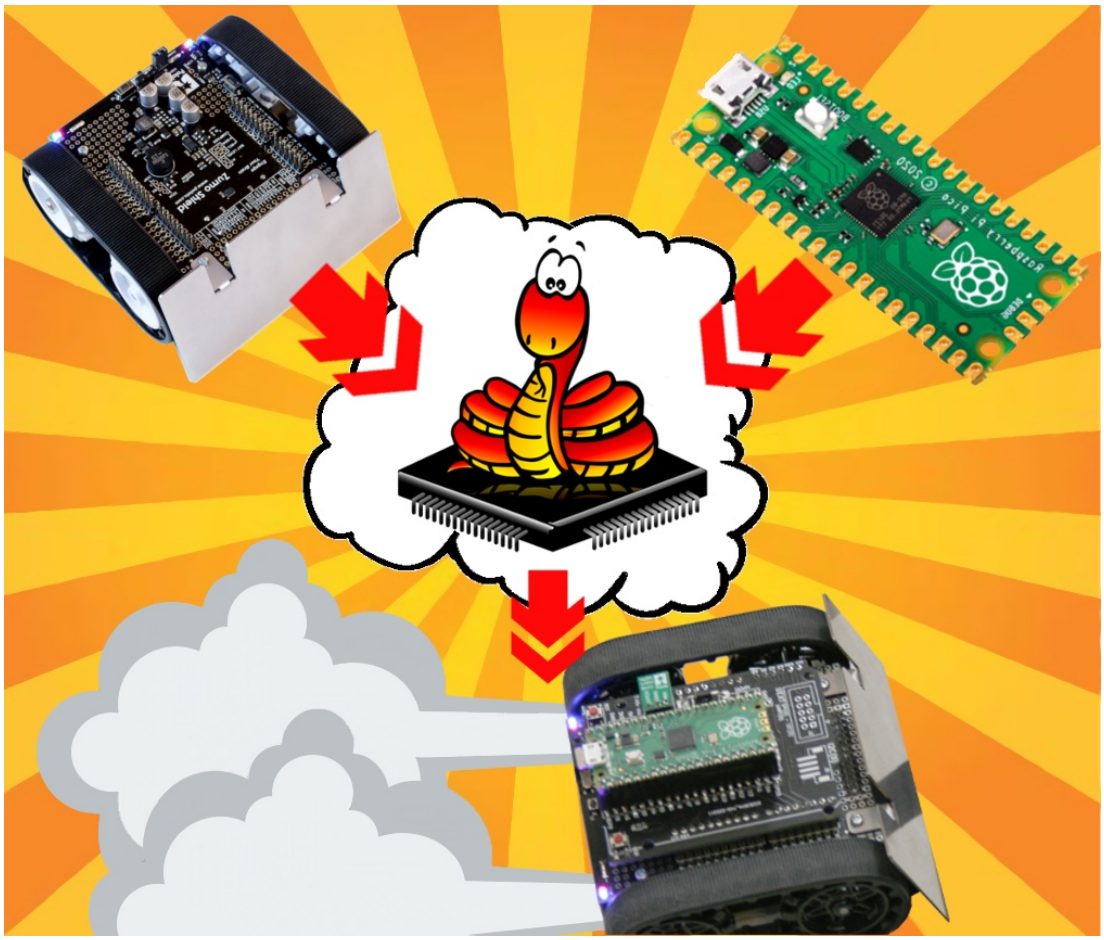
Pico, Zumo Robot et MicroPython

Programmer le Zumo Robot avec Python pour Microcontrôleur



Exemples

Table des matières

1.Introduction

Le chapitre précédent s'est concentrée sur la maîtrise des différentes fonctionnalités du Robot Zumo à l'aide de la bibliothèque MicroPython.

Le présent chapitre va combiner différentes fonctionnalités pour réaliser des « projets exemples » aux tâches plus complexes. Le fonctionnement des projets sera également détaillé pour permettre à tout un chacun de comprendre les rouages.

Voici quelques « projets exemples » autour du Zumo :

•Détection de bordure : évoluer dans l'espace clos d'un combat mini-sumo sans en sortir.

•Suiveur de ligne : suivre un parcours dessiné sur le sol

•Tourner en carré : réaliser un angle de 90° n'est pas aussi trivial qu'on pourrait le penser.

•Résolution de labyrinthe : Parcourir un labyrinthe tracer au sol et en trouver la sortie.

•Langage Logo : un langage et des commandes simples pour contrôle le Robot Zumo.

2.Activation automatique

L'intérêt des exemples présentés dans ce chapitre est de pouvoir les utiliser en situation et si possible sans avoir besoin d'établir une connexion REPL.

Pour se passer de la session REPL, il faut que le Zumo Robot démarre l'exemple automatiquement l'exemple dès sa mise sous tension.

2.1.Démarrer automatiquement le script

Chaque exemple est articulé autour d'un script unique (ex:`line_follower.py`) qui s'appuie sur les bibliothèques du Zumo Robot constitué des fichiers `zumoshield.py` et `zumoimu.py` .

Pour que le script d'exemple démarre à la mise sous tension du Zumo Robot il faut :

1.Copier le contenu de l'exemple dans le fichier `main.py` .

2.Éditer le fichier le fichier `main.py` pour ajouter l'instruction `import line_follower` .

2.2.Suspendre le démarrage du script

Pour suspendre le démarrage automatique du script, il faut désactiver l'exécution de main.py .

Pour y arriver :

1.Brancher le Pico en USB sur un ordinateur **mais ne pas activer l'alimentation du Zumo Robot.**

2.Etablir une liaison REPL avec le Pico avec votre logiciel favori (ThonnyIDE ou MPRemote).

3.Le script main.py est actuellement en cours d'exécution. Il faut **interrompre son exécution en pressant la combinaison de touche [CTRL]+C** .

4.L'invite de commande MicroPython « >>> » est maintenant visible dans la session REPL.

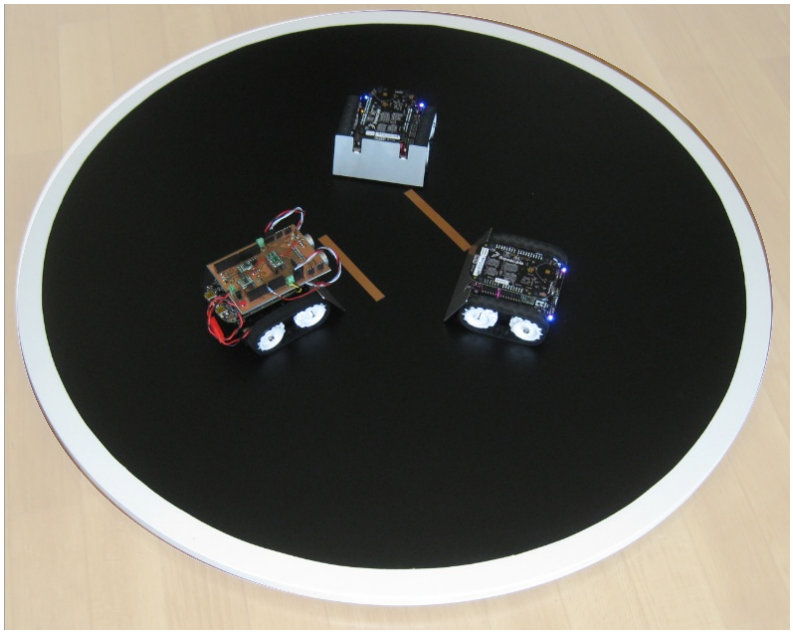
5.Saisir les instructions suivantes pour effacer le fichier `main.py` de sorte qu'il ne soit plus exécuté au prochain démarrage.

```
>>> import os
>>> os.remove( 'main.py' )
```

3.Détection de bordure

La détection de bordure est la fonctionnalité fondamentale utilisée dans les compétitions de robot mini-sumo, à savoir se déplacer sur la zone de combat sans en sortir.

A propos des compétitions mini-sumo



07RI20 – Combat de mini-sumo (source : Erich Styger, mcuoneclipse.com)

Le but d'une compétition de mini-sumo est de pousser l'adversaire hors du ring en respectant quelques règles :

1.La zone de combat (dit *Dohyo*) fait 77cm de diamètre, en noir et rehaussé. Cette zone est entourée d'une bordure blanche de 2,5cm de largeur.

2.Le robot doit avoir une taille maximale de 10x10 cm et peser moins de 500 gr.

3.Le combat se fait à 2 ou 3 robots.

4.Le robot peut augmenter sa taille après après le début du round (mais doit rester en une seule pièce).

5.Le robot doit attendre 5 secondes après le début du round pour commencer à bouger.

6.Le robot doit être entièrement autonome (pas de contrôle à distance ou autre procédé de contrôle).

Un robot perd le round s'il est expulsé de la zone de combat. Le robot qui perd deux rounds d'affilé perd le combat.

Idéalement, le robot-sumo doit cesser toute activité s'il tombe hors de la zone de combat ou qu'il est soulevé par l'utilisateur. Un accéléromètre peut être utilisé pour détecter ces deux cas.

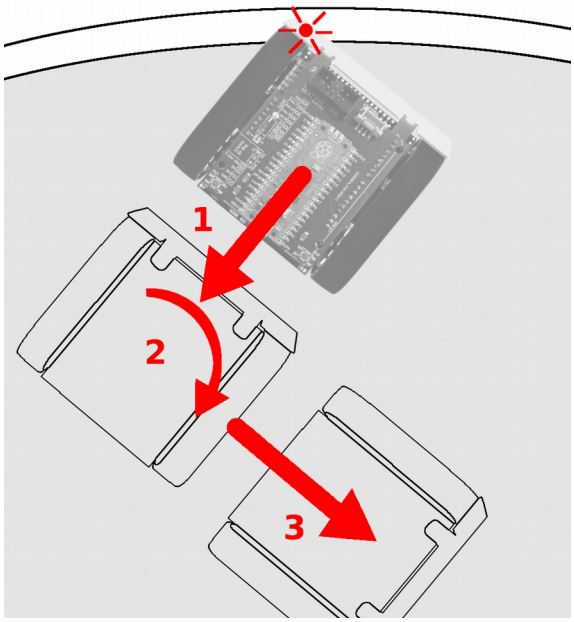
3.1.Principe de fonctionnement

Le Robot Sumo se déplace en ligne droit jusqu'à rencontrer la bordure du ring.

Comme la bordure du ring est blanche alors le détecteur infrarouge retourne une valeur inférieure à 1000.

Le ring présente une courbure ce qui signifie que ce sera le premier ou le dernier détecteur infrarouge qui sera activé.

Le graphique ci-dessous indique comment le Robot Zumo réagit lorsque sont premier capteur (celui de gauche) est activé par la bordure blanche de la zone de combat



07RI21 – premier détecteur activé

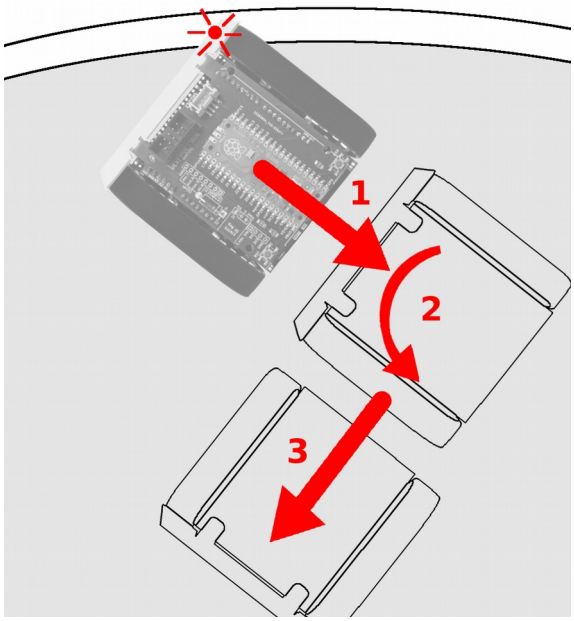
A la détection sur le premier capteur :

1.Le Zumo s'arrête et effectue une courte marche arrière

2.Ensuite le Zumo tourne sur sa droite (sens horlogique) pendant quelques instants

3.Enfin reprend une marche avant jusqu'à la prochaine détection de bordure

Le comportement est similaire (mais inversé) si c'est le dernier capteur qui est activé. Lorsque le Zumo rencontre la bordure du ring par la droite alors sa rotation sera faite dans le sens anti-horlogique.



07RI22 – dernier détecteur activé

Un peu de chaos dans la stratégie

S'il s'avère que le Zumo à déjà tourné plus de 3 fois à droite et plus de 3 fois à gauche (>3) alors à la prochaine détection à gauche, le Zumo reculera plus longtemps et tournera aussi plus longtemps.

De la sorte, le Zumo modifie sa position de façon significative sur la zone de combat avant de reprendre son comportement répétitif.

3.2.Script détection de bordure

Voici le contenu du script permettant de transformer le Zumo Robot en sumo-robot.

Ce script est disponible dans le dépôt du projet sur le lien suivant :

https://github.com/mchobby/micropython-zumo-robot/blob/main/examples/border_detect.py

```
01: from zumoshield import ZumoShield
02: from zumobuzzer import NOTE_G
03: import time
04:
05: QTR_THRESHOLD = 1000
06: FORWARD_SPEED = 200
07: REVERSE_SPEED = -150
08: TURN_SPEED    = 200
09: REVERSE_DURATION= 400
10: TURN_DURATION   = 400
11:
12: z=ZumoShield()
13:
14: def waitForButtonAndCountDown():
15:     global z
16:     z.led.value(1)
17:     z.button.waitForButton()
18:     z.led.value(0)
19:     for x in range(3):
20:         time.sleep(1)
21:         z.buzzer.playNote( NOTE_G(3),200,15 )
```

```
22:     time.sleep(1)
23:     z.buzzer.playNote( NOTE_G(4),500,15 )
24:     time.sleep(1)
25:
26: left_count = 0
27: right_count= 0
28: z.play_blip()
29: waitForButtonAndCountDown()
30:
31: while(True):
32:     z.ir.read()
33:     if z.ir.values[0] < QTR_THRESHOLD :
34:         z.motors.setSpeeds( REVERSE_SPEED,REVERSE_SPEED )
35:         time.sleep_ms( REVERSE_DURATION )
36:         z.motors.setSpeeds( TURN_SPEED,-1*TURN_SPEED )
37:         time.sleep_ms( TURN_DURATION )
38:         z.motors.setSpeeds( FORWARD_SPEED,FORWARD_SPEED )
39:         left_count+=1
40:
41:     elif z.ir.values[5] < QTR_THRESHOLD :
42:         if left_count>3 and right_count>3 :
43:             REVERSE_DURATION = 800
44:             TURN_DURATION    = 800
45:             left_count = 0
46:             right_count= 0
47:         z.motors.setSpeeds( REVERSE_SPEED,REVERSE_SPEED )
48:         time.sleep_ms( REVERSE_DURATION )
49:         z.motors.setSpeeds( -1*TURN_SPEED,TURN_SPEED )
50:         time.sleep_ms( TURN_DURATION )
51:         z.motors.setSpeeds( FORWARD_SPEED,FORWARD_SPEED )
52:
53:         REVERSE_DURATION = 400
54:         TURN_DURATION    = 400
55:         right_count+=1
56:
57:     else:
58:         z.motors.setSpeeds( FORWARD_SPEED,FORWARD_SPEED )
```

Voici quelques détails sur fonctionnement du script :

•Lignes 1 à 3 : Import des modules et fonctions nécessaires. La fonction NOTE_G permettra de produire une note Sol sur différents octaves.

•Ligne 5 : Définition de la constante `QTR_THRESHOLD` à 1000, valeur numérique en dessous de laquelle un capteur infrarouge détecte une **zone réfléchissante**.

Une surface réfléchissante (blanche) retourne une valeur de l'ordre de 250 à 400 tandis qu'une surface non réfléchissante (noire) retourne une valeur de l'ordre de 2000 .

•Ligne 6 : Définition de la constante `FORWARD_SPEED` à 250. Constante utilisée pour la **marche avant**. Pour rappel, la vitesse doit être située dans l'intervalle 0 et 400.

•Ligne 7 : Définition de la constante `REVERSE_SPEED` à -150. Constante utilisée durant les **marches arrières** (durant les changements de direction).

•Ligne 8 : Définition de la constante `TURN_SPEED` à 200. Constante utilisée lorsque le Zumo **tourne** sur lui-même.

