de manière similaire.

On obtient finalement pour K1 = 0.09 , K2 = 6.5 et une entrée qui vaut 10 la réponse suivante :

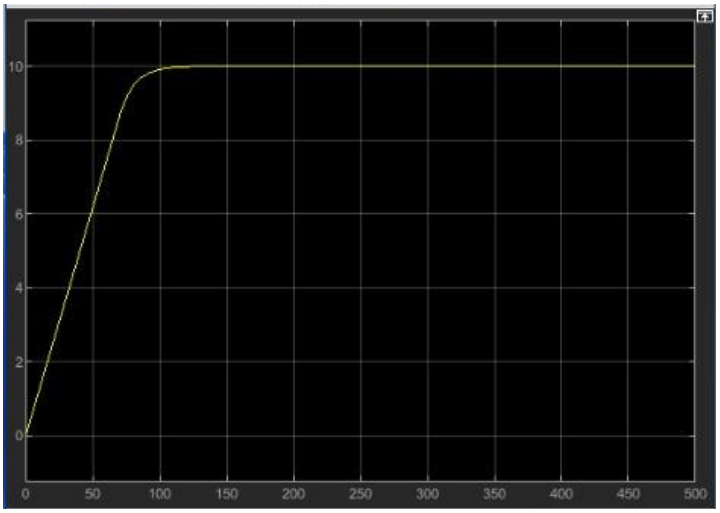


on teste pour une entrée plus grande, (x_d , y_d) = (30 , 10)

on obtient le graphe suivant :



Graphe de X



Graphe de Y

on trouve une réponse rapide, sans dépassement et sans divergence.

IV) Partie Informatique :

1) Description du problème :

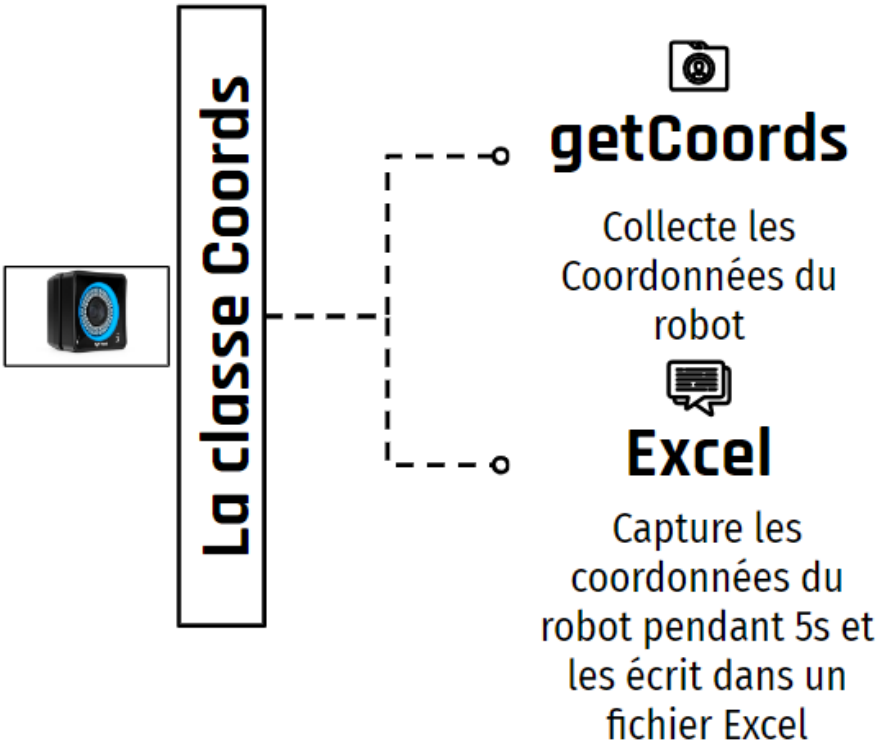
L'objectif essentiel de la partie informatique est de permettre à l'utilisateur de visualiser en temps réel la position du robot dans le circuit, et de synthétiser la solution finale en utilisant les résultats obtenus dans les parties mathématiques et automatique.
Dans un premier temps, il est nécessaire de connaître la position (X,Y,θ) de notre robot à tout moment et intégrer les coordonnées des points à atteindre par le robot selon l'ordre fournie par la partie mathématique.Puis dans un second temps, il faut également déplacer notre robot avec des vitesses précises de roue droite et roue gauche qui seront calculées et déterminées par la partie automatique.
Le cahier de charge exigeait aussi la réalisation d' une interface graphique permettant de visualiser en permanence la position du robot dans le circuit.

