IA pour la robotique

Master SETI

Alaf do Nascimento Santos

2023/2024

Contents

1 Introduction

Dans le cadre du module IA pour la robotique, le présent travail a comme principal but l'étude de l'Odométrie et déplacements d'un robot à mouvement différentiel, modèle Thymio, simulé avec le logiciel **webots**.

Les objectifs du TP sont listés par la suite :

- Programmer l'algorithme de localisation par intégration des données odométriques
- Qualifier les données de localisation par rapport à la référence
- Développer un algorithme de suivi de trajectoire

Trois fichiers Python ont été crées : le premier pour le `contrôleur` (**tp2.py**), qui se sert de la classe `kinematicsFunctions` (**kinematics_func.py**), dont les calcules de positions et cinématique sont réalisés, et de la classe `graphWalls` (**graph_walls.py**), dédiée à la partie plotage avec matplotlib.

2 Localisation

D'abord, pour la partie localisation, nous devons savoir qu'il n'y a pas de capteurs types encodeurs dans la simulation sur webots. Par contre, nous avons accès à la fonction `getVelocity()` qui nous retourne la vitesse du moteur en radian par seconde.

Étant donné que l'écartement de la roue du Thymio est d'environ 10.8 cm et que son rayon est d'environ 2.1 cm, et à partir des valeurs de vitesses acquises, les valeurs de déplacement de chaque roue du robot ont été calculées. Après, en s'utilisant des résultats obtenus, les déplacement longitudinal et rotationnel du robot ont été aussi trouvés. La fonction Python dédiée à cette étape est présentée par la suite :

```
def get_displacements(self, left_wheel_speed, right_wheel_speed, dt):
    # Convert wheel speeds from rad/s to m/s
    right_wheel_speed_m = self.wheel_radius * right_wheel_speed
    left_wheel_speed_m = self.wheel_radius * left_wheel_speed

    # Distance travelled by the right and left wheel
    right_wheel_displacement = right_wheel_speed_m * dt
    left_wheel_displacement = left_wheel_speed_m * dt

    # Calculate longitudinal displacement
    linear_displacement = (right_wheel_displacement +
   left_wheel_displacement) / 2

    # Calculate rotational displacement
    angular_displacement = (right_wheel_displacement -
   left_wheel_displacement) / self.track_width

    return linear_displacement, angular_displacement
```

Ensuite, encore si on considère le modèle de déplacement d'un robot holonome, il est possible de calculer les nouvelles positions du robot à partir des déplacements obtenus avant. Ici on s'intéresse aux valeurs des x, y, et θ, qui donnent la position du robot (sachant que z est toujours égale à zero). Nous allons aussi sauvegarder chaque point par où le robot fait sont parcours, parce que l'on souhaite tracer sa trajectoire au fur et à mesure de ces déplacements. La fonction Python dédiée à cette étape est illustrée par la suite :

```
def get_new_pose(self, left_wheel_speed, right_wheel_speed, dt):
    # Calculate linear and angular displacements
    linear_displacement, delta_theta = self.get_displacements(
   left_wheel_speed, right_wheel_speed, dt)

    self.robot_pose['x'] += linear_displacement * np.sin(self.robot_pose
   ['theta'] + delta_theta/2)
    self.robot_pose['y'] += linear_displacement * np.cos(self.robot_pose
   ['theta'] + delta_theta/2)
    self.robot_pose['theta'] += delta_theta

    # Update traveled path
    self.x_list.append(-100*self.robot_pose['x'])
    self.y_list.append(100*self.robot_pose['y'])

    return self.robot_pose
```