•Lignes 9 et 10 : Définition des constantes `REVERSE_DURATION` et `TURN_DURATION` qui représente, respectivement, le temps de marche arrière et

temps de rotation lors des phases de changement de direction. Ces temps sont fixés à 400 millisecondes par défaut.

•Ligne 12 : Création de l'objet `z`, instance de la classe `ZumoShield` permettant d'accéder au fonctionnalités du Zumo Robot.

•Ligne 14 à 24 : Définition de la fonction `waitForButtonAndCountDown` qui attend la pression du bouton utilisateur puis entame un décompte sonore avant de démarrer le Zumo.

•Ligne 15 : L'instruction `global` indique la variable `z` est de portée globale. Cela permet à la fonction d'accéder à la variable `z` définie dans le script principal.

•Ligne 16 : Activation de la LED utilisateur du Zumo

•Ligne 17 : Attendre que l'utilisateur presse le bouton du Zumo.

•Ligne 18 : Extinction de la LED utilisateur du Zumo. Le décompte audio va débuter.

•Lignes 19 à 21 : Boucle de 3 itérations de pause d'une seconde suivit de la production d'un Sol (3ieme octave) durant 200 millisecondes.

•Lignes 22 à 24 : Pause d'une seconde accompagnée de la production d'un Sol (4ieme octave) durant 500 millisecondes. Une dernière pause d'une seconde est encore exécutée avant de quitter la fonction `waitForButtonAndCountDown` .

•Ligne 25 : **Première ligne du script principal**.

•Lignes 26 à 27 : Déclaration de la variable `left_count` qui compte le nombre de contact à gauche sur le détecteur de ligne. La variable `right_count` quand à elle compte le nombre de contact à droite.

•Ligne 28 à 29 : Emission d'un blip sonore indiquant que le script est bien démarré juste avant l'appel de la fonction `waitForButtonAndCountDown` qui attend la pression du bouton utilisateur (et début la décompte audio). A noter que la LED utilisateur du Zumo est allumée pendant l'attente de la pression du bouton.

•Ligne 31 : Boucle `while` infinie qui prend en charge le fonctionnement du Robot Zumo dans la zone de combat. Les lignes 32 à 58 seront exécutées encore et encore. A noter que si cette ligne du script est exécutée c'est que l'utilisateur a effectivement appuyé sur le bouton utilisateur.

•Ligne 32 : lecture du détecteur de ligne (lecture non calibrée). Les données sont accessibles via la propriété `values` .

•Lignes 33, 41 et 57 : Test de différents cas de figures à savoir :

□Ligne 33 : Détection de la ligne blanche sur le capteur de gauche (premier capteur).

□Ligne 41 : Détection de la ligne blanche sur la droite du détecteur de ligne (dernier capteur).

□Ligne 57 : Si aucun des deux cas précédent est concerné alors la plateforme est simplement lancée en marche avant.

•Ligne 24 à 39 : Ces lignes sont exécutées lorsque le Zumo entre en contact avec la limite du ring **par la gauche** du détecteur de ligne (premier capteur).

□Ligne 34 : passe en marche arrière (la constante `REVERSE_SPEED` est négative donc le Zumo recule).

□Ligne 35 : recule pendant la pause de 400 millisecondes mentionnées dans la constante `REVERSE_DURATION` .

□Ligne 36 : rotation sur la droite du Zumo. L'utilisation du multiplicateur -1 fait tourner la chenille droite en marche arrière (vitesse négative) alors que la chenille gauche est en marche avant.

□Ligne 37 : Tourne durant la pause de 400 millisecondes mentionnées dans la constante `TURN_DURATION` .

□Ligne 38 : Remet le Zumo Robot en marche avant.

□Ligne 39 : Incrémenter le compte de tournant à gauche (variable `left_count`).

•Lignes 41 à 55 : Ces lignes sont exécutées lorsque le Zumo entre en contact avec la limite du ring **par la droite**.

□Ligne 42 à 46 : Si les compteurs contact à gauche (`left_count`) et contact à droite (`right_count`) ont tous deux dépasser 3, c'est le moment d'appliquer la stratégie alternative : tourner plus vite et plus longtemps. Les constantes `TURN_DURATION`, `REVERSE_DURATION` sont temporairement redéfinies à 800. Les compteurs `left_count` et `right_count` sont également remis à zéro.

□Lignes 47 à 51 : Marche arrière, tourner à gauche et remise en marche avant. Il est utile de noter que pour la rotation le multiplicateur -1 est appliqué à la chenille gauche. Ces lignes utilisent également les valeurs de `REVERSE_DURATION` et `TURN_DURATION` qui sont toutes deux à 400 millisecondes (ou 800 millisecondes dans le cas particulier des lignes 42 à 46).

□Lignes 53 et 54 : en tout état de cause, refixer les valeurs `REVERSE_DURATION` et `TURN_DURATION` à 400 millisecondes.

□Ligne 55 : comme le Zumo vient de détecter un contact par la droite, la variable `right_count` est incrémenté de 1.

3.3.Auto-critique : les constantes en Python

Par convention, les constante Python sont définie par des mots clés en lettres capitales. Une constante est supposée être immuable (non modifiable) une fois celle-ci définie.

Cependant, selon les préceptes Python, le programmeur reste au centre du développement et il faut lui faire confiance. Ainsi, en Python, un objet nommé en capital n'est rien d'autre qu'un objet, il reste donc possible d'en altérer la valeur !

Dans le script ci-dessus les constantes `REVERSE_DURATION` et `TURN_DURATION` sont modifiées à dessein et pour une très courte période, la valeur d'origine étant ensuite restaurée.

Dans une approche plus puriste, il aurait été préférable d'opter la définition de deux variables supplémentaire `reverse_duration` et `turn_duration` qui, elles, sont supposées être modifiables

3.4.Encore plus

Il est possible d'apporter quelques améliorations directe à ce premier script :

1.Détecter la pression du bouton utilisateur pour aussi arrêter le script.

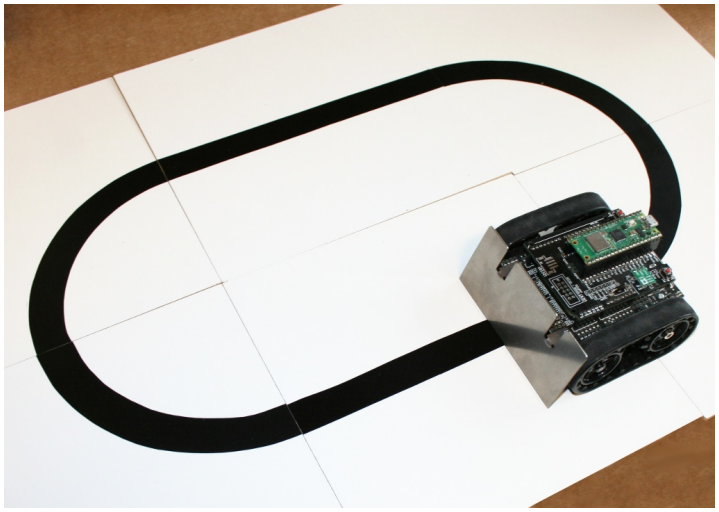
2.Utiliser l'accéléromètre pour détecter la chute du ring (ou soulever de la zone de combat).

3.Adapter le script pour qu'il fonctionne aussi sur un ring blanc bordé par une frontière noire.

4.Suiveur de ligne

Le suiveur de ligne, comme son nom l'indique permet au Zumo Robot de suivre une ligne. Celle-ci doit faire 15mm de large et être noire (opaque).

Il est très facile de réaliser un circuit modulaire en faisant découper un panneau blanc en carré de 20cm x 20cm puis d'équiper ces derniers avec des éléments découpés dans des feuilles de vinyle opaque/noir autocollant.



07RI06 – Zumo Robot suiveur de ligne

Voici les détails fonctionnels du script :

1.Produit un signal sonore et attends la pression du bouton utilisateur.

2.Effectue une calibration du capteur de ligne en faisant osciller le Zumo au dessus de celle-ci

3.Produit un second signal sonore

4.Démarre immédiatement les moteur du Zumo et le suivit de ligne.

Le suivit de ligne s'exécute dans une boucle infinie, le script ne s'interrompt donc jamais.

Pour arrêter le suiveur de ligne, il faut :

1.Soit mettre le Zumo Robot hors tension

2.Soit presser le bouton Reset (réinitialisation), le script sera en pause dès le début du script (juste avant la phase de calibration).

4.1.Principe de fonctionnement

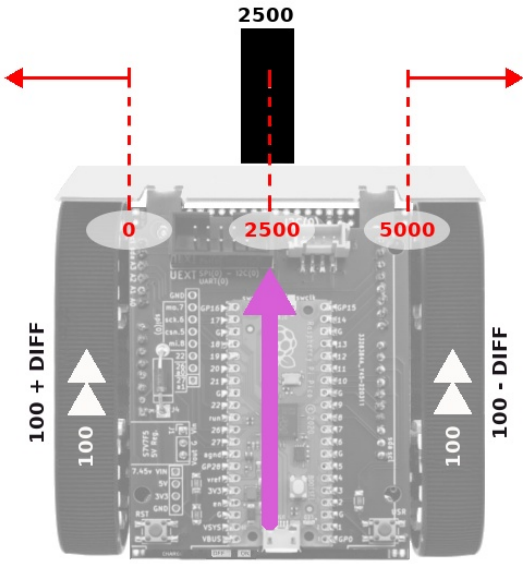
Le détecteur de ligne retourne une valeur numérique via la méthode `readLineBlack()` , valeur située entre 0 et 5000. Avec la ligne en position centrale, la valeur retournée est pile de 2500.

Avec une ligne en position centrale, les deux moteurs peuvent tourner à la même vitesse fixée à 100.

Si la ligne dévie sur la droite (ou la gauche) alors il faut évaluer l'erreur (l'écart par rapport à la ligne) et appliquer un correctif sur la vitesse des moteurs. Comme détaillé lors des tests moteurs, une différence modérée entre la vitesse du moteur droit et du moteur gauche courbe la trajectoire (cf. Tester – Commande Moteur).

4.1.1.Ligne centrée

Dans le cas d'une trajectoire rectiligne avec la ligne pile au centre du capteur de ligne, la situation est la suivante :



07RI07 – Ligne centrée sous le Zumo

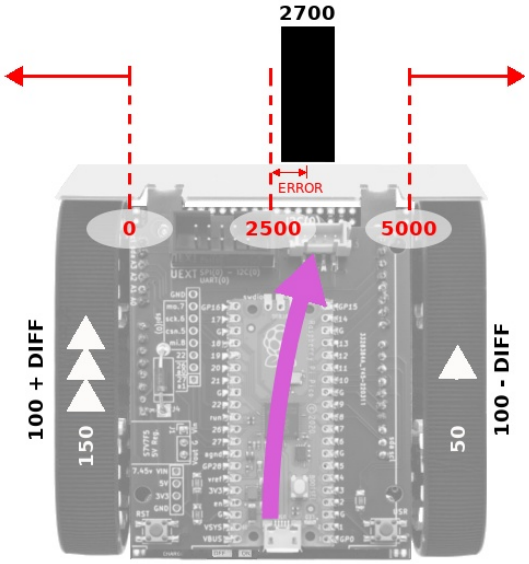
Avec la ligne pile au centre du capteur, il n'y a pas d'erreur (pas d'écart) par rapport à la trajectoire à suivre.

Dans pareil cas, il ne faut pas appliquer de différence de vitesse entre les moteurs. DIFF est égale à 0 et les deux moteurs sont propulsés à la même vitesse de 100.

Le lecteur attentif notera que les deux moteurs sont impactés par la même différence mais dans des signes opposés (ajouté d'un côté et soustrait de l'autre). Cela fait sensiblement « pivoter » le zumo autour de son centre de gravité. A noter aussi qu'une différence (DIFF) de 20 provoquera une différence de vitesse boublée (40) entre les moteurs.

4.1.2.Ligne décentrée à droite

Si la ligne à suivre passe sur la droite du capteur, il faut que le Zumo Robot entame une trajectoire courbe vers la droite pour ramener la ligne vers centre du capteur.



07RI08 – Ligne décentrée sous le Zumo

Dans le cas présent, l'erreur de trajectoire est ERROR = 2700 - **2500** = 200.

La différence de vitesse (DIFF) est une portion fixe de l'erreur. Ce **facteur proportionnel** est fixé à 1/4 dans ce projet.

DIFF = ERROR / 4 = 200 / 4 = 50

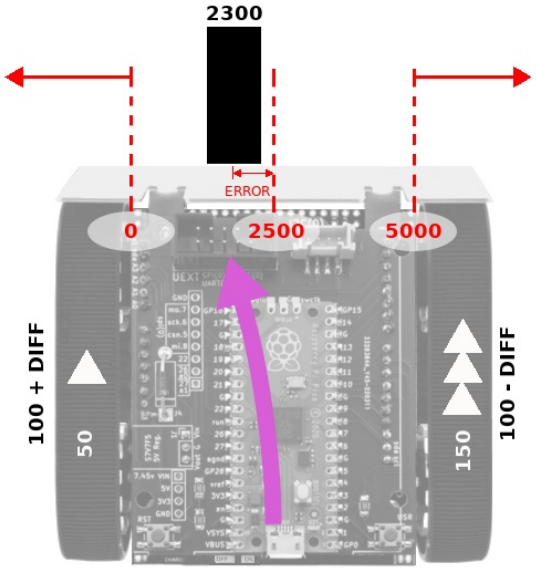
La vitesse du moteur gauche = 100 + DIFF = 150.

La vitesse du moteur droit = 100 – DIFF = 50.

Dans cette configuration, le Zumo entame une courbure vers la droite.

4.1.3.Ligne décentrée à gauche

Si la ligne à suivre passe sur la gauche du capteur, il faut que le Zumo Robot entame une trajectoire courbe vers la gauche pour ramener la ligne vers centre du capteur.



07RI09 – Ligné décentrée à gauche

L'erreur de trajectoire est ERROR = 2300 – **2500** = -200 .

La différence de vitesse est toujours calculée avec le même **facteur proportionnel** de 1/4 .

DIFF = ERROR / 4 = -200 / 4 = -50

Vitesse du moteur gauche = 100 + DIFF = 100 + (-50) = 50.

Vitesse du moteur droit = 100 – DIFF = 100 - (-50) = 100 + 50 = 150

4.2.Amélioration du principe

Cette section présente des concepts plus complexes et donc plus difficile a appréhender.

La compréhension de ces concepts n'est pas indispensable à la compréhension du projet mais représente néanmoins un savoir technique non négligeable.

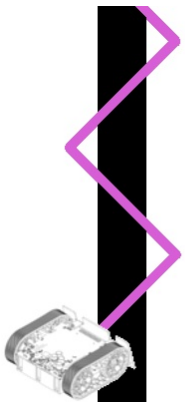
4.2.1.Limite de l'asservissement proportionnel

Les exemples ci-dessus utilisent exclusivement un facteur proportionnel (une proportion de l'erreur) pour corriger la trajectoire du Zumo Robot.

Dans cet asservissement proportionnel, le multiplicateur numérique est appelé **le gain** (gain de 0,25 ou 1/4 dans l'exemple ci-avant).

Si le principe de fonctionnement est correct et fonctionnel, il souffre d'un défaut notable car le **Zumo Robot sera constamment face à une erreur de trajectoire** qu'il faudra corriger.

Une erreur de trajectoire est logique dans une courbe mais cela sera également le cas pour une trajectoire en ligne droite !



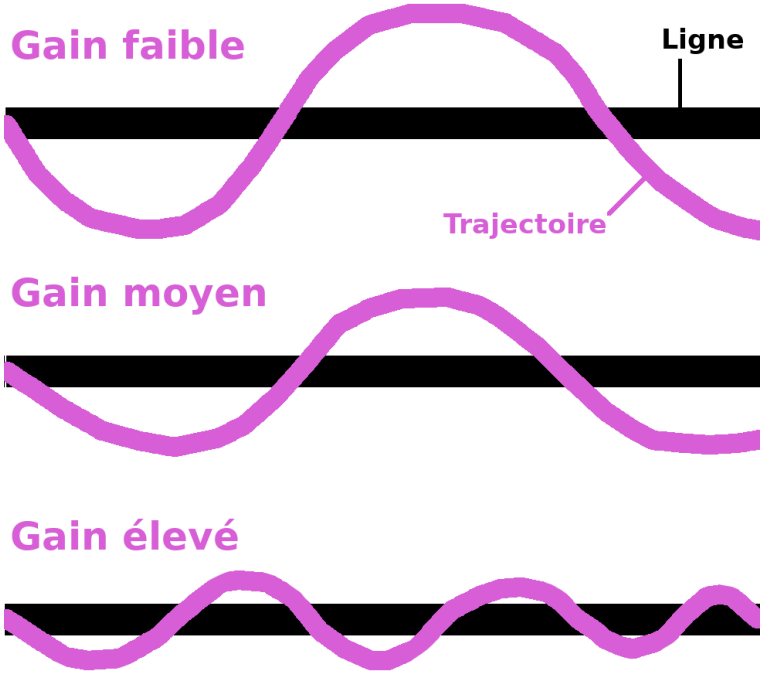
07RI10 – Trajectoire en facteur proportionnel

Sur une ligne droite le Zumo avec asservissement proportionnel avance en zig-zag, ce qui est nettement moins efficace qu'un vrai trajet en ligne droite. Ce déplacement oscillatoire représente aussi un surcroît de trajet et de consommation énergétique.