2) Description technique de la solution adoptée :

-Localisation du robot à l'aide des caméras du réseau Cortex :

Le robot est d'abord placé sur une plaque où sa localisation en temps réel est assurée par un système de caméra. La reconnaissance du robot et son affichage sur une interface graphique sont également gérés par le système Cortex. Chaque robot possède une signature spécifique dans ce système. Ainsi, dans notre programme, nous pouvons utiliser un système de multicast pour obtenir les données de position en temps réel de notre robot. Le programme nous renvoie donc la position (X, Y, θ) dont nous avons besoin pour guider notre robot vers les points désirés.
Afin de capturer à tout moment la position du robot, nous avons créé une classe 'Coords' qui contient deux méthodes principales:

- getCoords: Collecte les Coordonnées du robot et son orientation
- Excel : Capture la position du robot toutes les 50 ms et stocke les résultat dans un fichier Excel



-Création de l'interface graphique :

L'interface graphique offre une visualisation en temps réel du déplacement du robot. La classe présentée en annexe crée une fenêtre Tkinter qui affiche une carte avec les points à atteindre ainsi que le tracé du chemin optimal. Elle permet également de connecter le robot en entrant l'adresse IP dans le champ correspondant et en appuyant sur le bouton "Connecter". De plus, elle comporte des boutons "Avancer" et "Arrêter" qui permettent respectivement de déclencher ou d'interrompre le déplacement du robot. (voir Annexe 3)

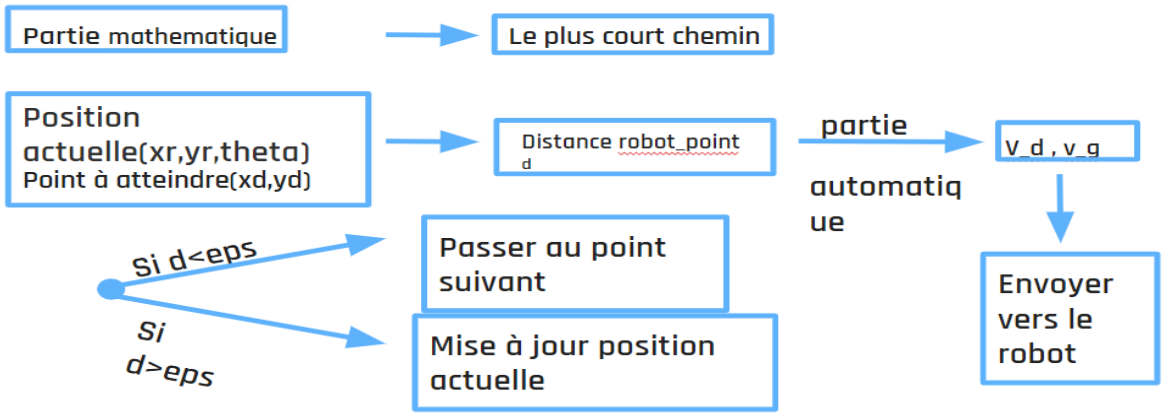
-Insertion de la partie mathématiques :

Après avoir récupéré de la partie mathématique les coordonnées des points à atteindre triées par ordre de parcours, il suffit d'entrer cette liste en argument dans la classe interface, celle-ci se charge d'afficher ces points ainsi que le chemin à parcourir.

