Le 04/06/2024

Rapport final

SAMI Team

Membres du groupe :

- Partie automatique : Emna DEBBECH et Hedi BAYOUDH
- Partie mathématiques : Imane BOUGHELEM et Fatima ezzahra WANKIDA
- Partie informatique : Haytam El OUAGARI et Mohamed El ARCHAOUI

Table de matières

Introduction :

1. Problème posé

Le projet SAMI est un bureau d'étude qui a pour but de faire fonctionner un robot en autonomie.

On se demande comment peut-on trouver le chemin optimal qui permet à un robot de parcourir un ensemble de points donnés ?

Pour répondre à ce questionnement, on a divisé le travail en trois parties : automatique, mathématiques et informatique.

Les objectifs du projet sont les suivants :

- Aborder et gérer un projet de manière autonome.
- Application des cours vus au semestre 5 et au semestre 6.
- Développer la notion d'approche scientifique pluridisciplinaire à travers l'ingénierie système sous forme théorique, opérationnelle et technologique.
- Contrôler la conception d'un système complexe.
- Décomposer un système en hiérarchisant les sous-systèmes.
- Gérer l'hétérogénéité des systèmes grâce à l'approche intégrative.

2. Organisation du travail :

Partie mathématique : La théorie concernant le problème du voyageur de commerce. Elle devait, en fonction des positions des différents points, créer le chemin le plus court possible sans pour autant avoir la solution exacte.

Partie automatique : Elle s'occupe de modéliser le mouvement du robot (vitesse, déplacement et régulation). Cette modélisation permet de trouver les lois de commandes.

Partie informatique : Faire le lien avec les parties automatique et mathématiques. Programmer le code compilé par le robot et l'interface graphique.

3. Cahier des charges

a) 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

b) 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 éclairé, autonomie pour un trajet

c) 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)

● logiciel Matlab/Simulink

Partie Automatique :

1. Description du problème :

Le robot Ev3 est un robot automatisé qui doit pouvoir parcourir un chemin à la suite de coordonnées d'entrées. Pour ce faire, le robot est automatisé, pour que sa trajectoire soit corrigée en temps réel avec une boucle retour. La partie automatique va permettre de déterminer les lois de commandes du robot pour qu'il puisse être automatisé et qu'il puisse se corriger en temps réel.

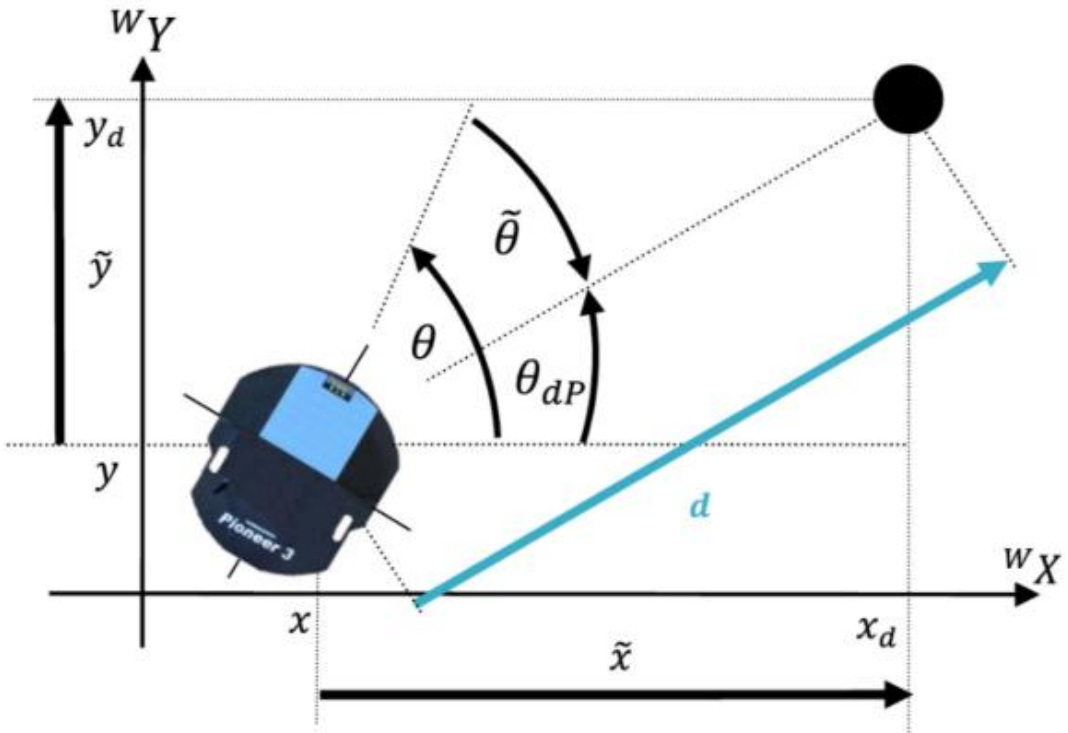


Figure 1 : Description du model

2. Partie théorie :

Les variables de notre système sont les suivantes :

$$\tilde{x} = x_d - x \qquad d = \sqrt{\tilde{x}^2 + \tilde{y}^2}$$
$$\tilde{y} = y_d - y \qquad \tilde{\theta} = \theta_{dP} - \theta = atan(\frac{\tilde{y}}{\tilde{x}}) - \theta$$

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

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

Puis on pose les variables k1 et k2 telque :

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

On obtient, alors, les expressions de la vitesse « v » et la vitesse angulaire «ω» :

En théorie, l'expression de la vitesse maximale s'obtient après identification :

$$v = \frac{k1 * d}{\cos(theta\_thilde)}$$

En pratique, on ne peut pas utiliser cette expression car notre vitesse est limitée, on a donc adopté cette expression :

$$\mathrm{v} = \frac{d}{\mathrm{k1}*\mathrm{d}} * vmax * \cos(theta\_thilde)$$

La vitesse angulaire s'obtient après l'identification des dérivées de theta thilde, on distingue deux cas:

Pour d non nul :

$$\omega=(\frac{v\times\sin(\text{theta\_thilde})}{d})+k2\times\text{theta\_thilde}$$

Sinon :

$$w=k1x\tan(\text{theta\_thilde})+k2\times\text{theta\_thilde}$$

On doit déterminer la vitesse maximale des roux du robot, elle est de 1050deg/1sec ce qui est équivalent à 0.495 m/s.

