Mines Nancy

Foundation of Robotics

Simulations V-rep sur le robot Pioneer

Auteur :
Omar KASSMI

Enseigant :
M. Henaff PATRICK

March 7, 2021

Introduction

Lorsqu'un système est représenté numériquement, sa dynamique est simulée en conséquence de quoi, la plupart des règles de fonctionnement lui sont propres et représentent de manière imparfaite celles du monde réel. Inversement, un système pour une utilisation dans le monde réel est fondé sur les lois de la physique et on ne peut alors tous les reproduire parfaitement dans le mode virtuelle.

L'objectif visé est de réaliser un travail progressif et exploratoire permettant de simuler les déplacements du robot Pioneer dans le monde virtuel, en se basant sur l'algorithme de la machine de Braintenberg.

Afin de répondre à cet objectif, Je vais dans un premier temps exposer quelques uns des fondements théoriques utiles pour la suite. Ensuite, je m'attacherai à exposer les choix que nous avons fait en terme d'implémentation et les raisons qui ont motivé ces choix.

1 Fondements théoriques

1.1 Véhicules de Brainterberg

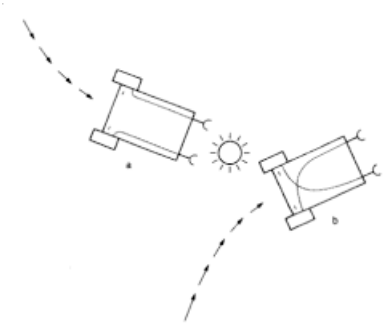

Figure 1: **Machine de Braitenberg**

Les véhicules de Braitenberg permettent de produire des comportements complexes à partir des règles simples. Possédants des capteurs de luminosité, les machines de Braitenberg sont capables de percevoir des informations simples, qui vont être traitées pour produire des mouvements.
Elles fonctionnent sur ce que l'on pourrait appeler une boucle de comportement fermée : un ou plusieurs moteurs peuvent être activé, si une intensité est perçue par les capteurs de luminosité. A cet effet, s'en résulte un déplacement de la machine et une modification de perception de l'environnement. Ce principe peut être fait en boucle continue et fermée selon le schéma suivant :

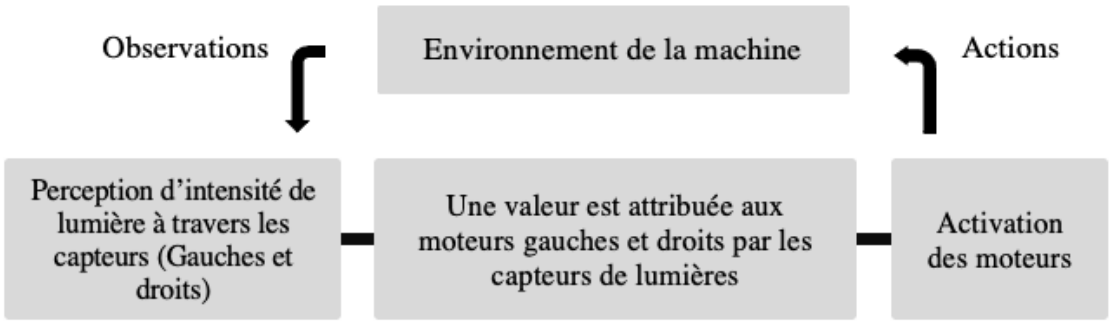

Figure 2: **Boucle de fonctionnement d'une machine de Braitenberg**

La façon dont la machine a été conçue, a permis à Braitenberg de définir plusieurs comportements différents. Par exemple, la liaison du capteur de luminosité droit avec le moteur gauche et du capteur de luminosité gauche avec le moteur droit, produit un mouvement vers la source de la lumière, autrement dit, le véhicule va suivre la lumière.

Ainsi, en considérant les différentes manières de relier les capteurs et les moteurs entre eux, et en changeant la façon dont la lumière agisse (Excitation / Inhibition), on peut identifier plus de huit comportements de la machine.

Plus généralement, ces véhicules réalisent simplement une remontée ou une descente de gradient sur l'intensité de la lumière, ils correspondent donc à un simple contrôleur propotionnel en automatique et sont donc relativement sujets à des oscillations dans le comportement du robot.