-Insertion de la partie automatique :

Pour que le robot atteigne les points de la liste du plus court chemin, il a besoin de suivre le chemin d'une manière précise et donc il a besoin d'un correcteur qui le guide à chaque fois qu'il

s'éloigne de sa destination à cause des frottements ou bien des obstacles qui peuvent le perturbé. C'est pour cette raison qu'on fait appel à la fonction 'Commande' dans la fonction 'Move_robot', qui représente l'implémentation de la partie automatique sur python et qui prend en argument la position du robot, son angle par rapport au repère de la piste ainsi que le point désiré et renvoie les vitesses nécessaires qu'il le faut pour l'amener vers le point suivant. Ainsi, tant que le robot n'arrive pas sur le point de la liste avec une précision 'eps_d' qu'on impose, le robot continue à suivre sa trajectoire en se corrigeant à chaque instant. Le schéma suivant résume le principe de fonctionnement de la méthode Move_robot:



V) Annexes :

Annexe 1 : Partie Mathématiques

Implémentation de la méthode du plus proche voisin, fonctions nécessaires aux programmes :

```
def generer(N):
    N-=1
    L=[]
    i=0
    while  i <N:
        print("Entez les coordonnées du ",i+2,"eme point :")
        a=float(input("Donner l'abscisse du point :"))
        b=float(input("Donner l'ordonnée du point :"))
        if [a,b] in L:
            print(" Le point est deja pris ")
            print( "essayez encore :")

        else:
            if 0<= a <=3 and 0 <= b <= 3 :
                L.append([a,b])
                i+=1
            else:
                print( "le point est hors piste [3,3]")
                print( "essayez encore :")
    return L
```

Code principale :

```
def plus_court_chemin(N):

    a=float(input("entrer l'abscisse du point de départ :"))      #Abscisse du point de départ
    b=float(input("entrer l'ordonnée du point de départ :"))      #Ordonnée du point de départ
    i=1
    fib=[]
    L=generer(N)
    print(L)
    ordre=[[a,b]]                          #Liste des points ordonnés
    p=N-1
    while(L!=[]):
        fib=[]
        i=0
        while(i<p):
            val=sqrt((L[i][0]-a)**2+(L[i][1]-b)**2) #Calcul de distances les points
            fib.append(val)
            i +=1

        s=min(fib)                                  #Selection du point a distance minimal
        k=fib.index(s)
        ordre= ordre + [L[k]]
        a=L[k][0]
        b=L[k][1]
        L.remove(L[k])                             #Suppression  des points déjà visités
        p=len(L)
        if p==0:
            break
    ordre.append(ordre[0])
    return ordre
```

Optimisation locale :

```
def decroisements(ordre):

    k=1
    while k!=0:
        k=0
        n=len(ordre)-1

        for i in range(n):
            for j in range(n):
                if (j!=i-1) and (j!=  i) and (j!=i+1):
                    if distance(ordre[i],ordre[i+1])+distance(ordre[j],ordre[j+1])>distance(ordre[i],ordre[j])
+distance(ordre[i+1],ordre[j+1]):
                        ordre[i+1],ordre[j]=ordre[j],ordre[i+1]
                        k=1
    if ordre[0]==ordre[-1]:
        return ordre
    else:
        ordre.append(ordre[0])
        return decroisements(ordre)
```

Calcul de la distance du parcours :

```
#Calcul de la distance du parcours

def Distance_parcours(ordre):
    Distance = 0
    for i in range(len(ordre)-1):
        Distance+=distance(ordre[i],ordre[i+1])
    return Distance
```

Affichage du graphe :

```
#Affichage des points


N=int(input("Donner le nombre de points à génerer :"))
print("Point de départ ;")
ordre1=plus_court_chemin(N)
X1=[ordre1[k][0] for k in range(len(ordre1))]
Y1=[ordre1[i][1] for i in range(len(ordre1))]
ordre=decroisements(ordre1)
X2=[ordre[k][0] for k in range(len(ordre))]
Y2=[ordre[i][1] for i in range(len(ordre))]
print("la distance optimal du parcours est : ",Distance_parcours(ordre))
fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.set_title('P.C.C avant décroisement')
ax1.plot(X1, Y1, 'o-')
ax1.set_xlabel('abscisse (m)')
ax1.set_ylabel('ordonnée(m)')

ax2.plot(X2, Y2, '.-')
ax2.set_title('P.C.C après décroisement')
ax2.set_xlabel('abscisse(m)')

plt.show()
```