Puis grâce à ces vitesses nous obtenons les consignes PWM_g et PWM_d à envoyer aux moteurs du robot :

$$\begin{bmatrix} \omega \\ v \end{bmatrix} = \begin{pmatrix} \frac{-1}{\Delta} & \frac{1}{\Delta} \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix} \begin{bmatrix} v_g \\ v_d \end{bmatrix} \qquad \begin{aligned} v_g &= K_s\ PWM_g \\ v_d &= K_s\ PWM_d \end{aligned}$$

Finalement on obtient les consignes suivantes :

```
PWM_d = (L*w + v)/Ks
PWM_g = (v - L*w)/Ks
```

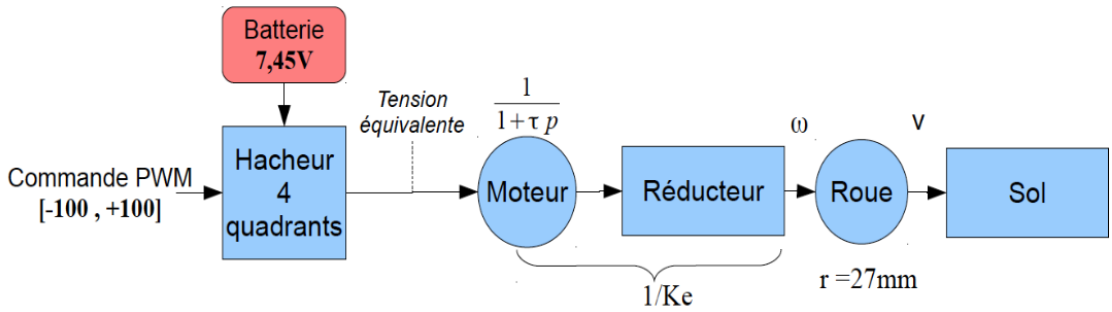
Etant données ces relations nous pouvons créer un programme Simulink qui permettrait d'obtenir une simulation de la réaction du robot. Ainsi nous pourrions faire des tests sur les constantes k1 et k2 vues précédemment dans les formules des vitesses.

Cependant, afin de simuler le fonctionnement réel du robot de manière optimale, il est essentiel de ne pas oublier de modéliser les moteurs gauche et droit du robot. En partant du principe que le robot est symétrique, nous supposerons que les deux moteurs seront modélisés de manière identique.

Dans un premier temps, nous savons que chaque motoréducteur peut être représenté par un système de transfert du premier ordre.

$$G(p)=\frac{1/Ke}{1+\tau p}$$

De plus on sait que l'ensemble qui relie la consigne au sol est de la forme suivante :



Le transfert entre PWM et Vsol est déterminé par :

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

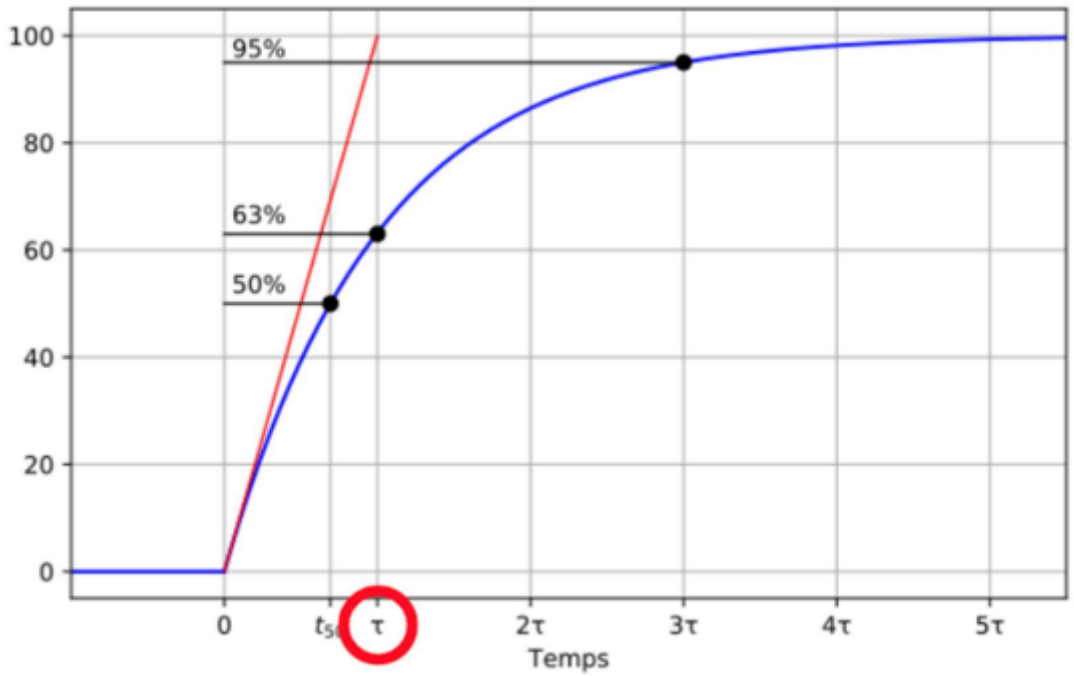
Pour obtenir les valeurs nécessaires au tracé du chemin parcouru on projette ces vitesses :

$$\dot{\theta} = \omega$$
$$\dot{x} = v_x = cos(\theta)v$$
$$\dot{y} = v_y = sin(\theta)v$$

3. <u>Mesure des constantes :</u>

Afin de mesurer les constantes du robot, il suffit de réaliser deux manipulations. Pour déterminer Ks, on demande au robot de se déplacer en