Le comportement de l'asservissement proportionnel est dépendant de la valeur du du gain. Un **gain proportionnel trop faible** et le Zumo fera de larges boucles (voir quitter la ligne).

Un **gain proportionnel trop élevé** et le Zumo aura un comportement frénétique cherchant à rejoindre la ligne aussi rapidement que possible (produisant ainsi les zig-zag serrés).

Un **gain proportionnel idéal** (plus adapté) offrira toujours un comportement sous forme de boucle mais ces dernières ne seront ni trop large (échappée), ni trop serrée (comportement frénétique).



07RI15 – gain proportionnel et trajectoire

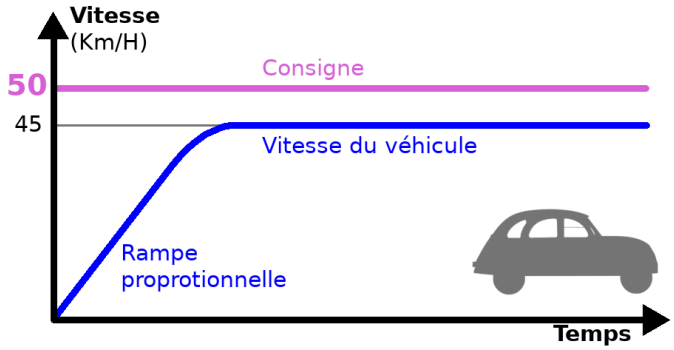
4.2.2.Différents types d'asservissement

Régulation P (Proportionnel)

Dans un système asservi utilisant un facteur de correction proportionnel (comme ci-dessus), le système à tendance a rester sous la consigne puisque seul un écart (une erreur) permet de ramener le signal vers la consigne.

Pour rendre ceci plus facile a appréhender (plus tangible), cette section se concentre sur le contrôle de la vitesse d'un véhicule plutôt que le contrôle de la direction du Zumo.

Soit un régulateur de vitesse proportionnel monté sur une 2 chevaux. Si ce régulateur à pour consigne 50 Km/h alors le régulateur accélérera jusqu'à une vitesse légèrement inférieure à 50 Km/H, vitesse réelle (ex : 45 Km/h) à laquelle le véhicule se stabilisera.



07RI11 – Asservissement proportionnel

Dans le cas du Zumo Robot, la consigne est à 2500, cela signifie que le Zumo resterait d'un côté de la ligne.

Ceci étant :

•d'une part, le robot est déjà au dessus de la ligne et,

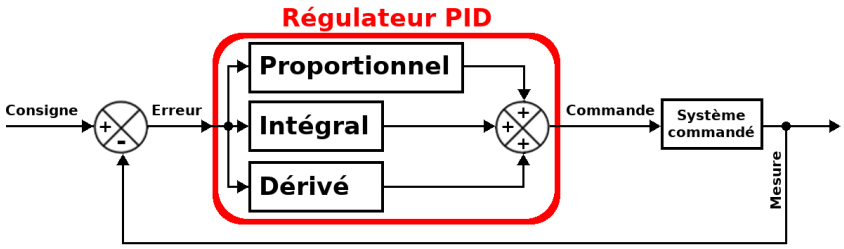
•d'autre part, le contrôle moteur fera assez rapidement pivoter le Zumo autour de son centre de gravité.

En conséquence, le capteur de ligne situé à l'avant du Zumo Robot aura vite fait de balayer la ligne sur une amplitude relativement importante (même avec une différence modérée dans les vitesse moteurs) d'où ce phénomène de zig-zag / d'aller-retour au dessus de la ligne.

Régulation PID

Dans le monde de la régulation, la référence est le régulateur PID (Proportionnel + Intégral + Dérivé). Sans vouloir rentrer dans de long détails mathématique il est cependant possible d'appréhender l'impact des différents composantes de cette régulation.

Comme indiqué, ce type de régulateur effectue **la somme** de 3 éléments distincts : partie proportionnelle, une partie intégrale et une partie dérivée.



07RI12 – Eléments d'un régulateur PID

Dans un régulateur PID, les différents éléments sont souvent présentés comme ceci :

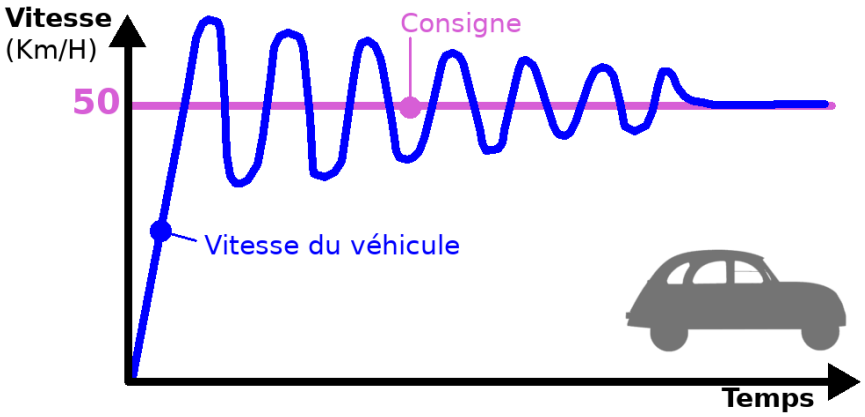
•**Partie Proportionnelle** : s'occupe du présent, en fonction de l'erreur immédiate.

•**Partie Intégrale** : s'occupe du passé en accumulant les erreurs (positive et négatives) constatées précédemment durant les cycles de régulation. Sert à approcher la consigne par « l'accumulation de petits extras » mais au prix d'une oscillation autour de la consigne.

•**Partie dérivée** : s'occupe du futur en corrigeant l'addition trop généreuse de la partie intégrale afin d'essayer de stopper celle-ci au bon moment. La dérivée compense l'excès de l'intégrale.

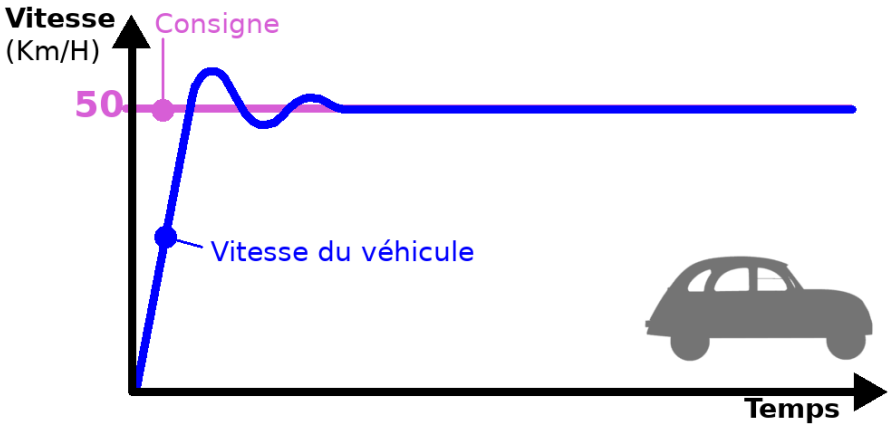
Pour reprendre l'exemple de la 2 chevaux ci-avant, voici ce que produirait une régulation de vitesse P+I.

Il est possible d'y constater que la consigne est atteinte mais aussi que la vitesse oscille autour de la consigne. Cette oscillation parfois relativement importante en amplitude présente surtout un faible amortissement, ce qui signifie que l'oscillation perdure assez longtemps dans le temps.



07RI13 – Régulation PI de la vitesse

L'ajout de la fonction dérivée (donc PI+D) permet d'apporter une correction au régulateur PI en apportant un amortissement plus rapide des oscillations, ce qui permet d'atteindre la consigne plus rapidement en diminuant significativement (voire annulant) l'effet oscillatoire de l'intégrale.



07RI14 – Régulation PID de la vitesse

Cette introduction sommaire à la régulation laisse entrevoir qu'il est possible d'améliorer la réponse du suiveur de ligne.

4.2.3.Asservissement Proportionnel-Dérivé

L'asservissement proportionnelle du Zumo Robot (DIFF = ERROR / 4) fait osciller celui-ci autour de la ligne, un peu comme l'oscillation d'un régulateur PI.

Ce qu'il faudrait, c'est amortir cette oscillation pour donner une change au Zumo de tendre vers la ligne plutôt que la couper sans cesse (passant ainsi d'un côté à l'autre).

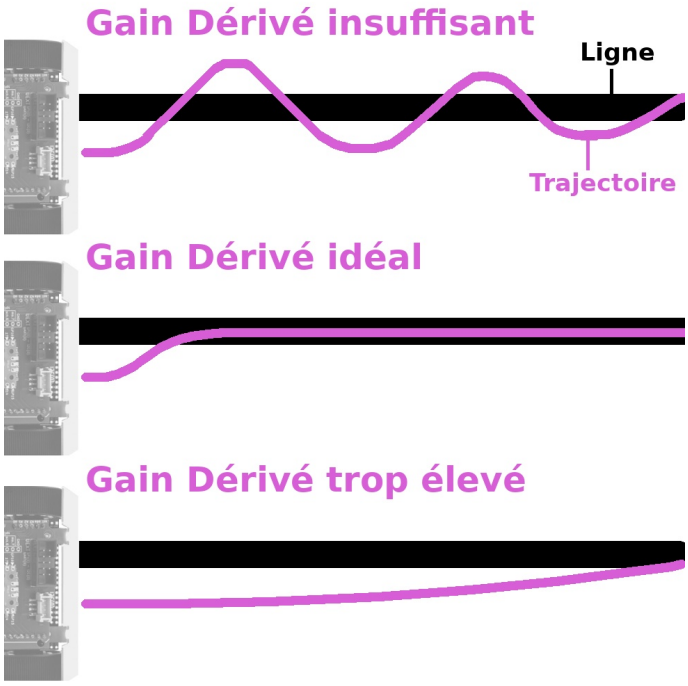
Dans un régulateur PID, c'est la composante Dérivée qui permet d'amortir la régulation. La composante dérive agit clairement dans le sens opposé du gain proportionnel (un peu comme un ressort de rappel qui tire de plus en plus fort en arrière).

La composante Dérivée dispose aussi de son propre **gain dérivé** permettant de moduler l'effet compensatoire sur l'asservissement proportionnel.

Si le **gain dérivé est insuffisant** il n'amortit pas assez la régulation proportionnelle du Zumo Robot, par conséquent le système continue d'osciller même s'il tend lentement vers la consigne.

Si le **gain dérivé est trop important** alors il amortit tellement le système que la consigne n'est atteinte qu'après une longue période de régulation.

Si le **gain dérivé idéal** est atteint alors l'effet oscillatoire sera parfaitement compensé et le Zumo s'arrête rapidement sur la consigne (la ligne).



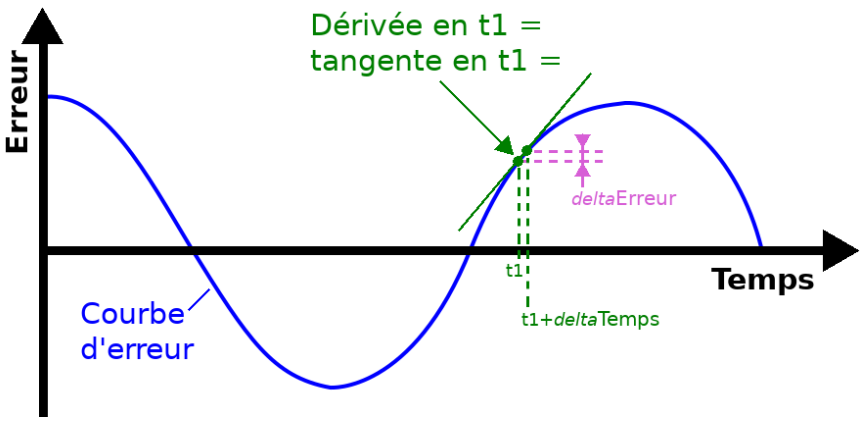
07RI16 – effet du « gain Dérivé » sur la composante proportionnelle

Il est assez facile d'approcher les gains Proportionnel puis Dérivés par essais erreurs lorsque les systèmes sont relativement stables. Il existe aussi des systèmes qui tendent facilement vers l'instabilité et dans pareil cas l'identification des gains est un exercice nettement plus périlleux.

Mise en place de la correction Dérivée

Visuellement, une dérivée en un point donné du temps correspond à la tangente (son coefficient angulaire) sur la courbe d'erreur.

Le graphique ci-dessous représente une courbe d'erreur avec le relevé de la dérivée en position t1.



07RI17 – appréciation de la Dérivée de l'erreur

Ce relevé dérivée, donc de la tangente en t1, se fait sous sa forme vulgarisée en prenant deux points très proches l'un de l'autre (en t1) et en traçant une droite passant par ces deux point. Nous obtenons ainsi une droite tangente à la courbe en ces points.

Le coefficient angulaire en t1 = dérivée en t1 = *delta*Erreur / *delta*Temps . Il ne manque plus que la multiplication par le **gain Dérivé** pour avoir l'élément correcteur supplémentaire.

Il se fait que l'algorithme du suiveur de ligne fonctionne en boucle fermée effectuant un contrôle d'erreur à chaque itération à raison de plusieurs dizaine d'itération par seconde.

La formulation du terme dérivé devient :

```
gainDérivé * deltaErreur / deltaTemps
gainDérivé * (ERROR – lastError ) / deltaTemps
```

Où l'erreur `ERROR` est déjà calculé pour la correction proportionnelle et `lastError` l'erreur calculée à l'itération précédente.

L'élément `deltaTemps` pourrait être calculé de la même façon que `deltaError` mais il se fait que le temps d'exécution de chaque itération sera constant (du moins assez pour être approximé comme une constante).

Etant donné que `gainDérivé / deltaTemps` représente une division de constantes produisant aussi une constante il est possible de remplacer `gainDérivé / deltaTemps` par `autreGainDérivé`.

La formulation du terme dérivé devient :

```
autreGainDérivé * (ERROR – lastError )
```

Le but du jeu est donc de trouver, par essai / erreur, la valeur de `autreGainDérivé` qui assure le retour le plus rapide sur la consigne (2500, ligne au centre du capteur de ligne).

Le terme dérivé sera le suivant :

```
6 * (ERROR – lastError )
```

Conclusion

Le formule de régulation PD (proportionnel + dérivé) est :

```
DIFF = (ERROR/4) + 6 * (ERROR–lastError )
```

où `lastError` est la valeur de `ERROR` calculée à l'itération précédente.

Ainsi équipé d'un régulateur PD, le Zumo Robot à la possibilité de tendre vers un suivit de ligne qui garde le Zumo sur la consigne (soit 2500 dans le cas du capteur de ligne).

4.3.Script suiveur de ligne

Voici le contenu du script permettant de transformer le Zumo Robot en suiveur de ligne.