Implémentation de la méthode d’insertion :

```python
import numpy as np
import matplotlib.pyplot as plt

def distance(a, b):
    return np.sqrt((a[0]-b[0])**2 + (a[1]-b[1])**2)

def generate_coords(num_points):
    coords = []
    for i in range(num_points):
        x = float(input("Entrer l''abscisse du point {}: ".format(i+1)))
        y = float(input("Entrer l'ordonnée du point {}: ".format(i+1)))
        coords.append([x, y])
    return coords
```

```python
def plus_proche_insertion(coords):
    n = len(coords)
    dist = [[0] * n for _ in range(n)]
    for i in range(n):
        for j in range(i+1, n):
            dist[i][j] = dist[j][i] = distance(coords[i], coords[j])

    tour = [0, 1]
    for k in range(2, n):
        delta_min = np.inf
        for j in range(len(tour)):
            for i in range(n):
                if i not in tour:
                    delta = dist[tour[j]][i] + dist[i][tour[(j+1)%len(tour)]] -
dist[tour[j]][tour[(j+1)%len(tour)]]
                    if delta < delta_min:
                        delta_min = delta
                        j_star = j
                        i_star = i
        tour.insert(j_star+1, i_star)
    return tour
```

Optimisation locale :

```python
def decroisements(coords, tour):
    n = len(coords)
    amélioration = True
    while amélioration:
        amélioration = False
        for i in range(n - 2):
            for j in range(i + 2, n):
                if i != 0 or j != n - 1:
                    if distance(coords[tour[i]], coords[tour[i + 1]]) +
distance(coords[tour[j]], coords[tour[(j + 1) % n]]) > distance(coords[tour[i]],
coords[tour[j]]) + distance(coords[tour[i + 1]], coords[tour[(j + 1) % n]]):
                        tour[i + 1:j + 1] = reversed(tour[i + 1:j + 1])
                        amélioration = True
    return tour
```

Calcul de la distance du parcours :

```python
#Calcul de la distance totale parcourue dans la liste triée (Tour)
def calculer_distance_totale(coords, tour):
    n = len(tour)
    distance_totale = 0
    for i in range(n - 1):
        distance_totale += distance(coords[tour[i]], coords[tour[i + 1]])
        distance_totale += distance(coords[tour[n-1]], coords[tour[0]])
    return distance_totale
```

Affichage du graphe :

```python
# Nombre de points à générer
num_points = int(input("Donnez le nombre de points : "))
# Génération des coordonnées
coords = generate_coords(num_points)
print("Coordonnées avant le tri :", coords)

# Calcul du tour initial avec la méthode de la plus proche insertion
tour_initial = plus_proche_insertion(coords)
tour_initial.append(tour_initial[0]) # Fermer la boucle
print(tour_initial)
# Amélioration du tour en appliquant une méthode de décroisements
tour_final = decroisements(coords, tour_initial)
print(tour_final)

# Calcul de la longueur avant decroisement du graphe
distance_totale1 = calculer_distance_totale(coords, tour_initial)
print("Longueur finale du graphe :", distance_totale1)

# Calcul de la longueur après décroisement du  du graphe
distance_totale = calculer_distance_totale(coords, tour_final)
print("Longueur finale du graphe :", distance_totale)

# Affichage de la liste des coordonnées après le tri
coords_triées = [coords[i] for i in tour_final]
print("Coordonnées après le tri :", coords_triées)

# Affichage des deux graphes
x_tour_sans_dec = [coords[i][0] for i in tour_initial]
y_tour_sans_dec= [coords[i][1] for i in tour_initial]


x_tour_avec_dec= [coords[i][0] for i in tour_final]
y_tour_avec_dec= [coords[i][1] for i in tour_final]
fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.set_title('Insertion  avant décroisement')
ax1.plot(x_tour_sans_dec, y_tour_sans_dec, 'o-')
ax1.set_xlabel('abscisse (m)')
ax1.set_ylabel('ordonnée(m)')
ax2.plot(x_tour_avec_dec,y_tour_avec_dec , '.-')
ax2.set_title('Insertion après décroisement')
ax2.set_xlabel('abscisse(m)')

plt.show()
```