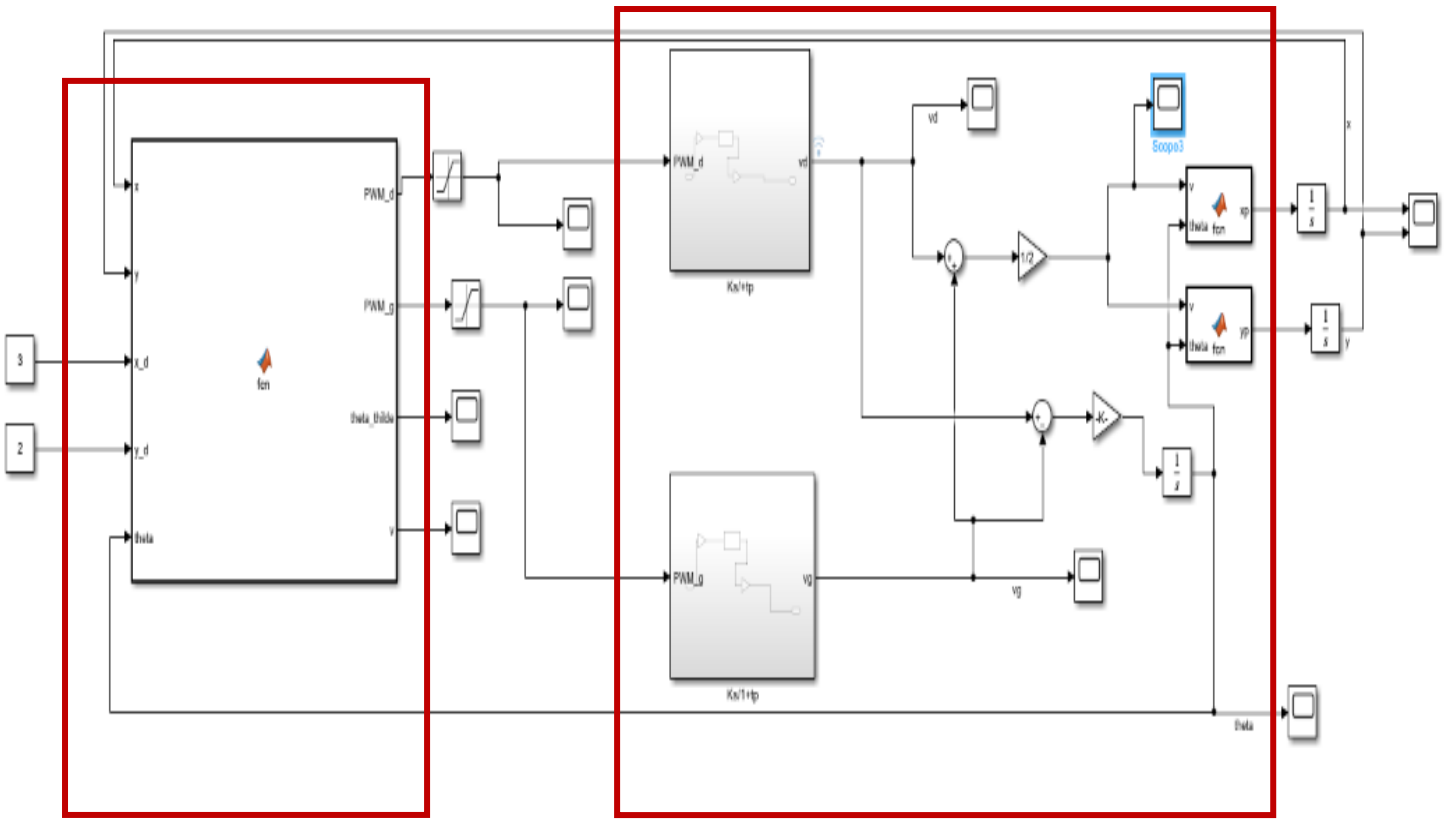
ligne droite et d'atteindre une certaine vitesse. En traçant la courbe de la vitesse en fonction du temps, la pente du graphique obtenu correspondra à Ks. Ensuite, pour mesurer τ, il faut demander au robot d'atteindre une certaine vitesse. Le temps de réponse à 63% de la vitesse finale correspondra alors à τ.



On prend τ=0.073 et Ks= 0.01

4. <u>Modèle Simulink :</u>

Pour le modèle simulink, on a construit le schéma suivant :



Les deux Blocs contrôleur et robot sont en boucle fermé

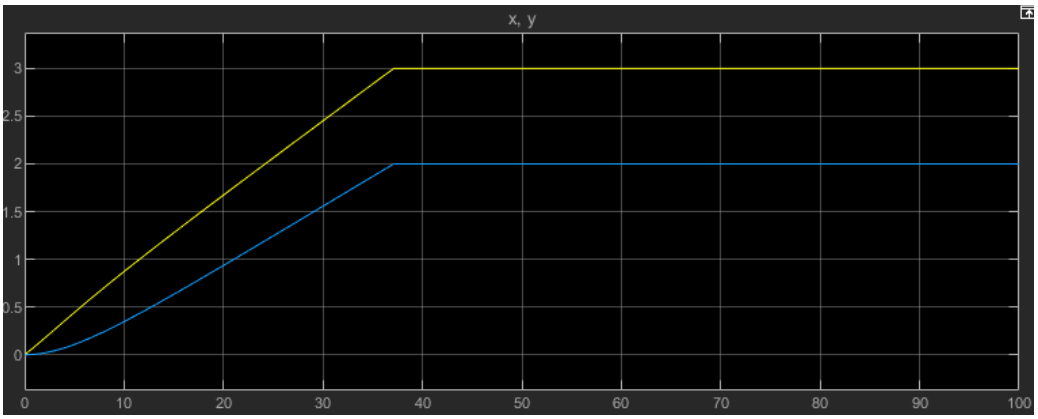
Les retours x, y et teta du bloc robot sont les entrées du contrôleur et Le retour du bloc contrôleur PMW sont les entrées du bloc Robot.

Finalement, On bloque k2 sur des valeurs aléatoires. Puis, on modifie k1 jusqu'à obtenir la valeur qui permet d'atteindre le résultat de la meilleure des façons.