Ce script est disponible dans le dépôt du projet sur le lien suivant :

https://github.com/mchobby/micropython-zumo-robot/blob/main/examples/line_follower.py

Le script fait moins de 40 lignes et inclus même quelques mécanisme de protection.

```
01: from zumoshield import ZumoShield
02: from machine import WDT
03: import time
04:
05: z = ZumoShield()
06:
07: MAX_SPEED = 100
08: last_error = 0
09:
10: def clamp( val, _min, _max ):
11:     return max(min(_max, val), _min)
12:
13: print( "Press Button to start calibration" )
14: z.buzzer.play(">g8>>c8")
15: z.button.waitForButton()
16: time.sleep(1)
17: z.ir_calibration( motors=True )
18: z.buzzer.play(">g8>>c8")
19:
20: try:
21:    wdt = WDT( timeout = 500 )
22:    while(True):
23:        wdt.feed()
24:        position = z.ir.readLineBlack()
25:
26:       error = position -2500
27:       speed_diff = (error/4) + (6*(error-last_error))
28:       last_error = error
29:
30:        m1Speed = MAX_SPEED+speed_diff
31:        m2Speed = MAX_SPEED-speed_diff
32:        m1Speed = clamp( m1Speed, 0, MAX_SPEED )
33:        m2Speed = clamp( m2Speed, 0, MAX_SPEED )
34:
35:       z.motors.setSpeeds(m1Speed,m2Speed)
36: finally:
37:     z.motors.stop()
```

Voici quelques détails utiles permettant d'en comprendre le fonctionnement :

•Lignes 1 à 3: import des modules et classes nécessaires. La classe WDT permet de créer un *Watch Dog* (chien de garde), élément qui sera abordé en temps utile.

•Ligne 5 : création de l'instance z du ZumoShield. Le ZumoShield permet d'accéder à toutes les fonctionnalités du Zumo Robot. Les moteurs sont volontairement mis à l'arrêt lors de l'initialisation du ZumoShield, ce qui est particulièrement utile si le ZumoRobot est réinitialisé ou redémarré inopinément.

•Ligne 7 : constante MAX_SPEED indique la vitesse de référence du Zumo (valeur entre 0 et 400)

•Ligne 8 : la variable last_error est destinée à recevoir l'erreur calculée lors du cycle précédent (utilisé pour le calcul de la dérivée de l'erreur).

•Lignes 10 et 11 : définition de la fonction `clamp( val, _min, _max )` permettant de borner une valeur `val` entre un minimum et un maximum. Ainsi, `clamp( 15 , 11, 22 )` retourne 15 puisque la valeur reste dans les limites, alors que `clamp( 8, 11, 22 )` retourne 11 puisque la valeur minimale autorisée est de 11. Enfin `clamp( 30, 11, 22 )` retourne 22 puisque la valeur maximale autorisée est de 22.

•Lignes 13 et 14 : Envoi d'un message (sur REPL) et signal sonore pour signaler l'attente de l'action utilisateur. C'est le moment de placer le Zumo Robot au dessus de la ligne pour l'étalonnage à venir.

•Lignes 15 et 16 : Attendre que l'utilisateur presse et relâche le bouton utilisateur du Zumo. Lorsque le bouton est relâché, une pause supplémentaire de 1 seconde laisse le temps à l'utilisateur de retirer sa main.

•Ligne 17 : Calibration motorisée du détecteur de ligne (cfr. Tester – Détecteur de ligne).

•Ligne 18 : Second signal sonore indiquant la fin de la calibration et le début du suivit de ligne prenant place dans la boucle infinie des lignes 22 à 35.

•Lignes 20, 36 et 37 : la section `try...finally` assure l'exécution de la section `finally` même en cas d'erreur (comme la levée d'une exception). Dans le cas présent, la section `finally` arrête les moteurs du Zumo en ligne 37.

En temps normal, la boucle infinie des lignes 22 à 35 ne permet pas d'atteindre la ligne 37. Cependant, durant une session REPL il est possible d'arrêter ce script avec un [CTRL]+C , ce qui produit la levée d'une exception `KeyboardInterrupt` . Dans ce cas, il est utile d'arrêter les moteurs.

•Ligne 21 : Création de l'instance du chien de garde (WatchDog, classe `WDT`). Un WatchDog est un élément matériel qui, une fois activé, doit être contacté avtn la fin d'intervalles réguliers (fixé ici à 500 millisecondes). Si cela n'est pas fait dans le temps imparti, le WatchDog réinitialise/redémarre le microcontrôleur.

•Ligne 22 : Boucle `while` infinie qui répète inlassablement le bloc d'instructions des lignes 23 à 35.

•Ligne 23 : `wdt.feed()` réinitialise le compteur du chien de garde à 0 ms. Cela est fait à chaque tour de boucle et signifie donc que le restant du bloc doit être exécuté endéans les 500ms.

La méthode utilisée pour contacter le chien de garde est `feed()` qui signifie « nourrir ». Il est amusant de constater qu'il faut « nourrir le chien de garde » pour le garder sous contrôle.

•Ligne 24 : Détecter la `position` de la ligne noire sous le détecteur de ligne. La variable position contient une valeur numérique entre 0 et 5000 (Cf. Tester – Détecteur de ligne).

•Ligne 26 : Calcul de l'erreur par rapport à la consigne qui est de 2500, le but étant de garder la ligne noire au centre du capteur de ligne.

•Ligne 27 : Calcul de la correction `speed_diff` (différence de vitesse) à l » à l'aide d'un régulateur PD (proportionnel et dérivé). Pour rappel l'élément dérivé utilise la différence entre l'erreur actuelle et l'erreur passée (du cycle précédent, stocké dans la variable `last_error`).

•Ligne 28 : Maintenant que la correction `speed_diff` est connue, la valeur de l'erreur `error` identifié durant ce cycle doit mémorisé pour être utilisée lors du prochain cycle. Ce que fait l'instruction `last_error = error` .

•Lignes 30 et 31 : Calcul de la nouvelle vitesse pour chacun des moteurs en appliquant la correction calculée sur la vitesse par défaut (`MAX_SPEED`).

•Lignes 32 et 33 : Ces instructions s'assurent que la vitesse de chaque moteur est bien bornée entre les valeurs 0 et 400.

•Ligne 35 : Applique les nouvelles vitesses sur les moteurs du Zumo Robot. Après cette instruction, la boucle `while` redémarre une nouvelle itération à partir de la ligne 35.

4.4.Encore plus

Ce script peut être amélioré de plusieurs façons :

1.Injecter les données de calibration dans le script afin d'éviter la phase de calibration (cf. Tester – Détecteur de ligne, voir point « calibration et lecture calibrée »).

2.Supporter les croisements (perpendiculaires)

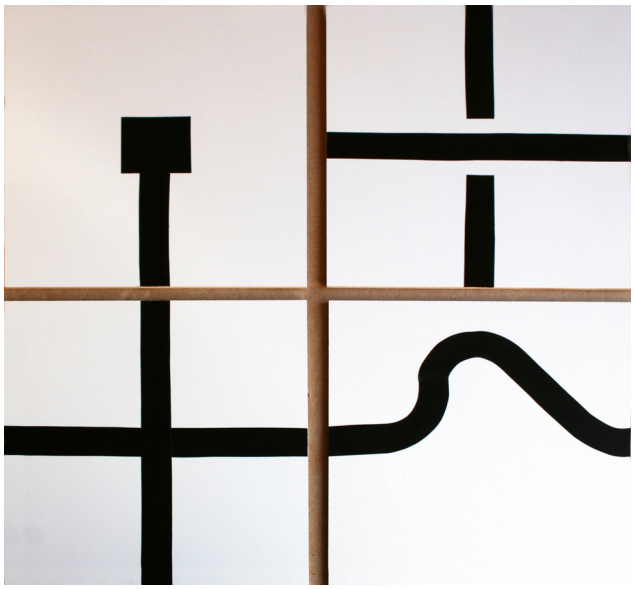
3.Supporter les courtes interruptions de parcours (40mm)

4.Augmenter la vitesse de déplacement (de 100 → 200).

5.Adapter automatiquement la vitesse au type de parcours

6.Prévoir une fin de parcours (un trait spécial 40 x 45mm)

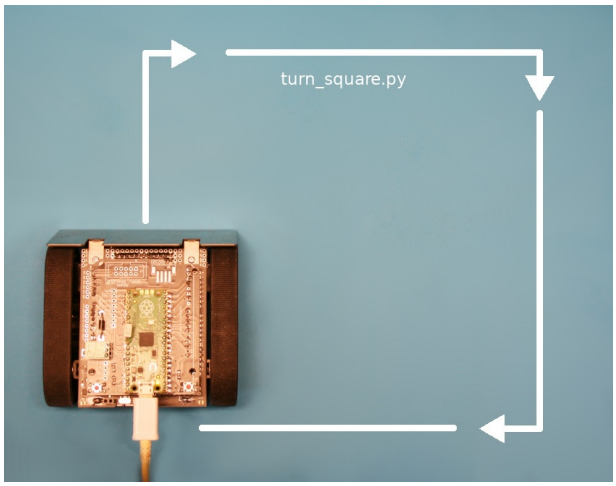
7.Pouvoir négocier un tournant serré



07RI18 – divers type de parcours envisageable

5.Tourner en carré

Effectuer une rotation à angle droit n'est pas une action fortuite comme cette section le démontrera.



07RI24 – tourner à angle droit avec le Zumo

5.1.Approche néophyte

De prime abord, il est tentant de penser que tourner à une vitesse donnée pendant un temps donné permettrait de facilement atteindre le but souhaité (par exemple à la vitesse de 100 pendant une 1,2 sec).

Cette approche présente pourtant un défaut majeur : la vitesse réelle de rotation dépend aussi du niveau de charge des piles.

Ce que pour assurer un résultat correct, il faut utiliser un procédé de permettant de contrôler l'angle de rotation du robot (ou des moteurs) !

5.2.Approches envisageables

Les capteurs disponibles sur le Zumo Robot permettent d'envisager les approches suivantes :

1.Utiliser le magnétomètre et le nord magnétique pour connaître l'orientation du robot.

2.Utiliser du gyroscope

3.Utiliser un marquage au sol et le détecteur de ligne.

Le magnétomètre sera l'approche utilisé dans cette section pour réaliser les rotations.

Limite des approches

Les trois approches ci-dessus présentent cependant des limites techniques limitant leurs efficacités respectives :

•La magnétomètre représente l'option la plus prometteuse. Il faut cependant admettre que la détection du nord magnétique se fait dans environnement électromagnétique assez bruyant et peut donc présenter une erreur de plusieurs degrés (compter ± 3° dans les cas les plus défavorables).

•Le gyroscope permet quand à lui de détecter la vitesse de rotation (en degrés par secondes). En réalisant de nombreux relevés multipliés par les écarts de temps entre les relevés, il est possible de calculer la rotation effectuée (l'angle). Cette approche requière un étalonnage très rigoureux et un filtrage tout aussi rigoureux des écarts (imprécision de lectures). La méthode requière par ailleurs une acquisition à haute fréquence car la vitesse de rotation n'est pas constante. Les méthodes de calcul doivent être tout aussi performantes.

•Le marquage au sol bien étudié (maillage/lignage/quadrillage) peut être une solution intéressante à condition que le Robot Zumo se déplace parallèle ou perpendiculaire à celui-ci.

5.3.L'approche technique idéale

L'approche idéale pour résoudre ce type de problème consiste en l'utilisation de moteur avec encodeur rotatif.

Cependant, cette option n'est pas disponible sur le Robot Zumo.

07RI28 – encodeur rotatif sur moteur continu

L'encodeur rotatif se présente sous la forme d'un disque magnétique solidaire de l'axe moteur alternant pôle nord et sud au dessus d'une carte solidaire du corps moteur. La carte est équipé de deux capteurs à effet hall (sensible au champ magnétique) permettant ainsi de détecter les changements N/S, S/N, N/S.

En comptant les successions de changement magnétique, à raison de 64 par rotation dans le cas présent, il est possible de connaître avec précision le déplacement de l'axe moteur. En connaissant le nombre de tours ou portion de tours de l'axe moteur il est possible de déduire la distance parcourue par la roue qui y est connectée.

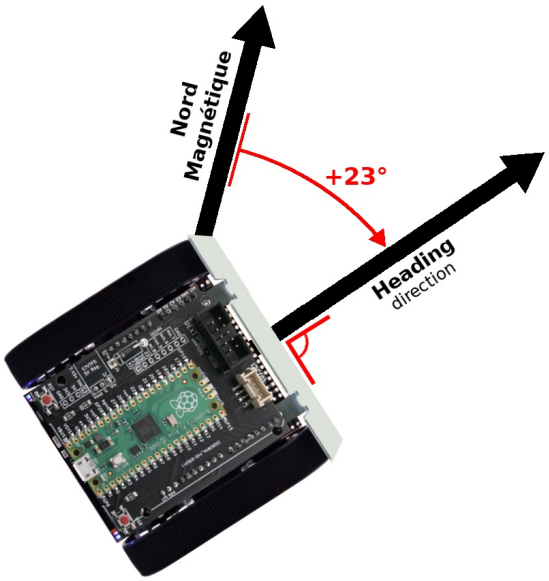
Il est aussi possible de coordonner les mouvements de plusieurs moteurs en vue de réaliser un mouvement/déplacement .

5.4.Principe de fonctionnement

Le script utilise la classe `Compass` qui s'appuie sur le magnétomètre du Zumo Robot pour :

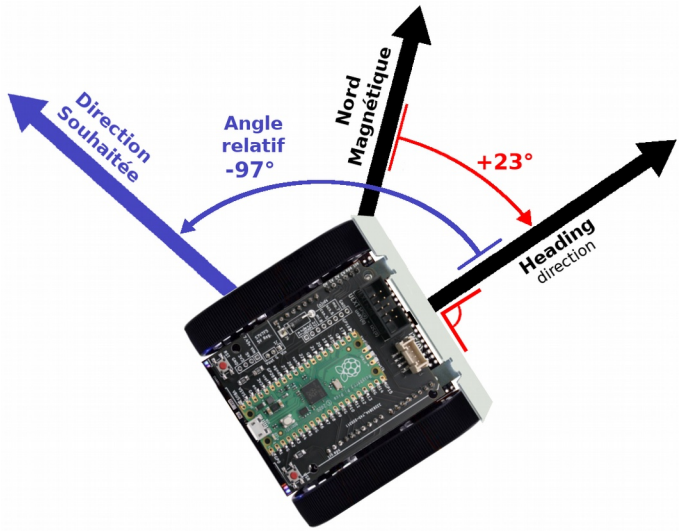
1.calibrer les données du magnétomètre afin de faciliter la détection du Nord Magnétique.

2.d'estimer la position angulaire du Zumo, dit *heading*, par rapport au Nord Magnétique (l'angle par rapport au Nord Magnétique).



07RI26 – Angle par rapport au Nord Magnétique

3.d'évaluer l'angle relatif (par rapport à *heading*) dont il faut encore tourner le Zumo pour atteindre la position angulaire souhaitée.



07RI27 – Angle relatif

L'angle relatif par rapport à la direction souhaitée permet de savoir dans quel sens iles moteurs doivent tourner.

Un angle relatif négatif produit une rotation à gauche avec

```
z.motors.speeds(-1*vitesse, vitesse )
```

Un angle relatif positif produit une rotation à droite avec :

```
z.motors.speeds(-1*vitesse, vitesse )
```

Enfin, plus l'angle relatif est grand et plus la vitesse de rotation peut être élevée. Lorsque l'angle relatif diminue, il convient de réduire la vitesse de rotation pour ne pas passer au dessus de la direction souhaitée.

Seuil de détection

Pour éviter le dépassement assuré durant la rotation, l'algorithme n'attend pas d'avoir dépasser l'objectif pour arrêter les moteurs. Il s'arrête dans une fourchette proche de l'objectif.

Une technique fort commune consiste à utiliser une valeur de seuil (dit *Thresold* en anglais) pour déterminer si l'objectif est atteint. Dans le cas du script, un seuil de ±5 degrés est appliqué. Ce qui signifie que le Zumo peut arrête sa rotation lorsque l'angle relatif est dans la fourchette -5° à +5°.

Cela permet aussi de contre-balancer l'imprécision de l'angle retourné par `heading()` lors de la détection du Nord Magnétique. Les relevés réalisés durant les tests du magnétomètre ont démontrés une variabilité de l'ordre de ± 2° à ± 3° (cf. Tester – Centrale inertielle, voir point « lecture boussole »).

Cumul des erreurs

Entre l'imprécision des lectures (variabilité) du Nord magnétique et déduction de l'angle `heading()` et l'arrêt de la rotation dans une fourchette de ± 5°. L'erreur sur le positionnement peut atteindre pas loin de 8°.

Au bout de multiples rotations à 90° les erreurs successives cumulée peuvent facilement atteindre 20 à 25° (voir plus dans des cas vraiment défavorables).

En résumé certains angle peuvent paraître bien droit (bien à 90°) alors que d'autres paraîtront modérément faussés. Après 4 rotations, il est fort à parier que le Zumo ne soit pas exactement à la position attendue.