Annexe 2 : Partie Automatique

```
function [x_thilde,y_thilde,theta_thilde] = thilde(x_d,y_d,x,y,theta)
x_thilde = x_d - x;
y_thilde = y_d - y;
theta_thilde = atan2(y_thilde,x_thilde)-theta;
if theta_thilde > pi
    theta_thilde = theta_thilde - 2*pi;
elseif theta_thilde < -pi
    theta_thilde = theta_thilde + 2*pi;
end
```

```
function d = distance(x_thilde,y_thilde)
d = (x_thilde^2 + y_thilde^2)^(1/2);
end
```

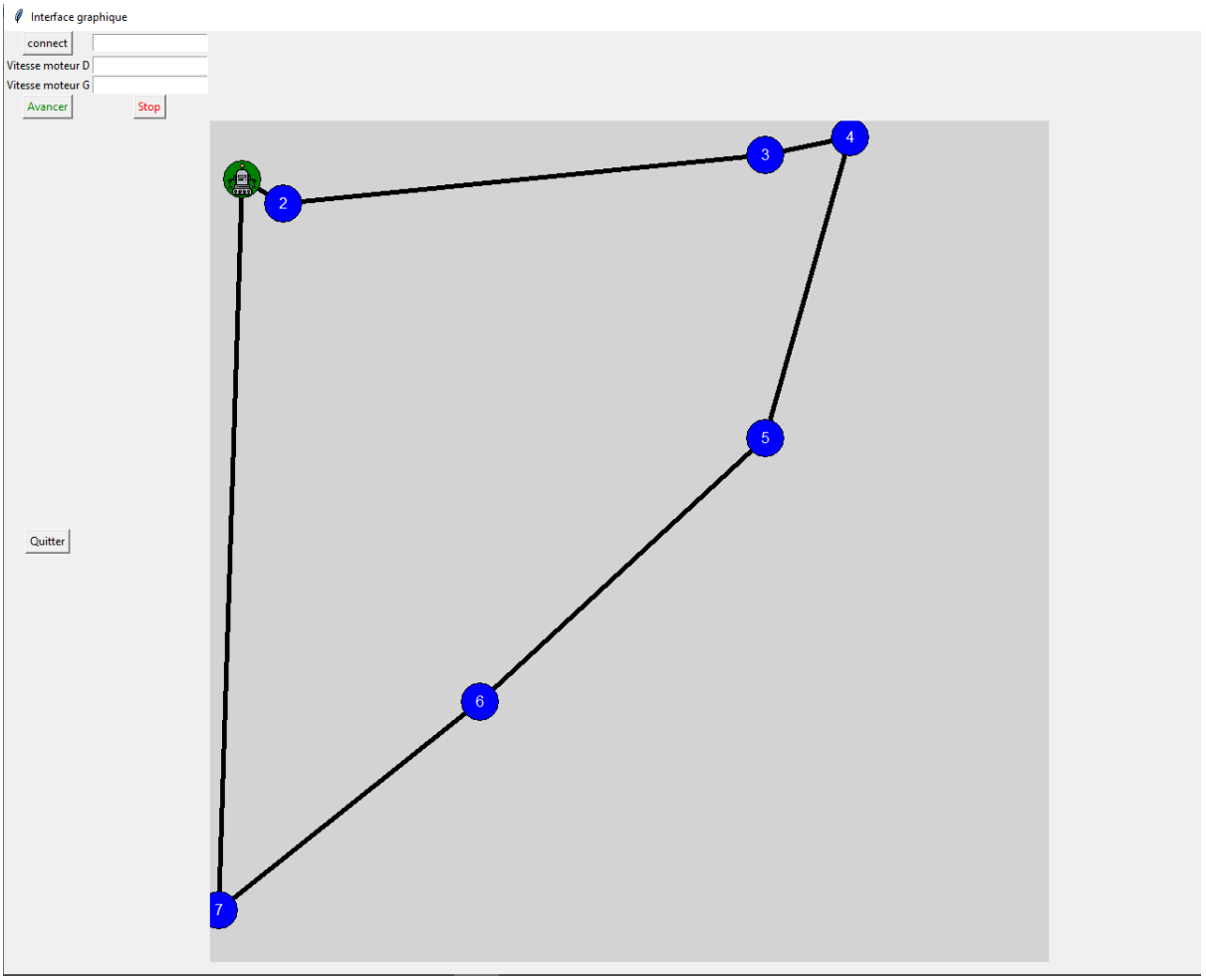
```
function [v, w] = vitesse_Vangulaire(d, theta_thilde)
k1 = 0.09;
k2 = 6.5;

v = (k1 * d)/ cos(theta_thilde);

w = (k2*theta_thilde - (v/d)* sin(theta_thilde));

end
```

```
function [v_d, v_g] = sortie(v, w)
delta = 0.1;

v_d = ((delta/2)*w + v);

v_g = (v - (delta/2)* w);

end
```

```
function [PWM_g, PWM_d] = sortie(v_d, v_g)
Ks= 0.06 ;
PWM_g = v_g / Ks;
PWM_d = v_d / Ks;
end
```