K1=0.1 et k2=0.5

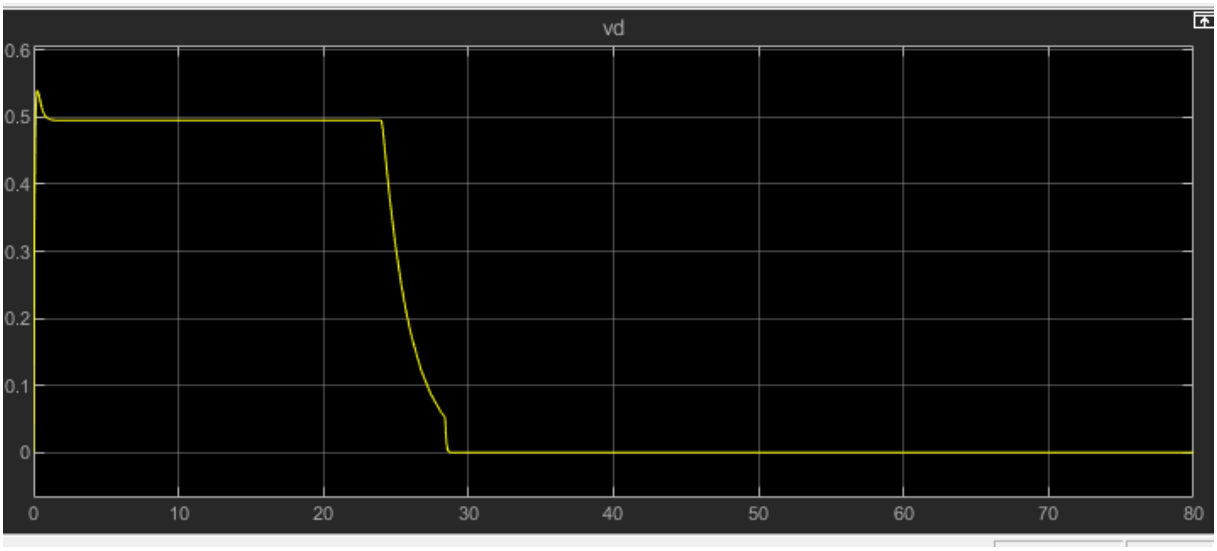
5. Conclusion :

On prend x désirée et y désirée 3 et 2. On observe que les valeurs de x et y convergent consécutivement vers x et y désirées.



Graphe de x et y en fonction du temps

Le comportement de la vitesse quant à lui est attendu, elle atteint vmax imposé pour ensuite diminuer une fois que la position correcte est obtenue.

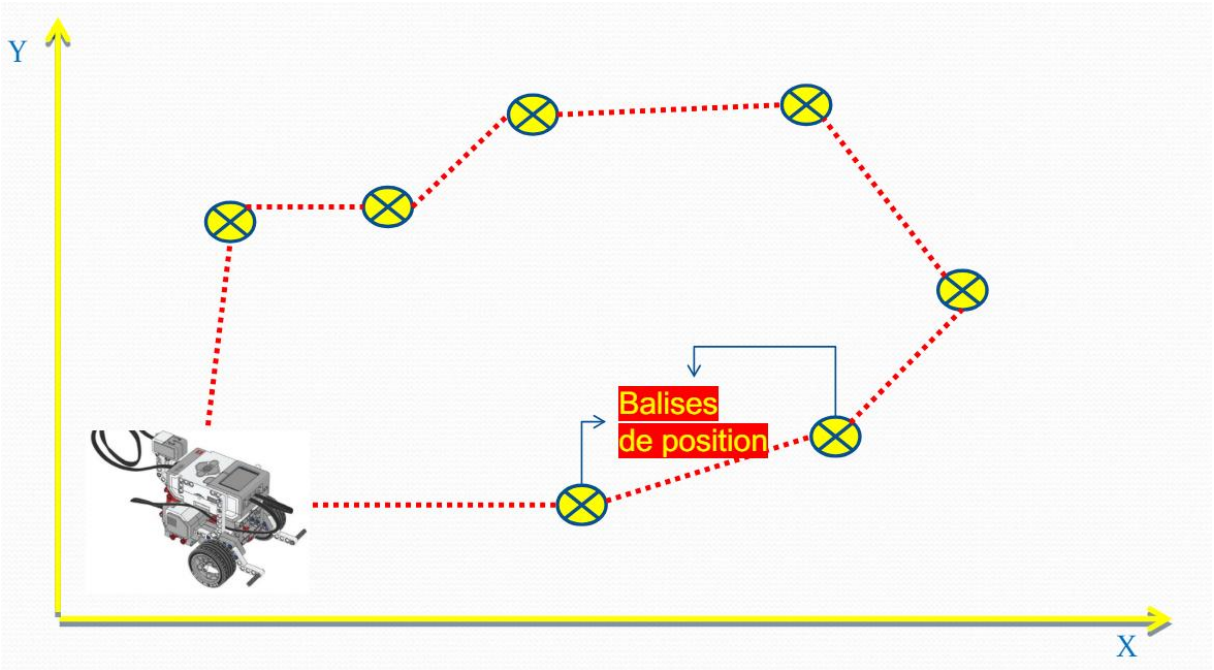


Graphe de la variation de la vitesse en fonction du temps

Partie Mathématique :

1. Position du Problème

Le problème du voyageur de commerce (TSP) est un classique de l'optimisation combinatoire et de la recherche opérationnelle. Il s'agit de trouver le chemin le plus court qu'un voyageur peut emprunter pour visiter une liste donnée de villes (ou de points) et revenir à son point de départ. Chaque ville doit être visitée une et une seule fois. Ce problème est notoirement difficile à résoudre de manière exacte, surtout lorsque le nombre de villes augmente, car le nombre de permutations possibles des villes augmente de manière exponentielle.



2. Méthode de Résolution Choisie

Pour résoudre ce problème, nous avons utilisé une méthode heuristique basée sur des permutations et des inversions de sous-segments de la liste des villes. Voici les étapes détaillées de la méthode :

a. Génération des Points :

Nous avons généré aléatoirement 15 points dans la zone spécifiée. Les points sont générés en utilisant la bibliothèque random de Python et sont affichés sur un graphique à l'aide de matplotlib.