Par la suite, à l'aide des points obtenus avec la fonction précédente, on veux tracer en temps réel la trajectoire suivie le robot, dans une figure qui illustre aussi les murs de l'environnement à travers la librairie matplotlib. Pour cela, les fonctions suivantes ont été développées :

```
def build_the_walls(self):
    walls = [
        {'x': [-0.25, 0.25, 0.25, 0.75, 0.75], 'y': [-0.75, -0.75, 0.25,
    0.25, 0.75]}, # external shape pt1
        {'x': [0.75, -0.25, -0.25, -0.75, -0.75, -0.25, -0.25], 'y':
   [0.75, 0.75, 0.25, 0.25, -0.25, -0.25, -0.75]}, # external shape pt2
        {'x': [0.5, 0, 0], 'y': [0.5, 0.5, -0.5]}, # center pt1
        {'x': [-0.5, 0], 'y': [0, 0]}, # center pt2
    ]

    for wall in walls:
        rotated_x = [round(math.cos(math.radians(90)) * x - math.sin(
   math.radians(90)) * y, 2) for x, y in zip(wall['x'], wall['y'])]
        rotated_y = [round(math.sin(math.radians(90)) * x + math.cos(
   math.radians(90)) * y, 2) for x, y in zip(wall['x'], wall['y'])]
        wall['x'] = rotated_x
        wall['y'] = rotated_y

    factor = 100

    for wall in walls:
        wall['x'] = [x * factor for x in wall['x']]
        wall['y'] = [y * factor for y in wall['y']]
```

```
20
21      return walls
22
23  def plot_robot(self, x, y, color='blue', destination=False):
24      walls = self.build_the_walls()
25
26      # Set aspect ratio to equal
27      self.ax.set_aspect('equal', 'box')
28
29      # Set labels and title
30      self.ax.set_xlabel('X-axis')
31      self.ax.set_ylabel('Y-axis')
32      self.ax.set_title('Robot walking through the labyrinth')
33
34      # Create animation
35      plt.ion()
36      if destination:
37          self.ax.plot(x, y, '+', color=color)
38      else:
39          self.ax.plot(x, y, color=color)
40
41      # Plot the walls
42      for wall_line in walls:
43          self.ax.add_line(Line2D(wall_line['x'], wall_line['y'], color='
    gray'))
44      self.ax.set_xlim([-self.lim, self.lim])
45      self.ax.set_ylim([-self.lim, self.lim])
46
47      plt.draw()
48      plt.pause(0.001)
```

Maintenant, nous voulons s'en servir de la fonction `getPosition()` pour tracer la trajectoire réelle du robot sur la même figure où on présente la trajectoire mesurée. En plus, nous nous intéressons aussi par tracer l'évolution de chaque composante de localisation (x,y,theta) sur un graphique ainsi que la référence de localisation. Les fonctions précedentes ont été utilisées pour cette partie du travail, il suffit juste de sauvegarder les variables d'intérêt. La Figure 1 montre un chemin fait par le robot avec les valeurs obtenus à travers la simulation, donc il s'agit de la figure de référence.

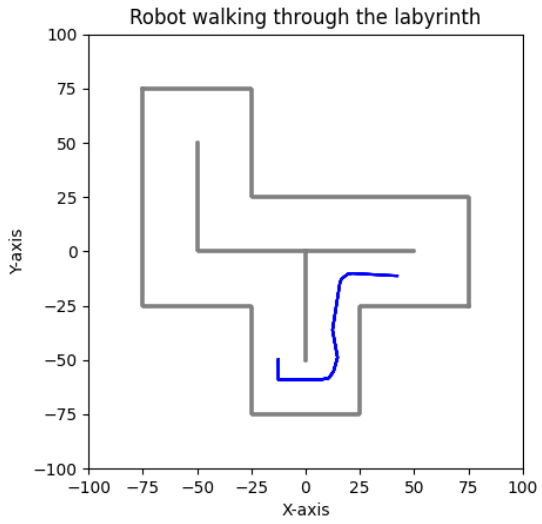


Figure 1: Chemin référence démonstratif.

La Figure 2 présente les valeurs mesurés à partir des valeurs d'odométrie. Il est possible de voir que le résultat est assez proche de celui de attendu. Par contre, quelques diferences peuvent être expliquées par le fait que l'on utilise quelques valeurs de la configuration physique du robot, comme le rayon des roues et la distance qui les sépare, ainsi que le temps nécessaire pour calculer les données.