Annexe 3 : Partie Informatique

Aperçu de l’interface graphique:



Code de la classe Coords:

```python
import sys, os
import mocap_node as mcn
import time
import csv
sys.path.append(os.path.join(os.path.dirname(sys.path[0]),'optitrack-master\\src'))
class Coords():
    def __init__(self):
        self.mymcn = mcn.MocapNode('PC2')
        self.mymcn.run()
        self.mymcn.updateModelInfo()

    def getCoords(self):
        self.pos2D , self.yaw = self.mymcn.getPos2DAndYaw('Lego1')
        self.mymcn.stop()
        return self.pos2D, self.yaw

    def stopGetCoords(self):
        self.mymcn.stop()

    def __str__(self):
        return "les coordonnées (x,y) sont: " + str(self.pos2D) + " et l'angle est: " +str(self.yaw)

    def excel(self):
        with open('coordonnees10.csv', 'w', newline='') as csvfile:
            for i in range(0,100): #1000 coordonnées stockées
                c=Coords()
                pos, angle = c.getCoords()
                spamwriter = csv.writer(csvfile, delimiter=';',quotechar='|', quoting=csv.QUOTE_MINIMAL)
                spamwriter.writerow([pos[0],pos[1],angle])
                time.sleep(0.05)


#c=Coords()
#c.excel()

#print(c.getCoords())
#i 0
```