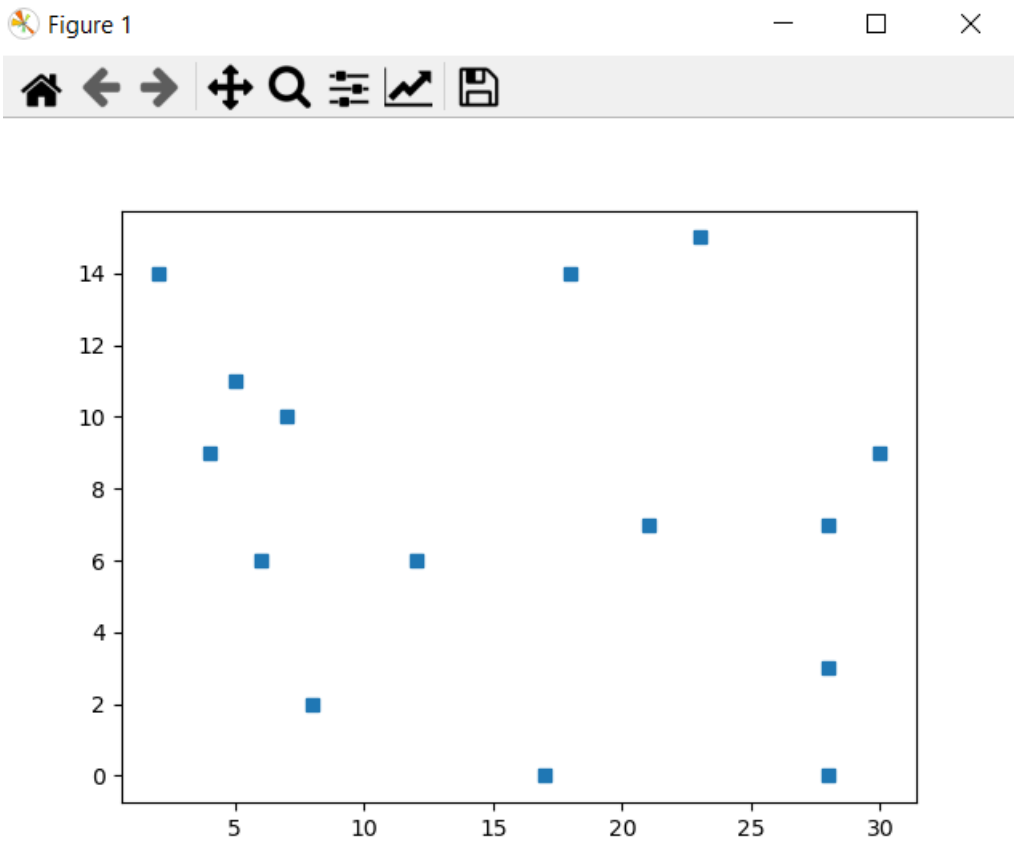


Figure1 : nuages de points aléatoires

```
import random
import matplotlib.pyplot as plt
import numpy as np
#x=[-75.0,810.0,-950.0,-550.0,190.0]
#y=[10.0 , 930.0 ,570.0,-470.0,-225.0]

for i in range(15):
    x.append(random.randint(0,12))
    y.append(random.randint(0,13))
    plt.plot(x,y,"o")
```

b. Calcul de la Longueur du Chemin :

La fonction longueur calcule la distance totale d'un parcours donné en suivant l'ordre des points spécifiés. Cette fonction utilise la distance euclidienne pour calculer la distance entre chaque paire de points consécutifs dans le parcours.

```
def longueur (x,y, ordre):
    i = ordre[-1]
    x0,y0 = x[i], y[i]
    d = 0
    for o in ordre:
        x1,y1 = x[o], y[o]
        d +=np.sqrt((x0-x1)**2 + (y0-y1)**2)
        x0,y0 = x1,y1
    return d

ordre = list(range(len(x)))
print("longueur initiale", longueur(x,y,ordre))
```

Figure2: fonction longueur

c. Optimisation de l'Ordre des Points :

La fonction permutation optimise l'ordre des points en utilisant une approche de recherche locale basée sur la permutation de sous-séquences. Elle commence par un ordre initial et cherche à améliorer la distance totale par des permutations successives de sous-séquences de points.

```
def permutation(x,y,ordre):
    d  = longueur(x,y,ordre)
    d0 = d+1
    it = 1
    while d < d0 :
        it += 1
        #print("iteration",it, "d=",d, "ordre[0]", ordre[0])
        d0 = d
        for i in range(1,len(ordre)-1) :  # on part de 1 et plus de 0, on est
sûr que le premier noeud ne bouge pas
            for j in range(i+2,len(ordre)):
                r = ordre[i:j].copy()
                r.reverse()
                ordre2 = ordre[:i] + r + ordre[j:]
                t = longueur(x,y,ordre2)
                if t < d :
                    d = t
                    ordre = ordre2
    return ordre
```

Figure 3 : fonction permutation

d. Visualisation du Parcours Optimisé :

Le parcours optimisé est visualisé avec matplotlib. Les points sont reliés dans l'ordre optimisé et le graphique montre le chemin optimal trouvé par l'algorithme.

```
ordre = permutation (x,y,list(range(len(x))))
print("longueur min", longueur(x,y,ordre))
xo = [ x[o] for o in ordre + [ordre[0]]]
yo = [ y[o] for o in ordre + [ordre[0]]]
print(x,y)
plt.plot(xo,yo, "o-")
plt.text(xo[0],yo[0],"0",color="r",weight="bold",size="x-large")
plt.text(xo[-2],yo[-2],"N-1",color="r",weight="bold",size="x-large")
plt.show()
```

e. Résultat Final :

Après optimisation, nous avons obtenu un ordre optimisé des villes qui minimise la distance totale. La fonction affiche également un graphique montrant le chemin optimal trouvé.