5.5.Script qui tourne en carré

Voici le contenu du script `turn_square.py` permettant au Zumo Robot d'utiliser son magnétomètre comme boussole afin de tourner à angle droit (ou presque).

Ce script est disponible dans le dépôt du projet sur le lien suivant :

https://github.com/mchobby/micropython-zumo-robot/blob/main/examples/turn_square.py

```
01: from zumoshield import ZumoShield
02: from zumoimu import ZumoIMU, Compass
03: from micropython import const
04: import time
05: import math
06:
07: SPEED           = const( 200 )
08: TURN_BASE_SPEED = const(  80 )
09: CALIBRATE_SPEED = const( 120 )
10:
11: z = ZumoShield()
12: imu = ZumoIMU( z.i2c )
13: compass = Compass( imu )
14:
15: print( "Starting calibration" )
16: z.play_blip()
17: z.button.waitForButton()
18:
19: z.motors.setSpeeds( CALIBRATE_SPEED,-1*CALIBRATE_SPEED )
20: compass.calibrate()
21: z.motors.stop()
22:
23: print( "Press the button to START" )
24: z.play_blip()
25: z.button.waitForButton()
26:
```

```
27: heading = 0.0
28: target_heading = None
29: relative_heading = 0.0
30: speed = 0
31:
32: while True:
33:   if target_heading == None:
34:     target_heading = compass.average_heading()
35:
36:   heading = compass.average_heading()
37:   rel_heading = compass.relative_heading(heading,
                          target_heading)
38:
39:   print("[Degrees] Target: %4i  ,Actual: %4i  ,Diff: %4i"
               % (target_heading, heading, rel_heading) )
40:
41:   if abs(rel_heading) < Compass.DEVIATION_THRESHOLD :
42:     z.motors.setSpeeds(SPEED, SPEED)
43:     print("   Go Straight")
44:     time.sleep(1)
45:     z.motors.stop()
46:     time.sleep_ms(100)
47:
48:     target_heading = compass.average_heading() + 90 % 360
49:   else:
50:     speed = int(SPEED*rel_heading/180)
51:     if speed < 0:
52:       speed -= TURN_BASE_SPEED
53:     else:
54:       speed += TURN_BASE_SPEED
55:
56:     z.motors.setSpeeds(speed, -speed)
57:     print("   Turning")
```

Voici quelques détails concernant le fonctionnement du script :

•Lignes 1 à 5: Import des modules et classes nécessaires. La classe `Compass` permettra d'utiliser le magnétomètre comme boussole.

•Ligne 7 : Définition de la constante `SPEED`, vitesse utilisée lors d'un déplacement en ligne droite. Le fonction `const()` de la bibliothèque `micropython` permet de définir l'information en mémoire de telle sorte qu'elle consomme le moins de mémoire possible.

•Ligne 8 : Définition de la constante `TURN_BASE_SPEED`, vitesse de référence utilisée durant la rotation du Zumo Robot.

•Ligne 9 : Définition de la constante `CALIBRATE_SPEED` définissant la vitesse de rotation du Zumo durant la phase de calibration du magnétomètre.

•Lignes 11 et 12 : Création de l'instance du `ZumoShield` (variable `z`) et de la centrale inertielle (variable `imu`).

•Ligne 13 : Création de la boussole, instance de `Compass` (variable `compass`). Comme la boussole exploite les données du magnétomètre de la centrale inertielle, la variable `imu` est passée en paramètre lors de la création de la boussole.

•Lignes 15 à 21 : Effectue la phase de calibration du magnétomètre. Cette calibration permet de calculer les valeurs minimales et maximales sur les axes X et Y. Cette calibration permettra ensuite de réaliser des relevés d'amplitude magnétiques entre 0 et 100 % sur ces deux axes.

•Lignes 15 à 16 : Message texte et audio affichant le début de la calibration.

•Ligne 17 : Script placé en attente jusqu'à ce que l'utilisateur presse le bouton utilisateur.

•Ligne 19 : Le robot Zumo doit tourner sur lui-même durant la calibration afin de pouvoir repérer les minimas et maximas sur les axes X et Y. La multiplicateur -1 inverse le sens de rotation du moteur droit.

•Ligne 20 : la méthode `calibrate()` effectue une rafale de 70 relevés sur le magnétomètre avant de terminer son exécution.

•Ligne 21 : La calibration terminée, les moteurs du Zumo sont arrêtés.

•Ligne 23 et 24 : C'est ici que commence le corps du script. Le message texte et sonore invite l'utilisateur à presser le bouton utilisateur pour démarrer le mouvement en carré.

•Ligne 25 : Attente de la pression du bouton utilisateur pour passer à la suite du script.

•Ligne 27 à 30 : définition des différentes variables nécessaires au bon fonctionnement de l'algorithme.

□`heading` : variable contenant la direction actuel du Zumo par rapport au Nord Magnétique. Il s'agit d'un angle entre 0 et 360° (sens horlogique positif). Cette valeur est constamment mise-à-jour à l'aide de la boussole (voir objet `compass`).

□`target_heading` : direction à atteindre à l'aide d'une rotation. Angle par rapport au Nord Magnétique exprimé en degrés. Cette valeur est initialisée à None pour indiqué qu'elle n'est pas initialisée.

□`relative_heading` : angle relatif dont le Zumo Robot doit tourner pour atteindre la direction indiquée par la variable `target_heading`. Une valeur négative indique qu'il faut tourner à gauche, une valeur positive indique qu'il faut tourner à droite. A zéro ou dans la fourchette du seuil (± 5°), la rotation doit cesser.

□`speed`: vitesse moteur durant la rotation. Valeur adaptée en fonction de l'angle de rotation qu'il faut encore effectuer. Vitesse élevée proche de `TURN_BASE_SPEED` lorsque l'angle relatif est proche de 180°. Vitesse proche de zéro lorsque l'angle relative s'approche de 0 (donc presque orienté dans la bonne direction).

•Ligne 32 : début de la boucle infinie `while True` dont le but est (1) d'avancer pendant une seconde puis (2) tourner à 90° sur la droite avant de (3) reprendre au point 1.

L'exécution de la boucle `while` s'étend jusqu'à la fin du script. La meilleure façon d'arrêter celui-ci est de presser le bouton Reset qui redémarre le script (et donc arrête les moteurs à l'initialisation du `ZumoShield`).

•Lignes 33 et 34 : s'il n'y a pas encore d'angle de rotation à atteindre (`target_heading`) alors celui-ci est initialisé avec l'orientation actuel du Zumo. Une autre façon de dire « le Zumo est aligné sur l'angle souhaité ».

Pour rappel, la méthode `compass.average_reading()` effectue une dizaine de lecture d'orientation sur le magnétomètre avant d'effectuer un calcul de moyenne sur les échantillons.

•Ligne 36 : lecture de l'orientation du Zumo et mémorisation de l'angle dans la variable `heading`.

•Ligne 37 : calcul de l'angle relatif `rel_heading` entre l'angle à atteindre (`target_heading`) et l'angle actuel du Zumo (`heading`).

↘ *Au premier tour d'exécution (à l'initialisation), l'angle relatif est de 0 degrées.*

•Ligne 39 : affichage de l'angle souhaité (à atteindre), l'angle actuel et la différence entre les deux (reste de la rotation à effectuer). Respectivement `target_heading`, `heading` et `rel_heading`.

•Lignes 41 à 57 : divisé en deux parties logiques :

□Lignes 42 à 48 : exécuté lorsque l'angle relatif est inférieur au seuil de ± 5° (la fonction `abs(rel_heading)` permet d'éliminer le signe de `rel_heading` . Le robot étant aligné sur l'angle souhaité, c'est le moment d'avancer et de calculer le prochain angle de rotation.

□Ligne 50 à 57 : le Zumo n'a pas encore atteint l'angle souhaité, il est donc toujours en cours de rotation. Calculer la vitesse de rotation en fonction de l'angle restant à parcourir.

•Ligne 42 à 45 : faire avancer le robot en ligne droite pendant 1 seconde puis arrêt moteur.

•Ligne 46 : attendre 1/10 seconde avant de commencer le relevé sur le magnétomètre.

•Ligne 48 : calculer le nouvel angle à atteindre (`target_heading`) comme suit :

□Relever l'angle actuel à l'aide de `compass.average_heading()`

□Ajouter 90°

□Effectuer la division modulaire par 360 (`% 360`) pour maintenir le résultat final dans la fourchette 0 et 359.

↘ *Au prochain tour de la boucle `while`, la différence entre `heading` et `target_heading` est supérieur au seuil de 5° et c'est la section des lignes 50 à 57 qui sera exécuté (contrôle de la vitesse de rotation).*

•Ligne 50 : Calcule la vitesse de rotation `speed` comme étant une proportion de l'angle restant à parcourir par rapport à 180°. Si l'écart d'angle était de 180° alors la vitesse serait de 200 (valeur de la constante `SPEED`). Pour un écart de 90°, la vitesse est réduite à 200*90/180=100. Lorsque l'écart se réduit et atteint 30°, la vitesse est réduite à 200*30/180 = 33,3

↘ *Le lecteur ayant explorer la commande moteur sait qu'une vitesse de 33 sur un maximum de 400 (donc un cycle utile PWM de 8%) ne permet pas à la plateforme Zumo de se mouvoir.*

•Ligne 51 et 54 : Afin de préserver un cycle PWM pour mouvoir le robot, la vitesse calculée `speed` est cumulée avec la vitesse de base de 80 (`TURN_BASE_SPEED`). La vitesse `speed` doit être cumulée dans le bon sens de son signe.

↘ *Le signe de speed peu changer si le robot dépasse la consigne auquel cas le robot doit revenir en arrière (la vitesse et le sens de rotation s'inversent).*

•Lign 56 : Application de la vitesse sur les moteurs du Zumo. Etant donné que le robot doit tourner sur lui-même, le sens de rotation de la chenille droite est inversée en multipliant la vitesse par -1.

•Ligne 57 : Message indiquant que le Zumo est entrain de tourner.

5.6.Encore plus

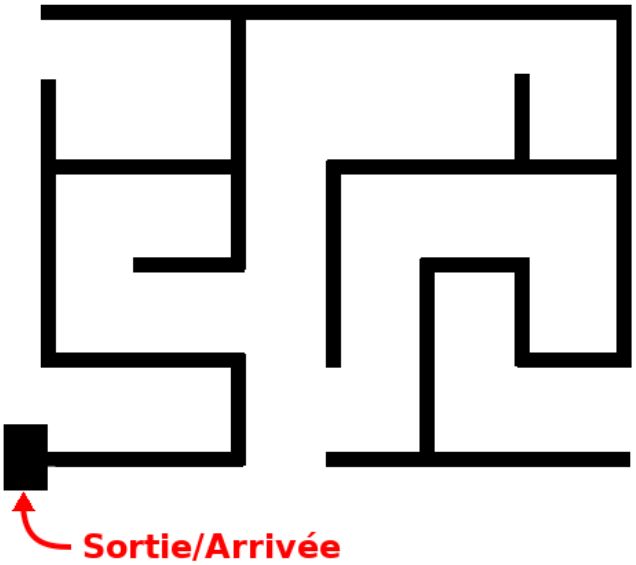
Il serait intéressant de réaliser des tests dans différents environnements pour tester l'influence des différentes structures environnantes sur la détection du Nord magnétique.

La méthode d'étalonnage du magnétomètre est relativement élémentaire et il se pourrait que les maxima/minima puissent être malencontreusement ratés. La recherche d'une méthode d'étalonnage plus complète serait appréciable (voir méthode `calibrate()` de la classe `Compass` dans le fichier `zumoimu.py`).

Adapter la technique pour réaliser la rotation sur des angles de 30, 60, 90 et 180°

6.Résolution de labyrinthe

La résolution d'un labyrinthe à ligne est un exemple vraiment très électrisant.



07RI28 – Labyrinthe pour ZumoRobot

Le principe consiste :

• Poser le Robot Zumo à l'une des extrémités du labyrinthe,

• A laisser le robot explorer le labyrinthe et trouver la cible de sortie (le carré).

Ensuite replacer le Robot Zumo à son point de départ et celui-ci se dirigera directement vers la sortie. Un algorithme permet de simplifier les « données de trajet » collectées durant l'exploration afin d'identifier une trajectoire directe vers la porte de sortie.

6.1.Constitution d'un labyrinthe

Le labyrinthe est constitué de sections de ligne noire sur fond blanc. Les sections de ligne font 15mm de large et 80mm de long (longueur allant jusqu'à 250mm).

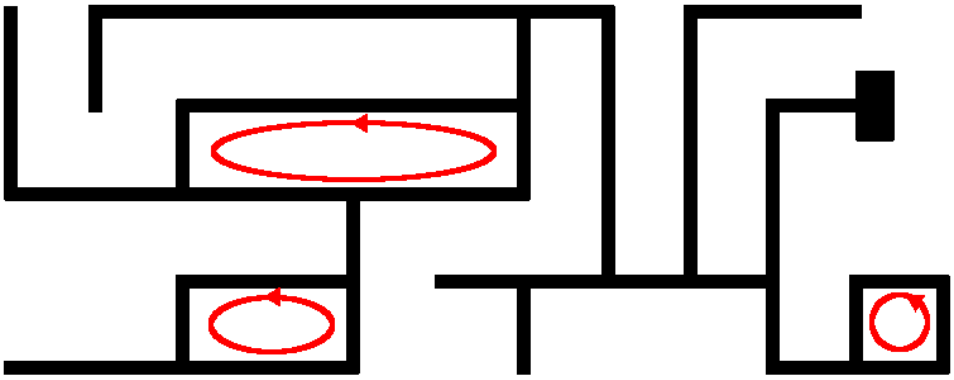
07RI29 – Autre type de Labyrinthe

Le labyrinthe contient des intersections de type : angle droit, en croix, en T ou en L.

Certaines des voies du labyrinthe terminent en cul-de-sac.

La porte/cible de sortie est constituée d'un rectangle de 100mm de large (pour couvrir tous les capteurs du détecteur de ligne) et de 60mm de long.

Le labyrinthe ne peut pas contenir de boucle fermée.



07RI30 – Labyrinthe avec boucle fermée.

6.2.Parcours du labyrinthe

La résolution du labyrinthe passe par la technique de « la main gauche sur le mur » pour parcourir le labyrinthe et permet toujours d'atteindre la fin du labyrinthe (si celui-ci ne contient pas de boucle).

Conformément à la technique de la « main gauche » :

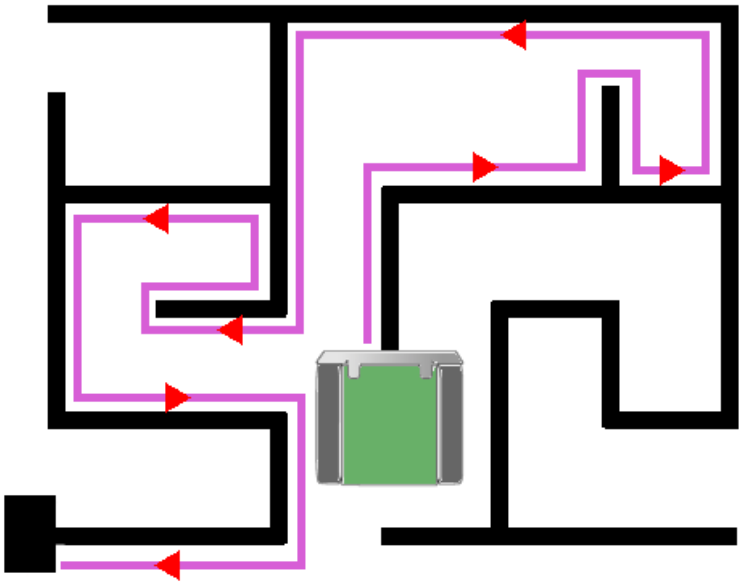
1.Toujours préférer un tournant à gauche plutôt que d'aller tout droit ou tourner à droite.

2.Toujours aller tout droit plutôt que tourner à droite.

3.Tourner à droite lorsqu'il n'y a pas d'autres choix.

Non indiqué, dans les règles, dans le cas d'un cul-de-sac, le Zumo fait demi-tour et reprend une marche avant. C'est l'équivalent indirecte de la 3ième règle, ce qui ramène le robot sur la voie qu'il bient d'emprunter.