1.2 Algorithme de Braintenberg : Evitement d'obstacle

L'evitement d'obstacles est un comportement de base présent dans quasiment tous les robots mobiles. Il est indispensable pour permettre au robot de fonctionner dans un environnement dynamique et pour gérer les écarts entre le modèle interne et le monde réelle.
Une nouvelle méthode dans les stratégies de navigation réactive en robotique mobile, est inspirée aujourd'hui du principe de machine de Braitenberg. Mais, plusieurs tests ont montré l'incapacité d'adaptation du robot face à différentes situations. Par exemple, en placant le robot proche d'un obstacle, le robot est souvent incapable de l'éviter. Un comportement souhaitable serait donc que la vitesse soit négative lorsque le robot se trouve à une distance faible d'un obstacle. D'où l'idée de l'utilisation d'une fonction de transfert pour permettre la transformation du signal reçu par le capteur en une vitesse pour le moteur. Sauf que l'utilisation de telle fonction nécessite plusieurs capteurs à la place de deux seulement. Ce qui nous ramène un autre problème : comment exploiter et gérer plusieurs capteurs tout en gardant le même principe de Braitenberg ?
Une solution possible serait de remplacer le capteur gauche du robot par la moyenne de plusieurs capteurs placés dans le côté gauche et réciproquement pour le côté droit. Il faut bien noter que cette moyenne doit être pondérée,

pour tenir compte de l'importance de chaque capteur.
Notons ϕ la fonction de transfert, $(x_{l,1}, x_{l,2}, ..., x_{l,n})$ et $(x_{r,1}, x_{r,2}, ..., x_{r,n})$ les capteurs gauches et droits respectivement, v_r la vitesse du moteur gauche et v_l la vitesse du moteur droit, alors on a :

$$v_r = \phi(\sum_{i=1}^{n} \omega_i x_{r,i})$$

$$v_l = \phi(\sum_{i=1}^{n} \omega_i x_{l,i})$$

Avec ω_i représentent le poids de chaque capteur dans le calcul de la vitesse du moteur correspondant.
Remarquons qu'en connectant tous les capteurs à tous les moteurs, on obtient la formulation d'un réseau de neurone à une couche.
Le schéma suivant résume ce principe :

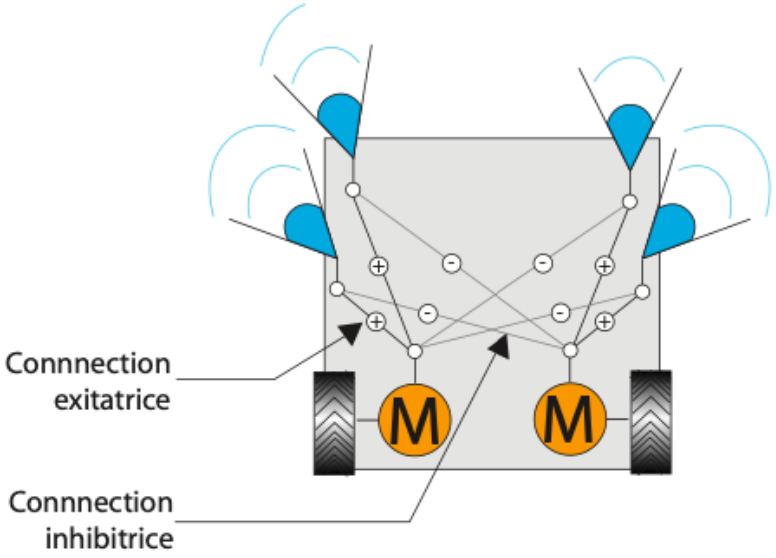

Figure 3: **Véhicule inspiré du véhicule de Braitenberg, capable d'éviter les obtacles**

1.3 Navigation vers un but

Il existe plusieurs stratégies permettants la navigation d'un robot vers un but, La plus simple étant la méthode proposée par Braitenberg, qui consiste à considérer la cible comme une source de lumière. Il suffira donc de munir

le robot des capteurs de luminosité et de le paramétrer pour qu'il soit attiré par la lumière (voir l'exemple expliqué à la fin de la partie 1.1). Malgré la simplicité de la méthode, il est difficile de la réaliser dans application réelle, vu que les capteurs ne peuvent pas détecter de la lumière au delà d'une certaine distance.