La classe Interface:

```
from tkinter import *
from myprogram import *
import time
import socket
import math
import numpy as np
from PCC import *
class TCPClient():
    def __init__(self):
        self.sock = socket.socket(
                            socket.AF_INET, socket.SOCK_STREAM)
    def connect(self, host, port):
        self.sock.connect((host, port))
    def send(self, msg):
        self.sock.send(msg.encode())

    def receive(self):
        msg_recu = self.sock.recv(1024)
        print(msg_recu.decode())
        return msg_recu
    def fermer(self):
        self.sock.close()


class InterfaceUpd():
    def __init__(self,ordre):
        self.fen = Tk()
        self.ordre = ordre
        #print(self.ordre)
        self.fen.geometry('1920x1080')
        self.fen.title('Interface graphique')
        self.initialisation()
        #constructeur de la classe pannel
        self.client = TCPClient()
        self.create_widgets()

        #constructeur de la classe interface
        self.can =Canvas(self.fen, width = 900, height = 900, bg = 'light gray')
        self.can.grid(row=4,column=4)
        self.echelle = 300
        self.dessinerCarte(self.ordre)
        self.dessinerRobotIcone()
        self.updCoordIcone()


        self.fen.mainloop()
```

```
    def dessinerCarte(self, ordre):
        n=len(ordre)
        self.dessinerLigne(0,ordre[0][0], ordre[0][1], ordre[1][0], ordre[1][1])

        for i in range(1,len(ordre)-1):
            self.dessinerLigne(i,ordre[i][0], ordre[i][1], ordre[i+1][0], ordre[i+1][1])
            self.dessinerCercle(i+1 , ordre[i][0], ordre[i][1],'blue')
        self.dessinerLigne(n,ordre[n-1][0], ordre[n-1][1], ordre[0][0], ordre[0][1])
        self.dessinerCercle(n , ordre[n-1][0], ordre[n-1][1],'blue')
        self.dessinerCercle(1 , ordre[0][0], ordre[0][1],'green')

    def dessinerCercle(self, num, x_center,y_center, color):
        r = 20
        self.can.create_oval((x_center*self.echelle-r),(y_center* self.echelle - r),(x_center* self.echelle +r),
(y_center*self.echelle + r), fill = color )
        self.can.create_text(x_center* self.echelle, y_center*self.echelle, text = str(num)  ,fill = 'white', font =
('Arial'))
        #self.can.pack()

    def dessinerLigne(self,num, x0, y0,x1, y1):
        self.can.create_line(x0*self.echelle,y0*self.echelle,x1*self.echelle,y1*self.echelle, fill ='black', width
=5)
        #self.can.pack()
```

```
    def dessinerRobotIcone(self):
        pos , yaw =Coords().getCoords()
        x , y = pos[0] , pos[1]
        #WARNING: mettre a jour le path de l'image png
        self.icone = PhotoImage(file = 'robot.png').subsample(15)
        self.icone2 =self.can.create_image(x * self.echelle, y * self.echelle, image = self.icone)
        #self.can.pack()

    def updCoordIcone(self):
        #pos , yaw =Coords().getCoords()

        #for i in range(2,100):
            pos , yaw =Coords().getCoords()
            x_old , y_old = self.can.coords(self.icone2)
            #x_old, y_old = 0.03*(i-1) , 0.03*(i-1)
            #x_new, y_new = 0.03*i , 0.03*i
            x_new, y_new = pos[0] , pos[1]
            #print(x_new , y_new)
            self.can.coords(self.icone2, x_new * self.echelle, y_new * self.echelle)
            self.can.update()
            #self.can.delete("line")
            #self.can.create_line(x_old * self.echelle, y_old * self.echelle, x_new * self.echelle, y_new *
self.echelle, fill ='sky blue', width = 4, dash = (2,2))
            time.sleep(0.2)



    #Méthodes de l'interface pannel:

    def create_widgets(self):
        self.connect = Button(self.fen, text= "connect", command=self.connect_robot)
        self.connect.grid(row=0, column=0)

        self.robot = Entry(self.fen)
        self.robot.grid(row=0, column=1)
        self.move = Button(self.fen, text="Avancer", fg="green", command= self.move_robot) # command= self.commande()
à définir
        self.move.grid(row=3, column=0)

        self.stop = Button(self.fen, text="Stop", fg="red", command=self.stop_robot)
        self.stop.grid(row=3, column=1)

        self.quit = Button(self.fen, text="Quitter",command=self.quitter)
        self.quit.grid(row=4,column=0)
```

```
def connect_robot(self):