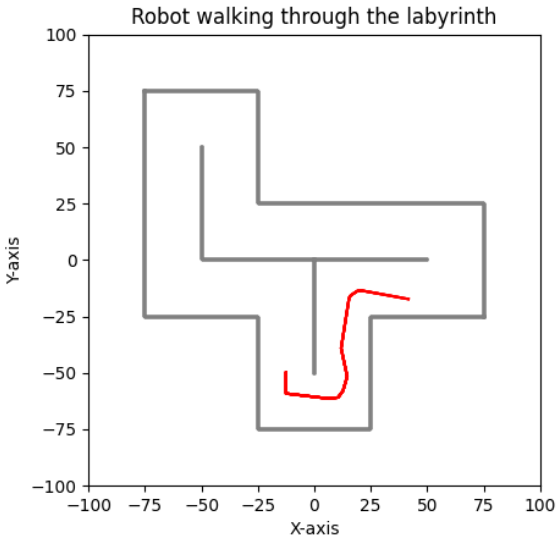
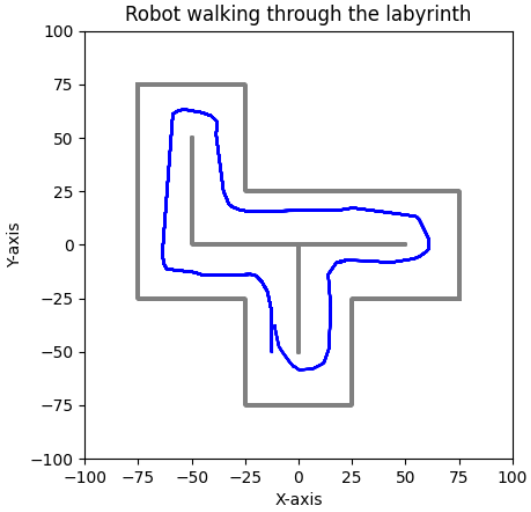
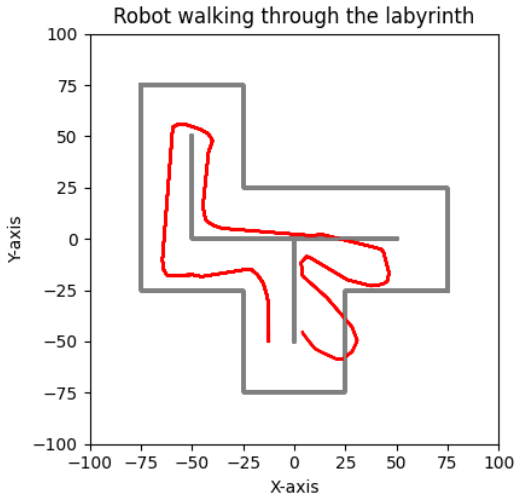


Figure 2: Chemin mesuré démonstratif.

L'erreur se propage dans le temps et lorsque vous faites le tour complet du labyrinthe, vous obtenez quelque chose comme le résultat de la figure 3. En 3a on a la référence et en 3b le chemin mesuré. L'erreur à la fin est considerablement plus important qu'au début.



(a) Chemin complet référence démonstratif.



(b) Chemin complet mesuré démonstratif.

Figure 3: Chemin complet - commençant à gauche, après montant jusqu'à finir le labyrinthe

La Figure 4 montre quelques valeurs mesurées et simulé. Il est possible de voir qu'ils sont relativement proche globalement, mais l'erreur est toujours présent. Il est important de savoir que l'y mésuré correspont au dernier valeur retourné par la fonction du simulateur.

```
Measured position:  {'x': 0.07251108624826158, 'y': -0.5993349708469438, 'theta': 0.4861111111111104}
Simulated position S:  [0.07492407942526316, -2.0067819477678772e-05, -0.6005642211729918]
Measured theta:  -27.852115041081642
Simulated theta:  -26.95155231554507
```

Figure 4: Quelques valeurs démonstratifs.

Sachant que l'idée initiale était de maintenir la compatibilité avec le simulateur fourni, Python3.9 et webots 2021b ont été utilisés. A partir de maintenant, nous ne montrerons plus d'images avec matplotlib en raison de problèmes d'incompatibilité sur mon ordinateur. Je travaillais sous Linux et le simulateur fourni était pour Windows, mais le contrôleur que j'ai développé fonctionne bien sur les deux plateformes. Il a également été testé dans la salle de classe de l'université, mais je n'ai pas pu terminer le projet là-bas car j'ai dû quitter la salle de classe (plus de détails ont été envoyés à l'adresse électronique du professeur).

3 Premiers déplacements

Dans cette partie, nous voulons explorer la manière la plus simple de se déplacer d'un point à un autre avec le thymio. Celle-là se donne par une rotation pour orienter le robot