Une autre approche consiste à considérer à chaque instant **t** la position du robot, la position de la cible et de calculer à chaque instant **t** la distance séparant le robot de son but. Ceci nécessitera l'utilisation de quelques notions de la mécanique pour déterminer les équations mathématiques.

Considérons le robot Pioneer dans le plan (O, x, y) et une cible "*Goal*".

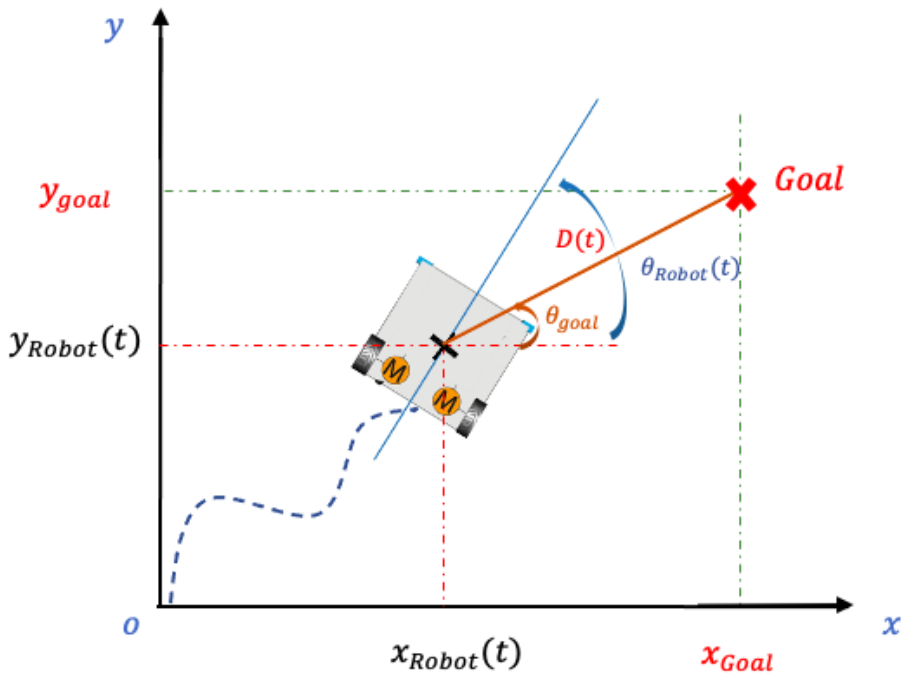

Figure 4: **Modélisation de la trajectoire du Robot Pioneer dans le plan** (O, x, y)

Connaissant les positions du robot à l'instant **t**, la position de la cible, on peut calculer $D(t)$ la distance séparant le robot du but comme :

$$D(t) = \sqrt{(x_{Goal} - x_{Robot}(t))^2 + (y_{Goal} - y_{Robot}(t))^2}$$

et à partir de la vitesse linéaire et angulaire du robot, on peut en déduire les vitesses des moteurs gauches et droits : notons V_{Robot} et ω_{Robot} la vitesse linéaire et angulaire respectivement du robot, r le rayon de la roue et R la

distance entre les roues. Alors sachant que :

$$\omega_{Robot} = k_1(v_r - v_l)$$

et

$$V_{Robot} = k_2(v_r + v_l)$$

Avec $k_1 = \frac{r}{2R}$ et $k_2 = r$. On obtient :

$$v_r = \frac{\omega_{Robot}}{2k_1} + \frac{V_{Robot}}{2k_2}$$

et

$$v_l = \frac{V_{Robot}}{2k_2} - \frac{\omega_{Robot}}{2k_1}$$

Calculons maintenant ω_{Robot} :
D'aprés la Figure 4, on remarque que :

$$\theta_{Goal} = arctan(\frac{y_{Goal} - y_{Robot}(t)}{x_{Goal} - x_{Robot}(t)})$$

Posons $\Delta\theta = \theta_{Robot}(t) - \theta_{Goal}$ et $V_{Robot} = V_0$. On a alors : $\omega_{Robot} = -\Delta\theta$.
Finalement, on obtient les expressions des vitesses des moteurs gauches et droits comme :

$$\begin{cases} v_r = (\frac{-\Delta\theta}{2k_1} + \frac{V_0}{(2k_2)}) \\ v_l = (\frac{V_0}{2k_2} - \frac{-\Delta\theta}{2k_1}) \end{cases}$$

2 Travail Réalisé

Toute les simulations sont faites sur V-rep (logiciel de simulation robotique) et tout les programmes sont implémentés en python.