    robot_addr = self.robot.get()
    print(robot_addr)

    self.client.connect(robot_addr,9999)
def stop_robot(self):
    msg= 'STOP'
    self.client.send(msg)
def quitter(self):
    #self.client.fermer()
    self.fen.destroy()

def move_robot(self):
    n = len(self.ordre)
    print(self.ordre)
    for i in range(1,n):
        print("point ", i)
        x_d , y_d = self.ordre[i][0] , self.ordre[i][1]
        pos , yaw =Coords().getCoords()
        #theta = yaw[0] -np.pi /2
        theta = yaw[0]
        #print(theta)
        xr , yr = pos[0] , pos[1]
        #print(xr , yr)
        #print(x_d , y_d)
        eps_d = 0.05   # en m
        #print(abs(yr - y_d))
        while(abs(xr - x_d)>eps_d or abs(yr - y_d)>eps_d):
            v_g , v_d = self.Commande(x_d,y_d,xr,yr,theta)
            msg = str(int(v_g)) + "/" + str(int(v_d))
            #print(msg)
            self.client.send(msg)
            time.sleep(0.2)
            pos , yaw =Coords().getCoords()
            xr , yr = pos[0] , pos[1]
            #theta = yaw[0] -np.pi /2
            theta = yaw[0]
            print(abs(xr - x_d) , abs(yr - y_d) )
            #print(theta)

            self.updCoordIcone()
```

```
    #Automatique:
    def Commande(self,x_d,y_d,x,y,theta):
        x_thilde = x_d - x
        y_thilde = y_d - y
        theta = theta - (math.pi /2)
        theta_thilde=math.atan2(y_thilde,x_thilde)-theta
        if theta_thilde > math.pi :
                theta_thilde = theta_thilde - 2*math.pi
        elif  theta_thilde < -math.pi :
            theta_thilde = theta_thilde + 2*math.pi
        d = (x_thilde**2 + y_thilde**2)**(1/2)
        k1=0.5
        k2=1
        k3=1
        v_max=0.32
        v = np.abs(d/(k1 + d)*v_max*math.cos(theta_thilde))
        w = ((v/d)*math.sin(theta_thilde) + k2*math.tanh(k3*theta_thilde))
        delta=0.1
        Ks= 1
        M = np.array([[-1/delta,1/delta], [1/2,1/2]])
        P=np.linalg.inv(M)
        #print(P)
        #print(np.dot(M,P))
        V=np.dot(P,[[w],[v]])
        #print(V)
        v_g = V[0][0]
        v_d = V[1][0]
        PWM_g= v_g/Ks
        PWM_d= v_d/Ks
        return 1000 * v_g, 1000 * v_d


ordre = plus_court_chemin(6)
#pos1 ,yaw =Coords().getCoords()
#ordre = [pos1] + [[2.7, pos1[1]], [2.7, 2.7] , [pos1[0], 2.7]] + [pos1]
#print(ordre)

i=InterfaceUpd(ordre)
```

```
import socketserver
from ev3dev2.motor import OUTPUT_B, OUTPUT_C, MoveDifferential, SpeedRPM, SpeedPercent, LargeMotor
from ev3dev2.wheel import EV3Tire
import time
import ast
#from myprogram import *
class Handler_TCPServer(socketserver.BaseRequestHandler):
    def executer_commande(self,msg):
        #print(msg)
        MSG = msg.decode() #convertit le message binaire en str
        print(MSG)
        if MSG == 'STOP' :
            self.stop()
        else:
            self.motor_B= LargeMotor(OUTPUT_B)
            self.motor_C= LargeMotor(OUTPUT_C)
            self.m = MoveDifferential(OUTPUT_B, OUTPUT_C, EV3Tire,126)
            v = MSG.split("/")
            #print(v)
            #self.m.on_for_distance(SpeedRPM(50),500)
            v_g = int(v[0])
            v_d = int(v[1])
            #print(v_g , v_d)
            self.motor_B.run_timed(speed_sp = v_d , time_sp = 500) # B droite
            self.motor_C.run_timed(speed_sp = v_g , time_sp = 500) # C gauche
            self.request.sendall("ACK from TCP Server".encode())
        #A compléter
        #voir codes tuto1
        #copier le fichier dans fourniture_tuto2
        #executer putty sudo python robot_server.py

    def stop(self):
        self.motor_B.stop()
        self.motor_C.stop()
    def handle(self):
        while 1:
            # self.request - TCP socket connected to the client
            data = self.request.recv(1024)
            if not data:
                break
            data.strip()
            print("{} sent:".format(self.client_address[0]))
            self.executer_commande(data)

if __name__ == "__main__":
    HOST, PORT = "100.75.155.131", 9999

    # Init the TCP server object, bind it to the localhost on 9999 port
    tcp_server = socketserver.TCPServer((HOST, PORT), Handler_TCPServer)

    # Activate the TCP server.
    # To abort the TCP server, press Ctrl-C.
    tcp_server.serve_forever()
```