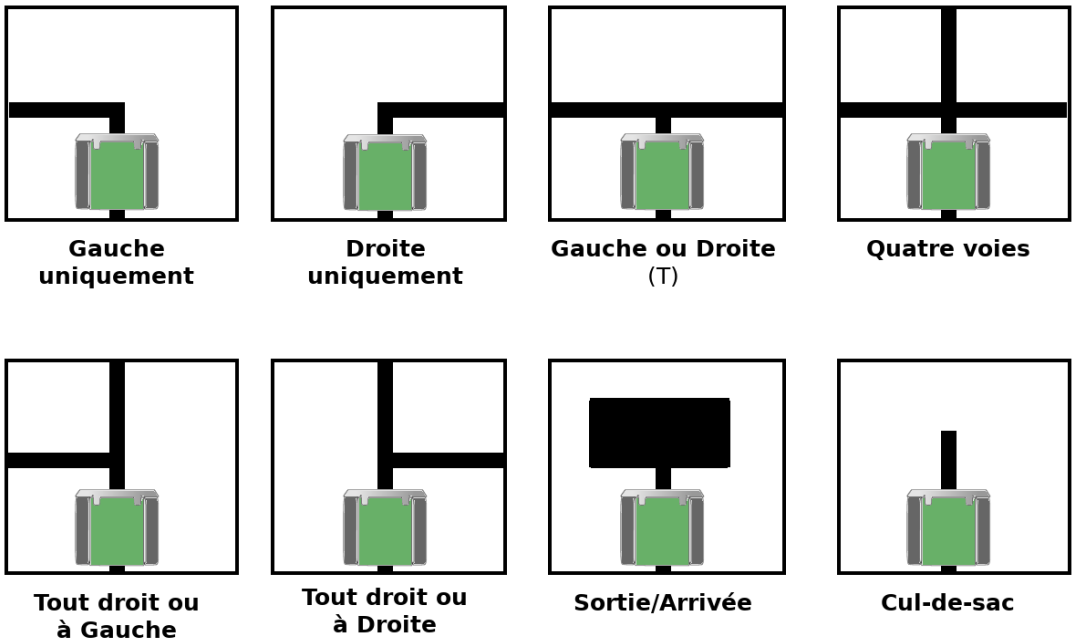
Le graphe ci-dessous présente le point de départ du Robot Zumo et le parcours suivit en suivant les règle édictées.



07RI28b – Parcours dans un labyrinthe

6.2.1.Les différents type de croisement

Durant la résolution du labyrinthe, 8 cas de figure se présente du point de vue du Robot.



07RI32 – 8 cas de figures du labyrinthe

Voici la description du comportement du Zumo suivant la technique de la « main gauche » et ce face aux différents cas de figure :

Cas de figure	Choix opéré	Action enregistrée	Description
Gauche	aucun	**L**	le Zumo n'a d'autre choix que de suivre

uniquement			la ligne, il tourne à gauche.
Droite uniquement	aucun	**R**	Comme le cas précédent, mais le Zumo tourne à droite.
Carrefour en T	Gauche	**L**	Prendre le tournant à gauche.
Quatre voies	Gauche	**L**	Prendre le tournant à gauche.
Tout droit ou à gauche	Gauche	**L**	Prendre le tournant à gauche.
Tout droit ou à droite	Tout droit	**S**	Poursuivre tout droit.
Sortie/ Arrivée	Arret	—	Fin de parcours. Le programme est prêt à retracer le parcours depuis le départ.
Cul-de-sac	Demi-tour	**B**	Faire demi tour sur place et poursuivre la résolution du labyrinthe.

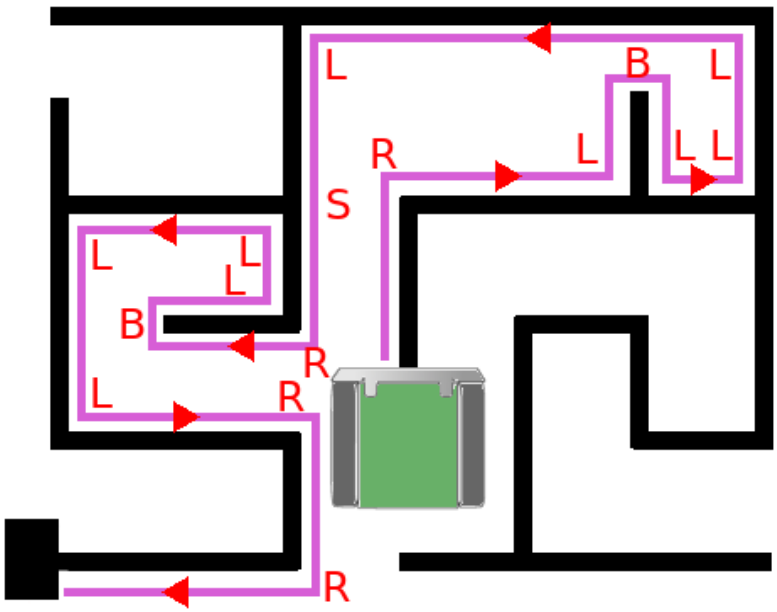
Les différentes actions possible sont :

- **L** : de l'anglais *Left* pour un tournant à gauche.
- **R** : de l'anglais *Right* pour un tournant à droite.
- **S** : de l'anglais *Straight* pour aller tout droit.
- **B** : de l'anglais *Back* pour revenir en arrière (faire demi-tour et avancer).

Durant la résolution du labyrinthe le script Python mémorise la suite des actions utilisées pour résoudre le labyrinthe. Cette suite s'appelle le chemin (dit *path* en anglais). Par la suite, parcourir le labyrinthe en appliquant les mêmes actions dans le même ordre permettra de retrouver la sortie (avec quelques simplifications).

Suivant la technique « de la main gauche » l'algorithme trouve toujours la sortie du labyrinthe.

Le graphique ci-dessous reprend reprend la résolution de labyrinthe précédemment rencontrée avec inclusion des actions effectuées.



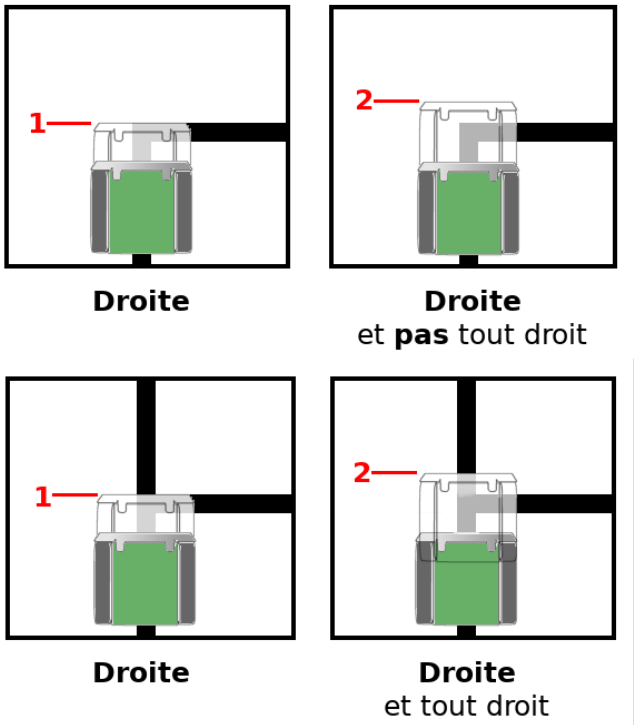
07RI33 – Résolution du labyrinthe + actions

Ce qui produit le chemin (*path* en anglais) RLBLLLLSRBLLLLRR

6.2.2.Détection correcte des croisements

Etant donné que le détecteur de ligne effectue sur une ligne, il existe forcement une ambiguïté lors de la détection des croisements impliquant une voie allant tout droit !

La détection des croisements doit donc se faire avec précaution avant de décider du chemin à suivre.



07RI35 - Ambiguïté de détection

Ainsi, dans l'exemple ci-dessus, un croisement présentant une voie « à droite uniquement » et une voie « à droite et tout droit » pourrait être erronément détecté comme un tournant « à droite uniquement ».

En effet, une fois arrivé en position 1, le détecteur de ligne détecte une voie sur la droite du détecteur pour les deux types de croisement. Cette détection sur la droite indique clairement que le Robot Zumo se trouve sur un croisement et celui-ci doit s'arrêter pour décider la voie à suivre.

Seulement voilà, le Robot Zumo ne dispose d'aucun moyen technique lui permettant de savoir que la voie se poursuit également en ligne droite.

Pour identifier correctement le type de croisement, il faut que le Robot s'avance encore (en position 2) pour vérifier si une ligne existe plus loin.

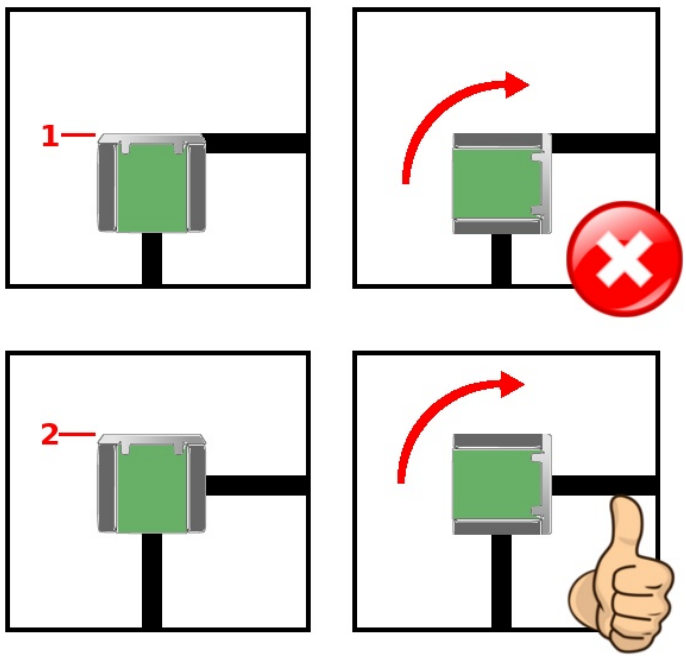
C'est seulement après cette exploration supplémentaire que l'algorithme sera en mesure de prendre la décision adéquate sur la voie à suivre.

La logique voudrait que le robot fasse une marche arrière pour reprendre sa place en position 1, le point suivant démontrera qu'il y a une position 2 optimale pour la détection et la rotation du Robot Zumo.

6.2.3.Croisement et rotation

Le point précédent mettait en évidence la nécessité d'avancer le robot plus avant pour détecter une ligne en amont du capteur de ligne.

Une avancée de 50mm (1,96 pouces ou 2 pouces), la moitié de la longueur du Zumo permet de placer le corps du robot au dessus du croisement. Ce qui permet, après une rotation, d'avoir le détecteur de ligne au dessus de la ligne après la rotation du robot.



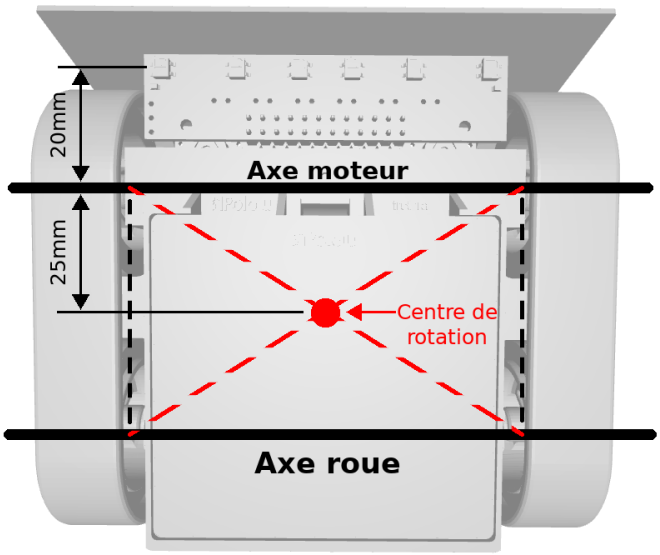
07RI36 – Bonne position du Zumo pour rotation

Dans l'image ci-dessus, si le robot effectue une rotation à l'endroit même où la ligne est détectée (position 1) alors le le robot finira totalement désaxé par rapport à la ligne.

Si le robot se déplace en position 2 (plus avant de 50mm par rapport à la position 1) alors le robot se trouvera en bonne position. Pour reprendre sa route. Le déplacement en position 2 permet également de détecter l'éventuellement ligne en amont.

Pour être précis, la distance parcourue doit permettre de placer le centre de rotation de la propulsion au dessus du croisement sur lequel il faut tourner. De la sorte, le robot tourne sur lui-même exactement au dessus du capteur.

Le graphique ci-dessous permet de visualiser et calculer la distance exacte dont-il faut avancer le Robot Zumo pour atteindre la « position 2 » idéale et amener le centre de rotation au dessus du carrefour.



07RI37 – Calcul de l'avancement nécessaire

L'entre-axe du robot faisant 50mm, il faudra parcourir 25mm à laquelle s'ajouter la distance de 20mm séparant le détecteur de ligne (premier élément qui détecte la carrefour) de l'axe moteur. Cela représente 45mm, pas très loin des 50mm annoncés ci-avant. Un imprécision sans réelle conséquence à cause des limitations techniques présentées ci-dessous.

6.2.4.Mesurer l'avancée

Cette avancée de 50mm après le croisement n'est pas une opération triviale ! Cette avancée de 50mm correspond à (2 pouces dans le système Impériale).

Le Zumo robot ne dispose pas d'outil de mesure permettant de quantifier l'avancée du robot. Il n'y a pas de capteur le permettant. Par conséquent, il faut donc l'estimer à partir des éléments disponibles.

Mesurer l'avancée de 50mm peut se faire en estimant le temps pendant lequel le robot doit se déplacer à une vitesse donnée. En considérant une charge des piles optimale à 90-100 %, une vitesse de 200 et des micro-moteurs ayant un rapport de 75:1 le robot avance d'une distance d'on l'ordre de grandeur est prévisible.

Le projet de résolution de labyrinthe est issu des travaux de Pololu USA où le système d'unités Impériales y est encore d'usage de nos jours ! Par conséquent c'est le « Pouce » (Inch en anglais) qui est utilisé comme unité de référence pour la mesure de longueur.

Ainsi, le code de l'exemple `maze_solver.py` utilise une constante `INCHES_TO_ZUNITS = 17142.0` (valeur par défaut) qui est une mesure fictive permettant de convertir la vitesse moteur en distance.

Ainsi, dans l'exemple `motors.setSpeeds(200,200)` la valeur 200 représente la vitesse en Zunits/seconde. Ainsi, au bout de 1,5 secondes (soit 1500ms), le Zumo Robot peut parcourir la distance de :

`(Temps_ms * Vitesse)/Inches_to_ZUnits = (1500 * 200) / 17142 = 17,5 pouces`

Soit une distance 44,45 cm (un peu optimiste mais raisonnable).

La fonction overshoot()

A l'opposé du point précédent qui calcule la distance parcourue pour un temps donné, la fonction `overshot()` fait l'opposé en évaluant le temps nécessaire au parcours d'une distance donnée.

La fonction `overshoot( distance_inches )` permet de calculer le temps d'activation nécessaire pour les moteurs afin de dépasser un croisement.

Un fois calculé, les moteurs seront activés pendant le nombre de millisecondes mentionnées en retour par la fonction `overshoot()`.

Voici l'implémentation de la fonction overshoot :

```
def overshoot( distancce_inch ):
   # Retourne des millisecondes
   return int((INCHES_TO_ZUNITS * distancce_inch) / SPEED)
```

La constante INCHES_TO_ZUNITS

La fonction `overshoot()` dépendant de la constante `Inches_to_ZUnits` .

Une valeur inappropriée de `Inches_to_ZUnits` peut gravement nuire au bon fonctionnement de la résolution de labyrinthe.

Augmenter/diminuer la valeur de `Inches_to_ZUnits` permet de tenir compte :

•**Du type de piles** : rechargeable ou alcaline, la tension nominale est différente

•**De la charge des piles** : les moteurs ralentissent au fur et a mesure que leur tension chute,

•**Du rapport du réduction des moteurs** : il est en effet possible de changer les moteurs du Zumo.

Adapter la distance parcourue en fonction des conditions de charge peut alors se faire en adaptant la valeur de la constante `Inches_to_ZUnits` :

•Augmenter `Inches_to_ZUnits` : augmente la distance parcourue.

•Réduire `Inches_to_ZUnits` : diminue la distance parcourue.

Calibrer INCHES_TO_ZUNITS

Le script `test_itoz.py` permet de tester l'avancée (*overshoot*) sur un parcours de test constitué d'une ligne droite présentant un virage à droite.