2.1 Evitement d'obstacle

Dans cette partie, on s'intéressera aux principales étapes d'implémentation de l'algortihme vu précedemment dans la section 1.2 :

2.1.1 Lecture des informations sur les capteurs :

On utilise la fonction *SimxGetObjectHandle* pour récupérer le descripteur des capteurs et *SimxReadProximitySensor* pour lire les informations contenues dans les capteurs. Ceci doit être fait pour tous les capteurs, d'où l'utilisation de la boucle "for".

```
1 for i in range(1,17) :
2         res , sensor_handle = vrep.simxGetObjectHandle(client_id , "
      Pioneer_p3dx_ultrasonicSensor" + str(i), vrep.simx_opmode_blocking)
3         sensor_handles[i-1] = sensor_handle
4         res, detectionState, detectedPoint, detectedObjectHandle,
      detectedSurfaceNormalVector = vrep.simxReadProximitySensor(client_id, sensor_handle,
      vrep.simx_opmode_streaming)
```

2.1.2 Paramètres de l'Algorithme de Braitenberg

Les Listes BraitenbergL et BraitenbergR stockent les valeurs des pondérations pour tenir compte de l'importance des capteurs.
minSafetyDist est la distance minimale séparant le robot du l'objet et au-dessous de laquelle le robot le considère comme un obstacle.

```
1
2 minSafetyDist = 0.2
3 braitenbergL = np.array([-0.2,-0.4,-0.6,-0.8,-1,-1.2,-1.4,-1.6, 0.0, 0.0, 0.0, 0.0, 0.0,
      0.0, 0.0, 0.0])
4 braitenbergR = np.array([-1.6,-1.4,-1.2,-1,-0.8,-0.6,-0.4,-0.2, 0.0, 0.0, 0.0, 0.0, 0.0,
      0.0, 0.0, 0.0])
```

2.1.3 La boucle principale : Algorithme de Braitenberg

La ligne 5 du code ci-dessous, permet de calculer la distance séparant chaque capteur de l'obstacle. Cette distance va être utile pour la comparer avec la distance *minSafetyDist* vu précedemment (ligne 7 à 13). Le résultat de la comparaison décidera de la présence d'un obstacle ou pas. le reste du code va permettre d'appliquer les nouvelles vitesses qui tiennent compte des pondérations.

```
    while(continue_running):

        for i in range(1,17) :
            res, detectionState, detectedPoint, detectedObjectHandle,
detectedSurfaceNormalVector = vrep.simxReadProximitySensor(client_id, int(sensor_handles
[i-1]), vrep.simx_opmode_buffer)
            distToObject = math.sqrt(math.pow(detectedPoint[0], 2) + math.pow(detectedPoint
[1], 2) + math.pow(detectedPoint[2], 2)) # Calculate distance to obstacle relative to
each sensor


            if (detectionState == True) and (distToObject < maxDetectionRadius):
                if (distToObject < minSafetyDist):
                    distToObject = minSafetyDist
                detectStatus[i-1] = 1-((distToObject - minSafetyDist)/(maxDetectionRadius -
minSafetyDist))
            else:
                detectStatus[i-1] = 0

        v_l = v0
        v_r = v0

        for i in range(1,17):
            v_l = v_l + braitenbergL[i-1] * detectStatus[i-1]
            v_r = v_r + braitenbergR[i-1] * detectStatus[i-1]


        res = vrep.simxSetJointTargetVelocity(client_id, left_motor, v_l, vrep.
simx_opmode_oneshot)
        res = vrep.simxSetJointTargetVelocity(client_id, right_motor, v_r, vrep.
simx_opmode_oneshot)
```