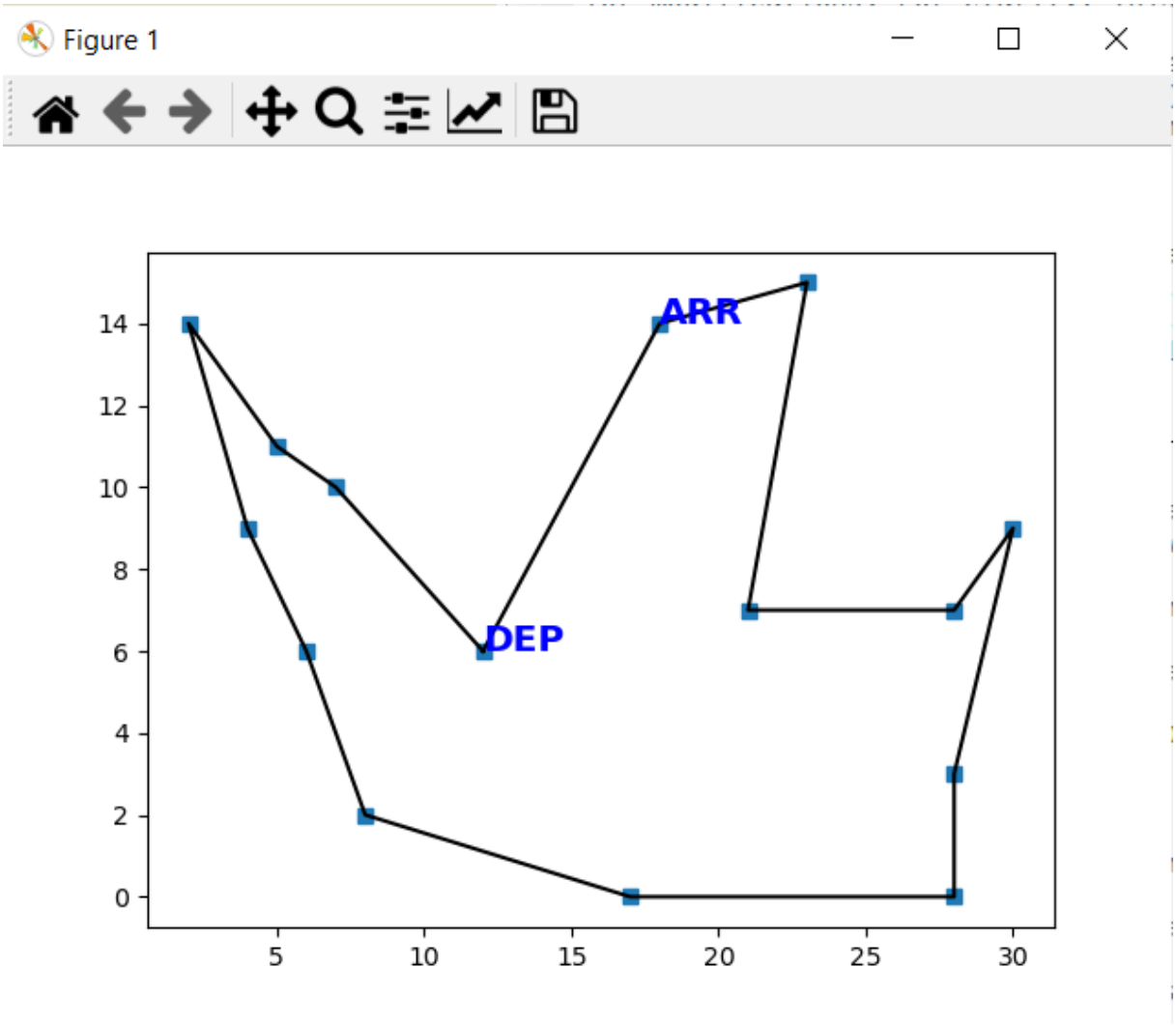


Figure 4 : Représentation d'un chemin optimal

3. Difficultés Rencontrées :

La complexité de l'algorithme est principalement déterminée par la fonction de permutation, qui optimise l'ordre des points pour minimiser la distance totale parcourue. Le calcul de la distance entre deux points prend un temps constant O(1). Dans le pire des cas, la fonction de permutation a une complexité de O(n^2), ce qui signifie que le temps de calcul augmente de manière significative avec le nombre de points. Bien que cette complexité quadratique puisse rendre

l'algorithme coûteux pour de grands ensembles de données, il reste performant et efficace pour des ensembles de points de taille modérée, permettant de trouver des solutions optimales ou quasi-optimales en un temps raisonnable.

Partie informatique

1. Description du problème :

L'objectif principal de la partie informatique consiste à 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 automatiques. Pour ce faire, nous devons d'abord déterminer la position (X, Y, θ) du robot à tout moment et intégrer les coordonnées des points à atteindre, conformément à l'ordre fourni par la partie mathématique. Ensuite, nous devons déplacer le robot avec des vitesses précises pour les roues droite et gauche, qui seront calculées et déterminées par la partie automatique. Enfin, le cahier des charges exige également la création d'une interface graphique permettant de visualiser en permanence la position du robot dans le circuit.