Après une phase de calibration du capteur de ligne, l'utilisateur doit presser une première fois le bouton utilisateur pour que le Zumo avance jusqu'au carrefour et s'arrête.

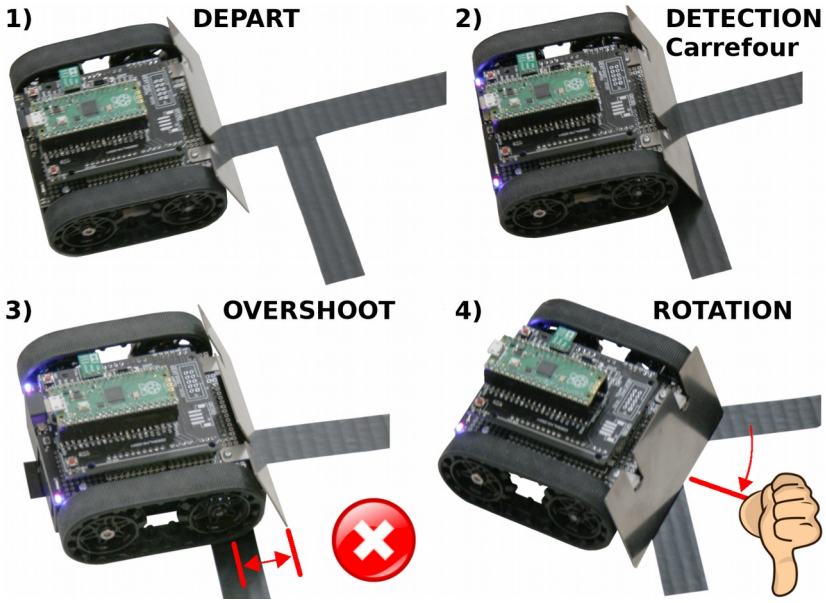
Appuyer une deuxième fois sur le bouton overshoot active le calcul de l'*overshoot* puis arrête une nouvelle fois le robot.

Cela permet d'évaluer la justesse du dépassement (*overshoot*) par essai-erreur et, ainsi, d'ajuster la valeur `Inches_to_ZUnits` .

Presser une dernière fois sur le bouton utilisateur pour que le robot entame une rotation à droite. De la sorte il est aussi possible d'évaluer le placement du Zumo après une rotation au carrefour.

L'écriture de cet ouvrage aura nécessité une modification de Inches_to_zunits de 17142.0 à 41000.0 !

Dans le cas ci-dessous, le script `test_itoz.py` montre que `Inches_to_ZUnits=17142.0` produit un dépassement (*overshoot*) insuffisant du carrefour, ce qui se traduit par une rotation également insuffisante !

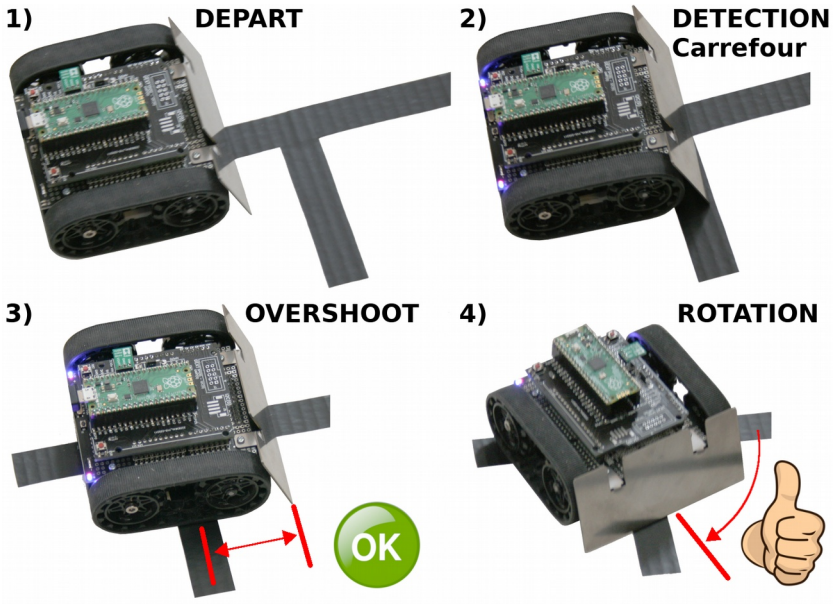


07RI38 – conséquence d'un overshoot insuffisant.

En effet, le capteur centrale du détecteur de ligne est déjà activé alors même que les capteurs à l'extrême gauche n'ont pas encore quitté l'autre voie.

Le Robot Zumo ne manquera pas de présenter un comportement difficilement prévisible dans pareille conditions.

Après des essais avec différentes valeurs, il s'avère que `Inches_to_ZUnits=41000.0` produit un dépassement suffisant pour produire une rotation adéquate dans le carrefour.



07RI39 – overshoot/dépassement suffisant

Un dépassement suffisant après le carrefour assure que le détecteur de ligne aura quitté la première voie lorsque le capteur centrale sera activé par la seconde voie.

6.3.Simplification du parcours

Reparcourir le même chemin pour trouver la sortie n'apporte rien par rapport au parcours initial.

Ce qui serait intéressant, c'est d'aller plus rapidement possible à la fin du labyrinthe.

Eviter un cul-de-sac est un exemple évident d'optimisation du parcours.

Voici la présentation de deux techniques permettant de simplifier le parcours du Zumo afin de réduire le chemin (*path*) à parcourir pour retrouver la sortie.

Il est important de rappeler que l'algorithme de résolution utilise le principe de la main gauche pour trouver la sortie d'un labyrinthe. Cela élimine de facto toute une série de situations.

6.3.1.Simplification élémentaire

L'exemple initial du résolveur de labyrinthe est une transposition du code Arduino (code écrit en C).

La programmation C est moins intuitive que Python et la manipulation de structure complexe en C peut nécessiter une gymnastique intellectuelle parfois difficile.

Voici les détails de la technique employée dans `maze_solver.py`.

Simplification systématique

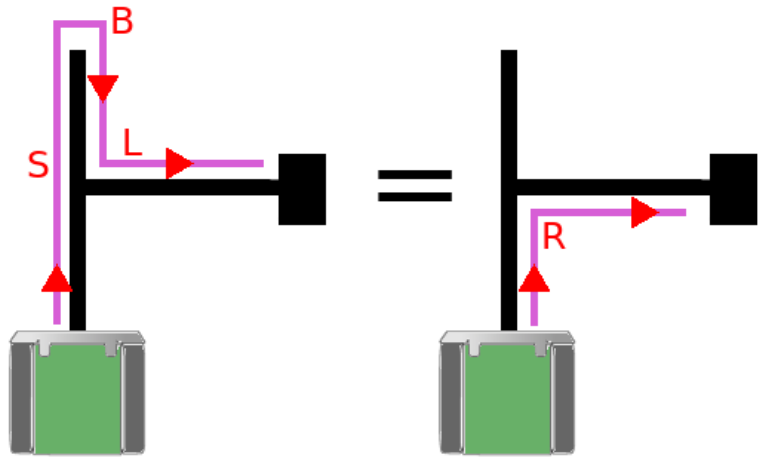
A chaque fois que le Zumo Robot rencontre un embranchement et fait une rotation, il ajoute un élément R,L,S ou B dans le chemin parcouru.

C'est le moment idéal pour pratiquer une inspection et simplification systématique.

Les derniers éléments du chemin parcourus seront inspectés en vue d'une tentative de simplification !

Quand opérer la simplification ?

L'exemple du cul-de-sac repris ci-dessous montre clairement qu'il faut un minimum de 3 éléments dans le chemin pour envisager une simplification.



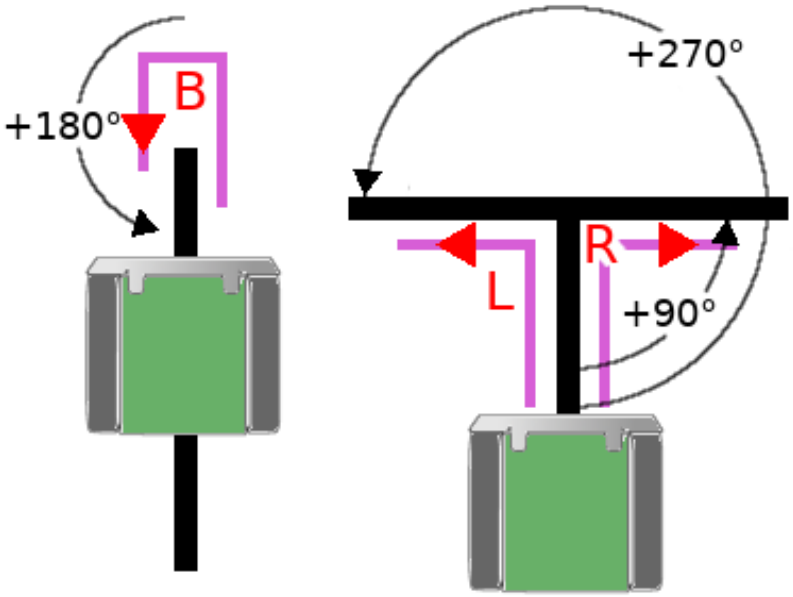
07RI40 – Simplification cul-de-sac

Le programme inspectera donc les trois dernières décisions pour opérer une simplification.

Par ailleurs, si la dernière décision est un B (retour en arrière) alors il n'y a pas de simplification immédiate, il faudra attendre la rotation suivante pour envisager une simplification.

Simplifier des angles de rotation

Sachant qu'une rotation complète représente 360°, le principe de simplification utilise une simple addition d'angle (dans le sens anti-horlogique) pour déterminé l'angle final de rotation.



07RI40a – Angles équivalent à la rotation

Voici les rotation et équivalent d'angle :

•R : tourner à droite = +90°

•S : tout droit = +0° car il n'y a pas de rotation !

•L : tourner à gauche = +270°

•B : demi-tour = +180°

Dans l'exemple, S + B + L = 0 + 180 + 270 = 450° = 90°

Il va de soi que la réponse attendue doit se trouver entre 0 et 360°. En prenant le reste de la division par 360 (division modulaire), cela retourne un angle de 90°. Ainsi, en Python, l'expression 450 % 360 produit la réponse 90 !

Une fois l'angle final obtenu, il est assez facile de retraduire celui-ci en instruction de rotation :

•0° : S → tout droit (il n'y a pas d'angle de rotation)

•90° : R → tourner à droite.

•180° : B → demi-tour

•270° : L tourner à gauche = +270°

•B : demi-tour = +180°

Amélioration possible

Une première simplification peut déboucher sur un nouveau chemin (suite de rotations) encore simplifiable.

En l'état, le script `maze_solver.py` n'effectue qu'une seule tentative de simplification après chaque rotation (comme le croquis Arduino).

La routine de simplification `simplify_path()` pourrait être immédiatement rappelée pour tenter une nouvelle opération de simplification.

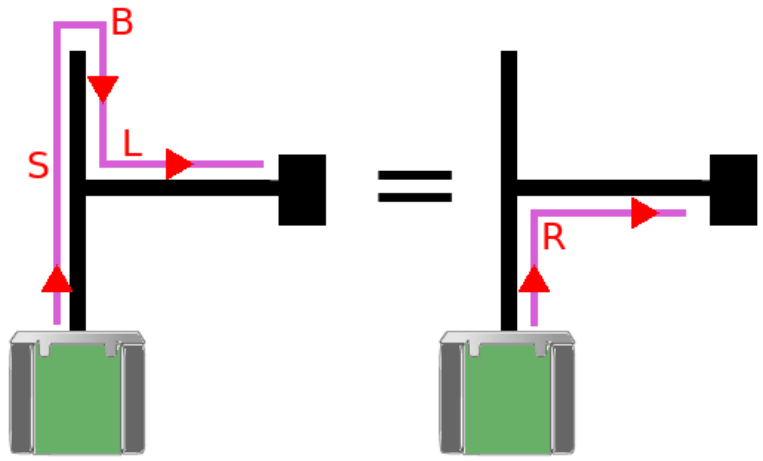
6.3.2.Simplifications avancées

Grâce à Python et sa capacité à modifier facilement des objets complexes, il est possible d'envisager des simplifications nettement plus avancés du chemin à parcourir.

Cette méthode de simplification n'est pas implémentée dans le script d'exemple. Cela reste néanmoins un exercice à la fois pertinent et intéressant.

Cul-de-sac

Dans l'exemple du cul-de-sac repris ci-dessous, le parcours « tout-droit, demi-tour, gauche » peut être raisonnablement simplifié en prenant directement le tournant à droite.



07RI40 – Simplification cul-de-sac

Par conséquent, les occurrences de SBL peuvent être remplacées par R dans un chemin (*path* en anglais) tel que R**SBL**LLL**SBL**LLLRR . Ainsi, le chemin simplifié devient R**R**LLL**R**LLLRR .

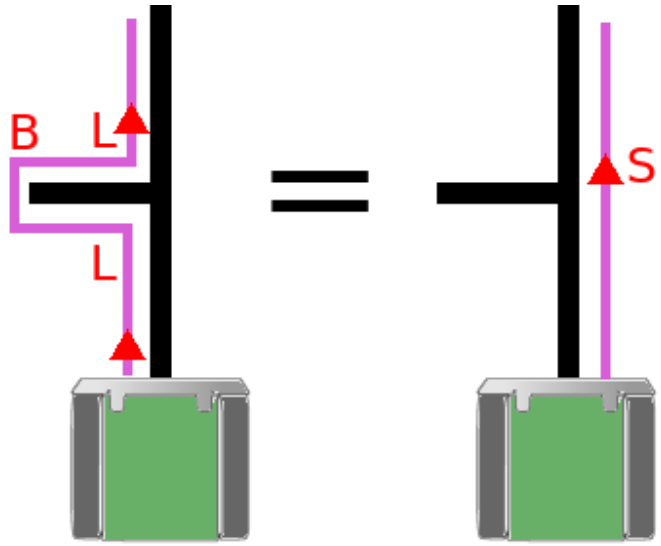
La simplification se note alors :

•SBL → R

•LBS → R (corollaire du cas précédent)

<u>Voie perdue simple</u>

Dans l'exemple ci-dessous, une voie en ligne droite présente une voie annexe à simple, sans issue, sur la gauche nécessitant de rebrousser chemin.



07RI41 – Voie perdue simple

Les occurrences de LBL peuvent être remplacées par S dans le chemin. Ainsi, le chemin (*path* en anglais) tel que R**LBL**BLLRBRR peut être simplifié en R**S**BLLRBRR .

La simplification se note alors :

•LBL → S

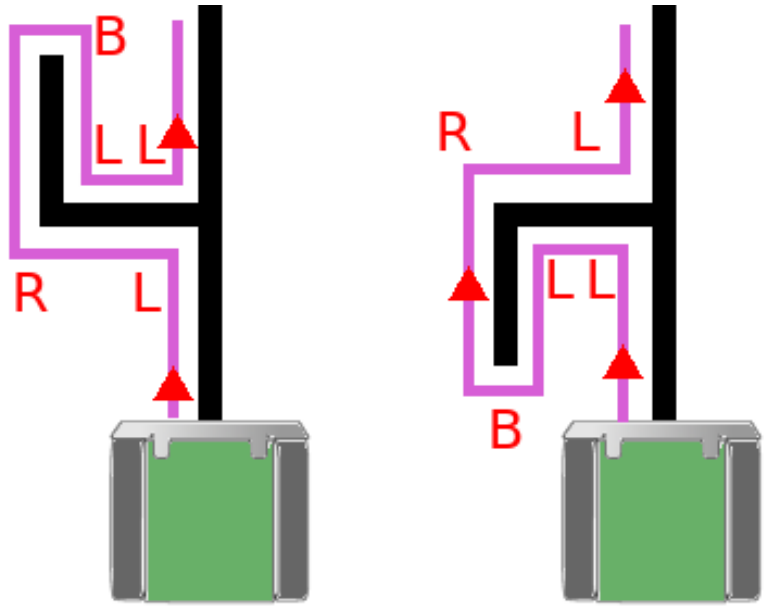
•RBR → S (corollaire du cas précédent)

Le lecteur attentif remarquera que le chemin R**S**BLLRBRR *peut être simplifié une deuxième fois de* R**SBL**LRBRR *vers* R**R**LRBRR .

Si un parcours à main gauche n'offre, à priori, pas l'opportunité d'avoir un parcours RBR, une simplification de parcours pourrit fort bien déboucher sur une telle solution.