2.1.4 Exemple de simulations

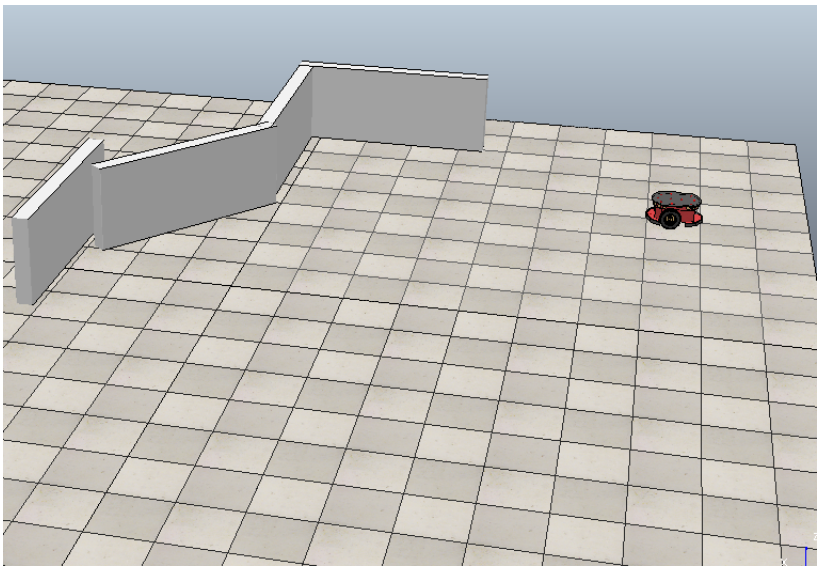

Figure 5: **Robot Pioneer - Simulation sur V-rep**

Voir la vidéo de la simulation. CLIQUEZ ICI

2.2 Evitement d'obstacle et Navigation vers un but

Dans cette partie, j'ai essayé de combiner les deux stratégies vus dans les sections 1.2 et 1.3 pour permettre au robot Pioneer d'arriver à sa cible sans toucher aucun obstacle.
Globalement, au code d'évitement d'obstacle, j'ai ajouté le code permettant la navigation vers un but.
Voici les principales parties du code implémentant ce principe :

2.2.1 Récupération des positions des objets

Deux fonctions sont utiles pour cette partie :

simxGetObjectPosition pour récupérer les positions d'un objet.
simxGetObjectOrientation pour récupérer la direction du robot.

```
1       # Goal position
2       res, tmp2 = vrep.simxGetObjectPosition(client_id,wall,-1, vrep.simx_opmode_oneshot_wait)
3       Goal_position[0]= tmp2[0]
4       Goal_position[1]= tmp2[1]
5
6       # Robot position
7       res, tmp = vrep.simxGetObjectPosition(client_id,pioneer,-1, vrep.
        simx_opmode_oneshot_wait)
8       Robot_position[0]= tmp[0] #X_r
9       Robot_position[1]= tmp[1] #Y_r
10      res, tmp = vrep.simxGetObjectOrientation(client_id, pioneer, -1, vrep.
        simx_opmode_oneshot_wait)
11      Robot_position[2] = tmp[2] # en radian
```

2.2.2 Les équations mathématiques

Le code suivant montre l'implémentation des équations mathématiques vus dans la section 1.3 ($D(t)$, $\Delta\theta$, θ_{Robot}, V_{Robot}, ω_{Robot}, v_l et v_r).

```
    #Distance to Goal
    d = math.sqrt(math.pow(Goal_position[0] - Robot_position[0],2) + math.pow(Goal_position
    [1]- Robot_position[1],2))

    Goal_teta = math.atan((Goal_position[1] - Robot_position[1])/(Goal_position[0] -
    Robot_position[0]))
    delta_teta = Robot_position[2] - Goal_teta
    w0  = -delta_teta

    # Wheel speeds if no obstacle is near the robot

    v_left = ((v0/(2*k1) - w0/(2*k2)))
    v_right = ((v0/(2*k2)) + w0/(2*k1))
```