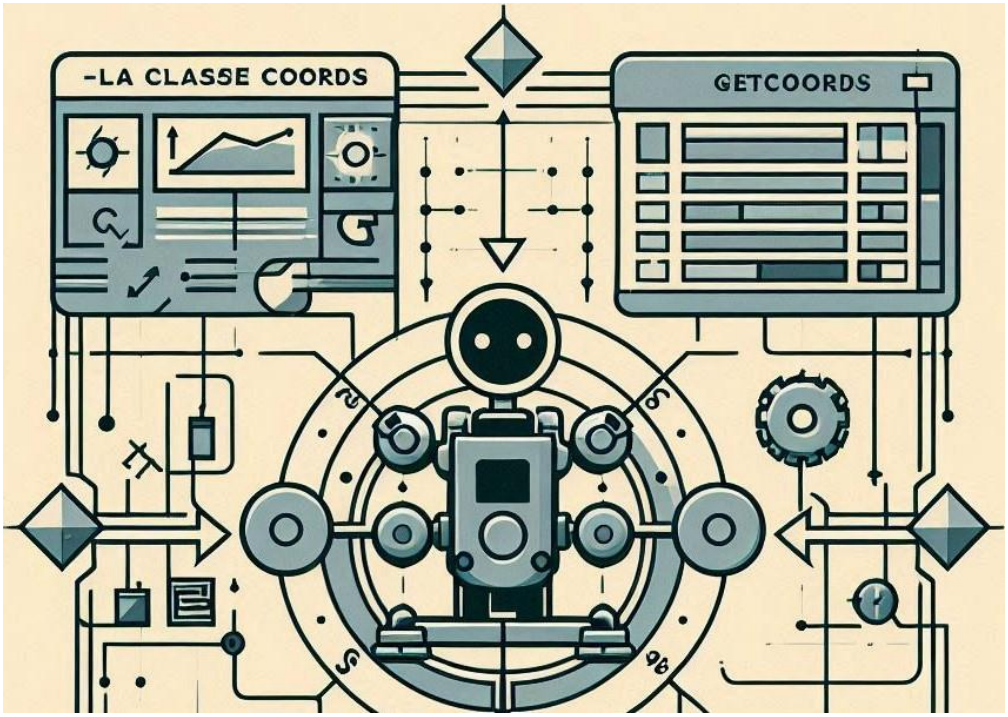
2. Description technique de la solution adoptée :

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

Le robot est d'abord installé sur une plaque, avec un système de caméra pour le localiser en temps réel. Le système Cortex assure également la reconnaissance du robot et son affichage sur une interface graphique. 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 résultats dans un fichier Excel.

- 

b) Création de l’interface graphique :

La visualisation en temps réel du déplacement du robot est offerte par l'interface graphique. 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". En outre, elle inclut des boutons "Avancer" et "Stop", qui peuvent respectivement déclencher ou stopper le déplacement du robot.

c) 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.

d) 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. Voici un schéma qui résume le principe de fonctionnement de la méthode Move_robot.

Conclusion :

Le projet n’a pas forcément atteint tous les objectifs, mais il a offert une précieuse expérience dans la gestion de projets multidisciplinaires, la modélisation de systèmes complexes et l'application de théories mathématiques à des problèmes pratiques. En suivant une démarche scientifique et méthodique, nous avons pu concevoir et contrôler un système autonome capable de remplir les tâches assignées.

Pour les travaux futurs, plusieurs pistes d'amélioration et d'approfondissement peuvent être envisagées :

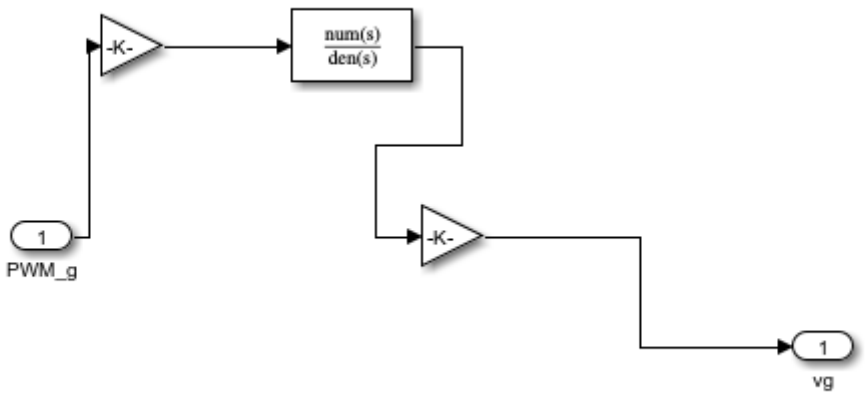
- Optimisation des algorithmes de calcul du chemin pour des solutions plus proches de l'optimum.
- Amélioration des systèmes de détection et de navigation pour une précision accrue.
- Exploration de nouvelles techniques de commande adaptative et d'intelligence artificielle pour une plus grande autonomie et adaptabilité du robot.

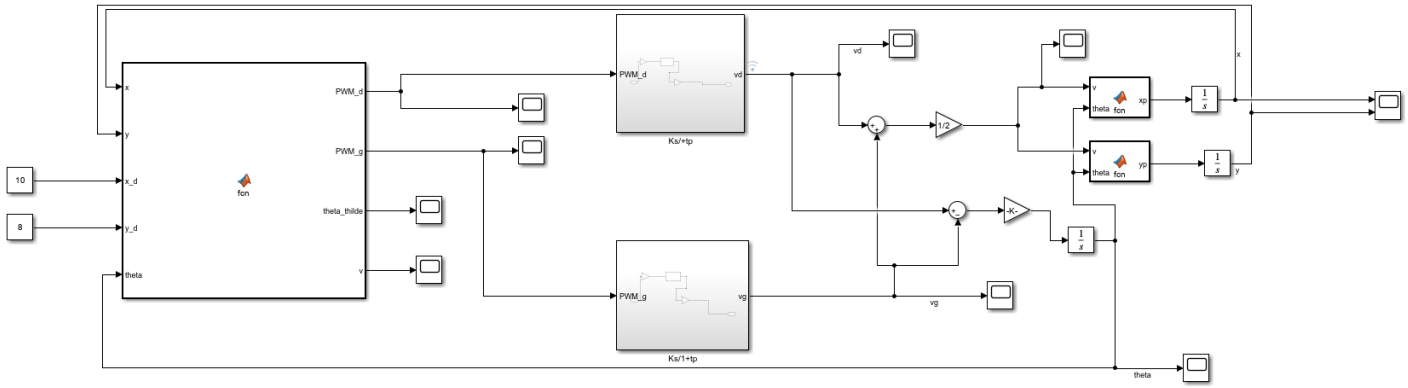
En conclusion, le projet SAMI a été un succès significatif, illustrant la puissance de l'approche pluridisciplinaire en ingénierie et ouvrant la voie à de futures innovations dans le domaine de la robotique autonome.