vers le point souhaité puis d'avancer jusqu'au point. Dans cette partie, les fonctions suivantes ont été développées :

```
def rotation_matrix(self, angle):
    rotation_mat = np.array([[np.cos(angle), -np.sin(angle)], [np.sin(
    angle), np.cos(angle)]])
    return rotation_mat

def rotate(self, angle):
    # 2D rotation matrix
    rotation_matrix = self.rotation_matrix(angle)
    print("Rotation matrix: \n", rotation_matrix)

    # Position vector
    position_vector = np.array([[self.robot_pose['x']], [self.robot_pose
    ['y']]])

    # Rotating
    rotated_vector = np.dot(rotation_matrix, position_vector)

    # Updating pose
    self.robot_pose['x'] = rotated_vector[0][0]
    self.robot_pose['y'] = rotated_vector[1][0]
    self.robot_pose['theta'] += math.degrees(angle)

    def translate(self, p1, p2):
        # Finding the distance p2-p1
        distance = math.sqrt((p2[0] - p1[0])**2 + (p2[1] - p1[1])**2)

        # Translation
        self.robot_pose['x'] += distance * math.cos(math.radians(self.
    robot_pose['theta']))
        self.robot_pose['y'] += distance * math.sin(math.radians(self.
    robot_pose['theta']))
```

Donc on commence prenant les coordonnées d'un point de destination et on fait un un changement de repère contenant une rotation et une translation. La Figure 5 montre le résultat obtenu, pour un point de départ (-0.5, 0.125) et point de destination (-0.1, 0.1). À la fin, nous avons arrivés au point (-0.09, 0.13), ce qui est proche du point choisi (-0.1, 0.1) avec un erreur maximale de 3 cm pour la coordonnée y, donc on a pu vérifié la cohérence des résultats.

```
Destination:  [-0.1, 0.1]
Initial pose:  {'x': -0.5, 'y': 0.125, 'theta': 0}
Rotation matrix:
 [[ 0.99805258  0.06237829]
 [-0.06237829  0.99805258]]
Final pose:  {'x': -0.09122900347204665, 'y': 0.13094571538795133, 'theta': -3.57633437499735}
```

Figure 5: Résultats du premier déplacement

Ici il est important de donner aussi le code pour le teste, il n'a été utilisé que pour pouvoir appellé les fonctions en dehors du simulateur pour les validées.

```
from kinematics_func import kinematicsFunctions

if __name__ == "__main__":
    p1 = [-0.5, 0.125]
    p2 = [-0.1, 0.1]

    k = kinematicsFunctions(-0.5, 0.125, 0)
    print("Destination: ", p2)

    print("Initial pose: ", k.get_pose())

    angle_to_destination = k.angle_between_vectors(p1, p2)

    k.rotate(angle_to_destination)

    k.translate(p1, p2)

    print("Final pose: ", k.get_pose())
```

4 Déplacements automatisé

Pour la dernière partie du projet, afin d'utiliser les fonctions précédentes pour la navigation autonome dans le labyrinthe, la fonction suivante a été développée :

```
def trajectory_update(p1, p2, point_counter, orientation):
    k = kinematicsFunctions(p1[0], p1[1], 0)

    global arrived, need_to_rotate, remaining_distance,
   travelled_distance

    angle_to_destination, distance_to_destination = k.get_target(p1, p2,
    orientation)

    if distance_to_destination == 0:
        point_counter += 1
        return point_counter

    if debug_pt3:
        print("point: ", point_counter)

    if arrived and need_to_rotate:
        if debug_pt3:
            print("hey 1")
        need_to_rotate = True
        arrived = False
        travelled_distance = 0

    elif not arrived and not need_to_rotate : # go straight
        if debug_pt3:
            print("hey 2")
        motor_left.setVelocity(speed)
        motor_right.setVelocity(speed)
```

```
        travelled_distance += linear_displacement
        if debug_pt3:
            print("travelled_distance", travelled_distance)
            print("distance_to_destination", distance_to_destination)

        if  0.99 < abs(travelled_distance/(distance_to_destination +
    0.02)) < 1.01:
            arrived = True

    elif not arrived and need_to_rotate:
        if debug_pt3:
            print("hey 3", angle_to_destination, orientation)

        if angle_to_destination < -1.57:
            need_to_rotate = False
        elif abs(angle_to_destination - orientation) < 0.008:
            motor_left.setVelocity(-speed)
            motor_right.setVelocity(speed)
        elif abs(orientation - angle_to_destination) < 0.008:
            motor_left.setVelocity(speed)
            motor_right.setVelocity(-speed)
        elif orientation != 0:
            if 0.99 < angle_to_destination/orientation < 1.01:
                need_to_rotate = False
        elif angle_to_destination != 0:
            if 0.99 < orientation/angle_to_destination < 1.01:
                need_to_rotate = False

    else:
        arrived = True
        need_to_rotate = True
        point_counter += 1

    return point_counter
```