2.2.3 Evitement d'obstacle

Notre programme doit détecter les obtacles et les éviter, donc les expressions des vitesses changeront pour tenir compte de l'algorithme de Bertenberg. Plus concrètement, les vitesses des roues doivent valoir d'abord une vitesse linéaire v_0, pour ralentir et fixer une même direction des roues du robot, puis on pourra après appliquer les vitesses v_r et v_l de Braitenberg.
La variable *state* permet de détecter la présence d'un obstacle.

```
    #Distance to Goal
    d = math.sqrt(math.pow(Goal_position[0] - Robot_position[0],2) + math.pow(Goal_position
    [1]- Robot_position[1],2))

    Goal_teta = math.atan((Goal_position[1] - Robot_position[1])/(Goal_position[0] -
    Robot_position[0]))
    delta_teta = Robot_position[2] - Goal_teta
    w0  = -delta_teta

    # Wheel speeds if no obstacle is near the robot

    v_left = ((v0/(2*k1) - w0/(2*k2)))
    v_right = ((v0/(2*k2)) + w0/(2*k1))
```

2.2.4 L'arrivée au but

Pour éviter les oscillations du robots à son arrivée à la cible, on peut considérer une distance seuil (variable *thresold* dans le code) au-dessous de laquelle le robot est considéré arrivé au but. On peut aussi annuler la vitesse des roues pour le stabiliser.

```
    # cancel the speed and stop the simulation if the robot has reached the objective
    if (d<=thresold_distance):
        res = vrep.simxSetJointTargetVelocity(client_id, left_motor, 0, vrep.
    simx_opmode_oneshot)
        res = vrep.simxSetJointTargetVelocity(client_id, right_motor, 0, vrep.
    simx_opmode_oneshot)
        print("Robot reached the goal ")
        continue_running = False
```

2.2.5 Exemple de simulations

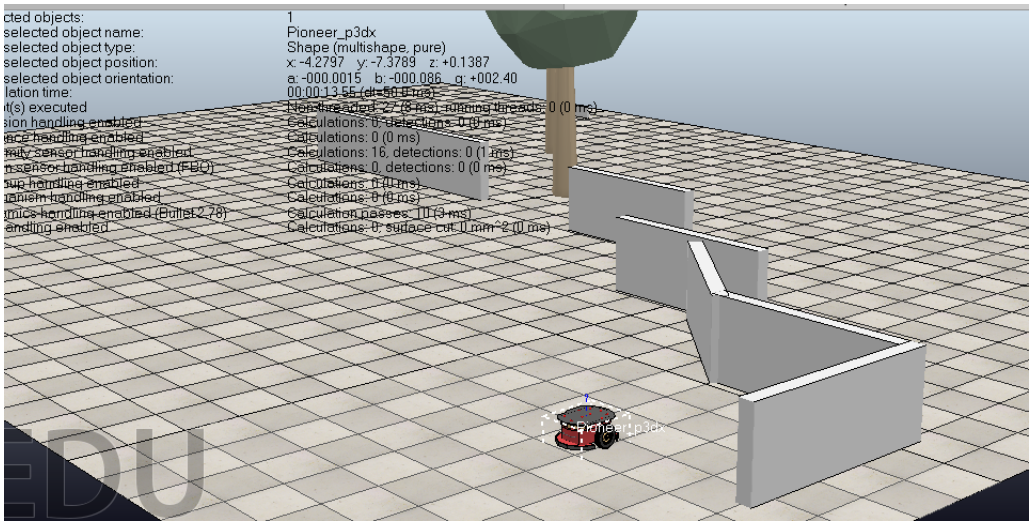

Figure 6: **Robot Pioneer - Simulation sur V-rep**

Remarque

La cible est modélisée par l'objet "Wall" dans la simulation. Le choix est motivé par le fait qu'il est plus simple de récupérer sa position. Dans ce cas il suffira d'utiliser la fonction *simxGetObjectPosition*.

Voir la vidéo de la simulation: CLIQUEZ ICI

Conlusion

Les simulations faites à travers ce travail montre qu'à partir des systèmes de structure très simple, on peut obtenir des comportements très complexes qui peuvent être traduits par une capacité d'adaptation et une remarquable robustesse du robot.
Cependant, l'efficacité de ce modèle devient très vite sans interêt, surtout quand il s'agit des comportements plus réaliste de nature humaine. Grâce à l'intelligence artificielle et plus précisement de l'apprentissage profond, on arrive aujourd'hui à simuler ces comportements et à s'approcher du modèle réelle.