Annexe partie Automatique

```
function [PWM_d,PWM_g,theta_thilde,v]= fcn(x,y,x_d,y_d,theta)
k1 = 0.5 ;
k2 = 3;
L = 0.0445 ;
Ks = (7.45*0.027)/100 ;
vmax = 0.495 ;
x_tilde = x_d - x ;
y_tilde = y_d - y ;
d = sqrt(x_tilde^2+y_tilde^2);
theta_thilde = atan2(y_tilde,x_tilde)- theta ;
if theta_thilde>pi
    theta_thilde=theta_thilde-(2*pi);
elseif theta_thilde<-pi
    theta_thilde=theta_thilde+(2*pi);
end
v=min(k1*d/cos(theta_thilde),vmax);
w=(v*sin(theta_thilde)/d)+k2*theta_thilde;
if d<0.1
    w=k1*tan(theta_thilde)+k2*theta_thilde;
    v=0;
end
PWM_d = (L*w + v)/Ks;
PWM_g = (v - L*w)/Ks ;
end
```

Programme Matlab : Lois de commande




10
8
x
y
x_d
y_d
theta
fcn
PWM_d
PWM_g
theta_thilde
v
PWM_d
vd
Ka/+tp
PWM_g
vg
Ka/1+tp
vd
1/2
-K-
v
theta
fcn
xp
v
theta
fcn
yp
x
y
theta

Annexe Partie mathématique :

```
import random
import matplotlib.pyplot as plt
import numpy as np
x=[1.043,1.836,1.859,1.885,1.898,1.697,1.688,1.759,1.783]
y=[0.932, 0.844,0.798,0.793,0.771,0.914,1.36,2.215,2.354]

#for i in range(15):
    #x.append(random.randint(0,12))
    #y.append(random.randint(0,13))
    #plt.plot(x,y,"o")

def longueur (x,y, ordre):
    i = ordre[-1]
    x0,y0 = x[i], y[i]
    d = 0
    for o in ordre:
        x1,y1 = x[o], y[o]
        d +=np.sqrt((x0-x1)*2 + (y0-y1)*2)
        x0,y0 = x1,y1
    return d

ordre = list(range(len(x)))
print("longueur initiale", longueur(x,y,ordre))

def permutation(x,y,ordre):
    d  = longueur(x,y,ordre)
    d0 = d+1
    it = 1
    while d < d0 :
        it += 1
        #print("iteration",it, "d=",d, "ordre[0]", ordre[0])
        d0 = d
        for i in range(1,len(ordre)-1) :  # on part de 1 et plus de 0, on
est sûr que le premier noeud ne bouge pas
```

```
        for i in range(1,len(ordre)-1) :  # on part de 1 et plus de 0, on
est sûr que le premier noeud ne bouge pas
            for j in range(i+2,len(ordre)):
                r = ordre[i:j].copy()
                r.reverse()
                ordre2 = ordre[:i] + r + ordre[j:]
                t = longueur(x,y,ordre2)
                if t < d :
                    d = t
                    ordre = ordre2
    return ordre
def matrice(x,y,ordre):
    indice=permutation(x,y,ordre)
    L=[]
    for k in range(len(x)):
        L.append([x[indice[k]],y[indice[k]]])
    return L
print(matrice(x,y,ordre))


ordre = permutation (x,y,list(range(len(x))))
print("longueur min", longueur(x,y,ordre))
xo = [ x[o] for o in ordre + [ordre[0]]]
yo = [ y[o] for o in ordre + [ordre[0]]]
print(x,y)
plt.plot(xo,yo, "o-")
plt.text(xo[0],yo[0],"0",color="r",weight="bold",size="x-large")
plt.text(xo[-2],yo[-2],"N-1",color="r",weight="bold",size="x-large")
plt.show()
```

Annexe Partie informatique :

```
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)
        self.request.sendall("ACK from TCP Server".encode())
        valeurs=msg.decode('utf-8').split(' ')
        try:
            v1=int(valeurs[1])
            v2=int(valeurs[2])
        except:
            print("Marche pas",v1,v2)
            v1,v2=0,0
        if valeurs[0]=='MOVE':
            STUD_MM = 8
            m=LargeMotor(OUTPUT_B)
            m1=LargeMotor(OUTPUT_C)
            m.run_timed(speed_sp=v1,time_sp=200)
            m1.run_timed(speed_sp=v2,time_sp=200)

    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 = "100.75.155.137", 9999

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

    try:
        tcp_server.serve_forever()
    except:
        tcp_server.shutdown()
        tcp_server.server_close()
```

```python
from tkinter import *
#from pannel 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
        self.fen.geometry('1920x1080')
        self.fen.title('Interface graphique')
        self.initialisation()
        self.client = TCPClient()
        self.create_widgets()

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


        self.fen.mainloop()

    def initialisation(self):
        self.VD=0
        self.VG=0
        self.vd, self.vg= StringVar(), StringVar()


    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'))
```

```
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)


def dessinerRobotIcone(self):
    pos , yaw =Coords().getCoords()
    x , y = pos[0] , pos[1]

    self.icone = PhotoImage(file = 'robot.png').subsample(15)
    self.icone2 =self.can.create_image(x * self.echelle, y * self.echelle, image = self.icone)

def updCoordIcone(self):

        pos , yaw =Coords().getCoords()
        x_old , y_old = self.can.coords(self.icone2)
        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()

        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) #
    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)
```

erface2.py | robot_server.py | myprogram.py | common.py

```
    def move_robot(self):
        # vitesseA = self.vitesse_motorA.get()
        # vitesseB = self.vitesse_motorB.get()
        n = len(ordre)
        print(self.ordre)
        for i in range(1,n):
            print("point ", i)
            x_d , y_d = ordre[i][0] , ordre[i][1]
            pos , yaw =Coords().getCoords()
            theta = yaw
            #print(theta)
            xr , yr = pos[0] , pos[1]
            #print(xr , yr)
            eps_d = 0.05   # en m
            #print(abs(xr - x_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 = "MOVE "+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
                print(abs(xr - x_d) , abs(yr - y_d))
                self.updCoordIcone()
                # print(theta)
    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.master.destroy()
```