La fonction précédente est appelée pour la liste de points suivante, afin de réaliser le labyrinthe automatiquement, sans utiliser le clavier pour contrôler le robot. La théorie explorée dans la fonction présentée dans cette section est celle vue en classe, dans laquelle la rotation du robot est utilisée pour atteindre un point qui a été cartographié sur son référentiel.

```
self.labyrinth_trajectory = [ 'x': -0.500, 'y': 0.125},
            {'x': -0.150, 'y': 0.125},
            {'x': -0.150, 'y': 0.900},
            {'x': 0.574, 'y': 0.574},
            {'x': 0.584, 'y': 0.409},
            {'x': 0.187, 'y': 0.402},
            {'x': 0.134, 'y': -0.556},
            {'x': -0.083, 'y': -0.564},
            {'x': -0.100, 'y': -0.176},
            {'x': -0.569, 'y': -0.134},
            {'x': -0.572, 'y': 0.079},
            {'x': -0.500, 'y': 0.125} ]
```

La Figure 6 montre le résultat obtenu en essayant de franchir les points définis de manière autonome avec le code développé. Le robot commence son parcour sur le X rouge et il finit par heurter un mur à son troisième point et ne parvient pas à terminer le labyrinthe. Il est important de souligner ici que trop de données d'erreur et de limites de variation numérique peuvent entraîner des erreurs dans le logiciel développé, de sorte qu'en fin de compte, lors des tests, le système n'a pas réagi comme prévu.

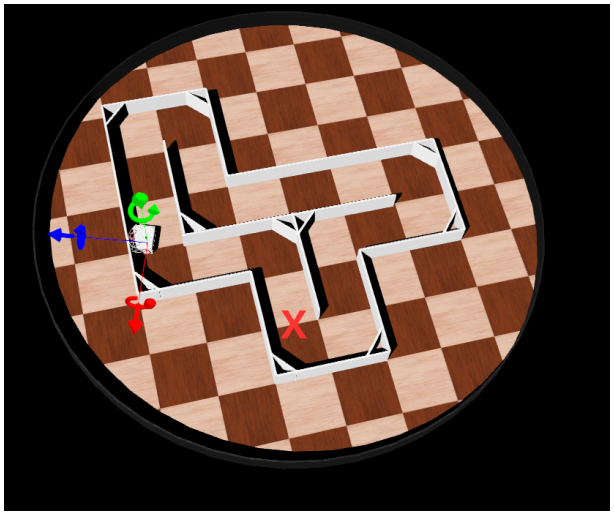


Figure 6: Résultat finale.

5 Conclusions

En conclusion de cette étude, nous avons entrepris la simulation d'un robot différentiel dans l'environnement Webots, avec pour objectif ultime la conception d'un algorithme en Python permettant de naviguer efficacement à travers un labyrinthe. Les fondements de la trigonométrie basique et de la cinématique ont été employés pour élaborer un algorithme prometteur. Cependant, malgré les efforts investis dans le développement de l'algorithme, des défis significatifs ont émergé lors des tests et simulations ultérieurs. Après le troisième point du chemin dans le labyrinthe, l'algorithme a présenté des défaillances imprévues, compromettant sa performance. Ce constat souligne la nécessité d'une réévaluation approfondie de l'approche algorithmique actuelle.

Le temps de travail dédié à ce projet a été conséquent, mettant en lumière la complexité inhérente à la navigation autonome. Il est crucial de reconnaître les limitations actuelles de l'algorithme, mais également d'envisager des pistes d'amélioration pour maximiser son efficacité. Pour améliorer l'algorithme, plusieurs avenues peuvent être explorées. Tout d'abord, une analyse approfondie des erreurs rencontrées après le troisième point du chemin est nécessaire. Des ajustements dans les paramètres de contrôle, l'optimisation des seuils de détection, ou l'intégration de techniques avancées de planification de trajectoire pourraient être envisagés.

De plus, l'utilisation de capteurs plus sophistiqués ou la mise en œuvre de techniques avancées de perception pourraient contribuer à une meilleure compréhension de

l'environnement, permettant ainsi une prise de décision plus précise. Les résultats actuels ne répondent pas entièrement à nos attentes, cette expérience a fourni des insights précieux pour le perfectionnement futur de l'algorithme de navigation. Les défis rencontrés constituent autant d'opportunités d'apprentissage et d'amélioration, et ils soulignent l'importance d'une approche itérative dans le domaine complexe de la robotique autonome.