Voie perdue - généralisation

Le cas précédent peut être généralisé avec un deuxième niveau.



07RI42 – Voie perdue avec 2ieme niveau

Ainsi, les séquences LRBLL et LLBRL peuvent être remplacées par un S.

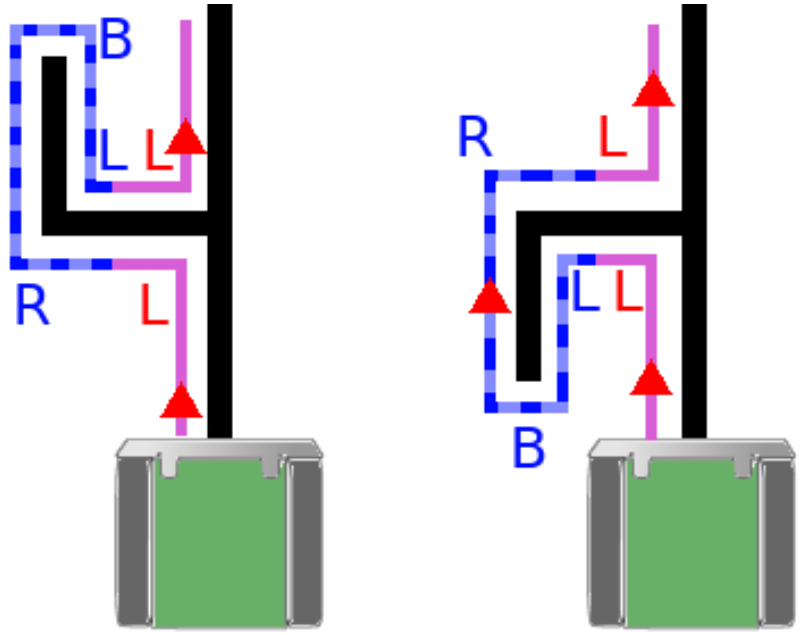
Il y a une certaine beauté dans ces séquences étant donnée qu'en lisant la première à l'envers cela produit la seconde séquence.

En prolongeant la voie perdue sur un troisième niveau, cela produit alors les quatre séquences suivantes pouvant être remplacées par un S :

- LLRBLRL
- LRLBRLL
- LLLBRRL
- LRRBLLL

Simplification en cascade

L'exemple ci-dessous met en lumière une simplification qui peut être conduite sur l'exemple précédent.

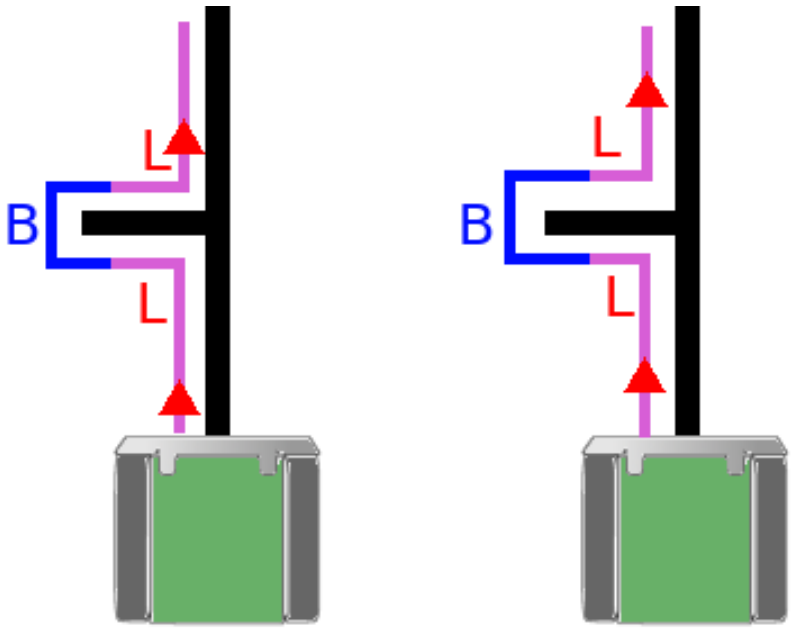


07RI43 – Simplification en cascade de deux parcours

Les appendices RBL et LBR en bout de voie (marquée en pointillés bleus) peuvent être simplifié en simple cul-de-sac avec un B (demi-tour).

- RBL → B
- LBR → B

Ce qui conduit aux parcours deux parcours simplifiés suivants :



07RI44 – Parcours en première simplification

Au final, ces parcours simplifiés en LBL sont des cul-de-sacs pouvant à nouveau être simplifiés en LBL → S

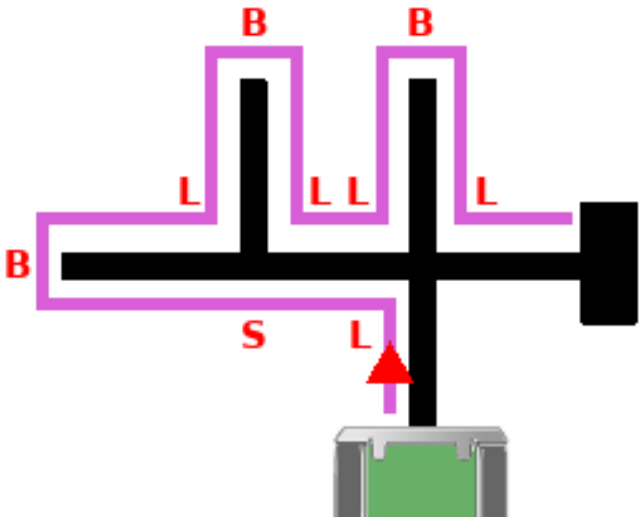
Avec la simplification en cascade, il devient possible de simplifier les parcours en cul vraiment très complexes.

Ainsi, le chemin LLRBLRL peut être simplifié comme suit :

LL**RBL**RL → L**LBR**L → **LBL** → S

6.3.3.Tester la simplifications avancées

En utilisant les règles précédentes sous forme d'un dictionnaire Python, il est possible d'écrire une fonction de simplification qui tient en 10 lignes de code !



07RI45 – Parcours à simplifier

Il est évident que le chemin LSBLBLLBL peut être simplifié par un simple parcours R !

Le script `maze_simplify.py` détaillé ci-dessous est capable de simplifier efficacement un tel parcours en détaillant les opérations intermédiaires.

A partir de l'appel

```
chemin = ['L', 'S', 'B', 'L', 'B', 'L', 'L', 'B', 'L']
print( "Results is", simplify(chemin) )
```

le script produit une série d'information détaillant la simplification du chemin à suivre :

```
Start path: LSBLBLLBL
SBL -> R | path= LRBLLBL
RBL -> B | path= LBLBL
LBL -> S | path= SBL
SBL -> R | path= R
Results is ['R']
```

Voici les détails du script maze_simplify.py :

```
01: SUBS = {"RBL":"B", "LBR":"B", "SBL":"R", "LBS":"R",
02:            "LBL":"S", "RBR":"S" }
03: def simplify( path ):
04:   path_str = ''.join( path ) # List to String
05:   print( 'Start path: %s' % path_str )
06:
07:  try_again = True # Enter the loop
08:   while try_again:
09:     try_again = False
10:     for rpl_from, rpl_by in SUBS.items():
11:       if rpl_from in path_str:
12:         path_str = path_str.replace( rpl_from, rpl_by )
13:         print( rpl_from, "->", rpl_by , "| path=", path_str )
14:
15:         try_again = True
16:         break
17:   return list( path_str )
18:
19: chemin = ['L', 'S', 'B', 'L', 'B', 'L', 'L', 'B', 'L']
```

```
20: print( "Results is", simplify(chemin) )
```

•Lignes 1 et 2 : définition du dictionnaire `SUBS` encodant les simplifications. SUBS est déclaré en capitale, c'est donc une constante qui sera disponible globalement dans le script. L'intérêt du dictionnaire est de pouvoir associé une clé et la valeur correspondante. Ainsi, la séquence « RBL » est mise en correspondance avec « B », la séquence « LBL » correspond à sa simplification « S ». Les séquences et correspondances sont habilement encodées sous forme de chaînes de caractères.

•Lignes 3 à 17 : définition de la fonction `simplify()` qui sera détaillé un peu plus loin. Cette fonction reçoit le chemin à simplifier sous forme d'une liste comme c'est le cas pour le script `maze_solver.py` . Par exemple `simplify( ['L', 'S', 'B', 'L', 'B', 'L', 'L', 'B', 'L'] )` . La fonction `simplify()` retourne également le chemin simplifié sous forme d'une liste.

•Ligne 19 : définition d'un chemin à simplifier. Ce chemin est une liste d'action.

•Ligne 20 : appel de la fonction `simplify( chemin )`. Le résultat, qui est la liste `['R']`, est affiché à l'aide de l'instruction `print()` .

Détail de simplify()

Python, tout comme MicroPython, se montre très efficace dès qu'il est question de réaliser des traitements sur les chaînes de caractères.

Ainsi, le traitement d'un chemin et remplacement des simplifications est beaucoup plus simple à opérer sur des chaînes de caractères que dans des listes.

•Ligne 4 : transformer la liste en une chaîne de caractères. Ainsi, l'instruction `''.join( ['a', '4', 'b', 'c'] )` produit la chaîne de caractères `'a4bc'`.

•Ligne 5 : affichage du chemin alors qu'il est déjà transformé en chaîne de caractères.

•Lignes 7 et 8 : met en place une boucle infinie autour de la variable `try_again` (signifiant littéralement « essayer encore »). Cette variable est préalablement placée à `True` pour démarrer la boucle au moins une fois. Cette boucle poursuit son exécution aussi longtemps qu'il y a une simplification à réaliser. La boucle infinie s'arrête dès lors que la dernière tentative de simplification s'est avérée nulle.

•Ligne 9 : placer la variable `try_again` à `False`. De la sorte, par défaut, il n'y aura pas d'autres exécutions de la boucle à moins que le traitement des lignes 10 à 16 décide d'en faire autrement.

•Ligne 10 : boucle `for` passant en revue toutes les associations existant dans le dictionnaire `SUBS`. Exécute les lignes de 11 à 16 pour chaque itération du dictionnaire. Les variables `rpl_from` et `rpl_by` seront respectivement associées aux valeurs `RBL` et `B` ensuite `LBR` et `B` puis `SBL` et `R`… et ainsi de suite jusqu'à la fin du dictionnaire. La variale `rpl_from` (remplacer_depuis) contient une séquence à remplacer et la variable `repl_by` (remplacer_par) contient la valeur de substitution.

•Ligne 11 : vérifie si la séquence de caractères `rpl_from` (par exemple RBL) existe dans le chemin. Si `rpl_from` n'est pas détecté dans le chemin alors la boucle `for` passe à l'itération suivante. Si `rpl_from` est détecté dans le chemin alors les lignes 12 à 16 seront exécutées.

•Ligne 12 : remplace toutes les occurrences de repl_from (ex : RBL) par son substitut (ex : B) dans le chemin. La méthode `replace()` produit une nouvelle chaîne de caractères, celle-ci est réassignée à la variable du chemin `path_str`.

•Ligne 13 : affichage d'information concernant la substitution et affichage du nouveau chemin.

•Ligne 15 : une substitution vient d'être réalisée, il est donc opportun de poursuivre les tentatives de simplifications pour un tour de plus. La variable `try_again` repasse à `True` pour autoriser une nouvelle exécution.

•Ligne 16 : la boucle `for` est interrompue pour redémarrer une nouvelle itération de la boucle `while`. Il est en effet possible que la remplacement opéré en ligne 12 produise une nouvelle occurrence de `repl_from`, il ne faudrait pas rater cet éventuel remplacement.

•Ligne 17 : si cette ligne est exécutée c'est qu'il n'y a pas eu de substitution opéré par la ligne 10 à 16 (par la boucle `for`). Par conséquent `try_again` reste à `False` et la boucle `while` est interrompue. L'instruction `list( path_str )` permet de transformer la chaîne de caractères en liste d'éléments. Par exemple, `list( 'L4S3Bz' )` produit le résultat `['L', '4', 'S', '3', 'B', 'z']`.

6.4.Principe de fonctionnement

La résolution du labyrinthe se base sur le principe d'héritage, concept tout droit issus de la programmation orienté objet.

Ainsi, la classe `MazeSolver` hérite de classe `ZumoShield` (cf. Bibliothèque Zumo Robot – Diagramme des classes).

La nouvelle classe `MazeSolver` permet ainsi d'ajouter les fonctionnalités complémentaires au Robot Zumo pour qu'il puisse résoudre des labyrinthes.

Le script fonctionne en 3 temps :

1.Calibration du capteur de ligne.

2.Exploration et simplification du labyrinthe.

3.Piloter a nouveau le Zumo vers la sortie.

6.5.Script Maze Solver

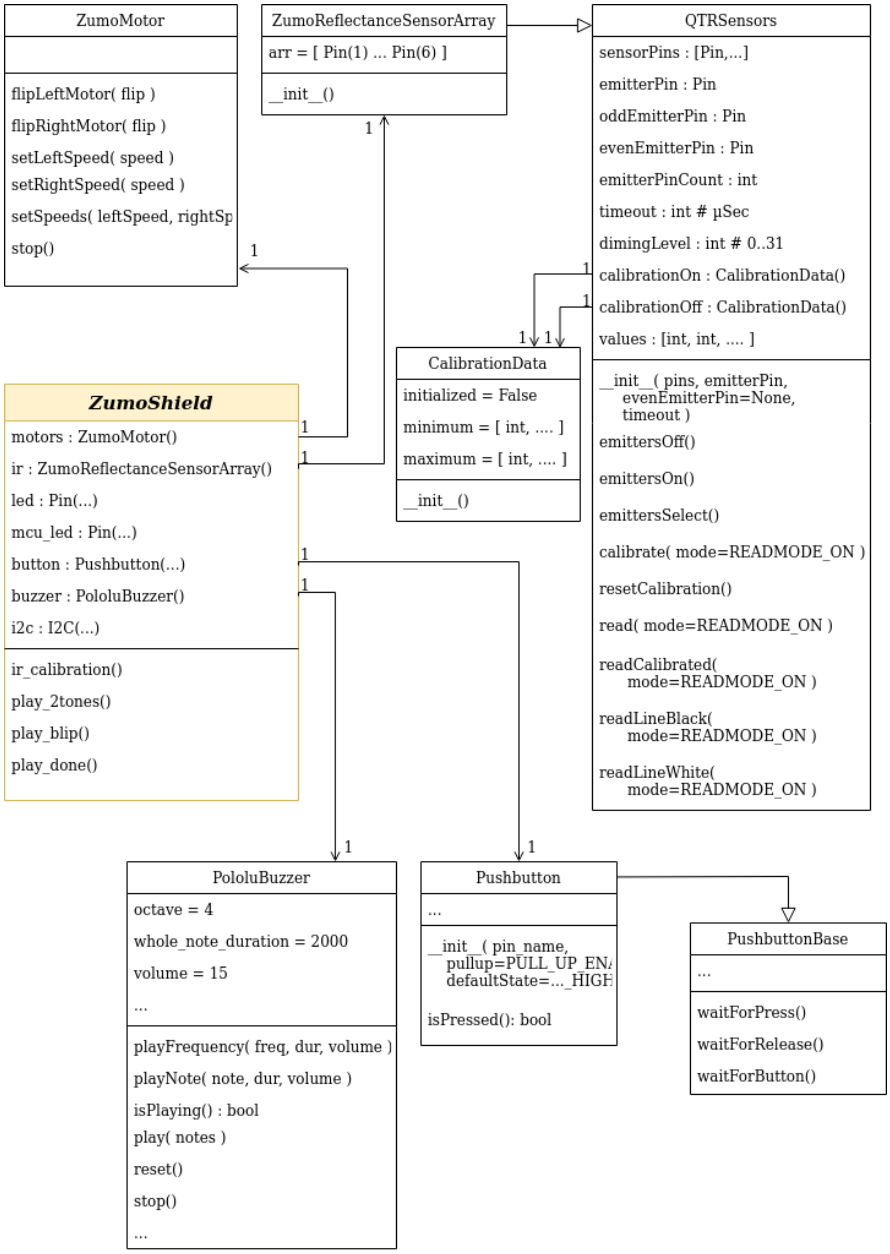
Voici le contenu du script permettant au Zumo Robot de résoudre un labyrinthe s'appelle `maze_solver.py`.

Ce script est disponible dans le dépôt du projet sur le lien suivant :

https://github.com/mchobby/micropython-zumo-robot/blob/main/examples/maze_solver.py

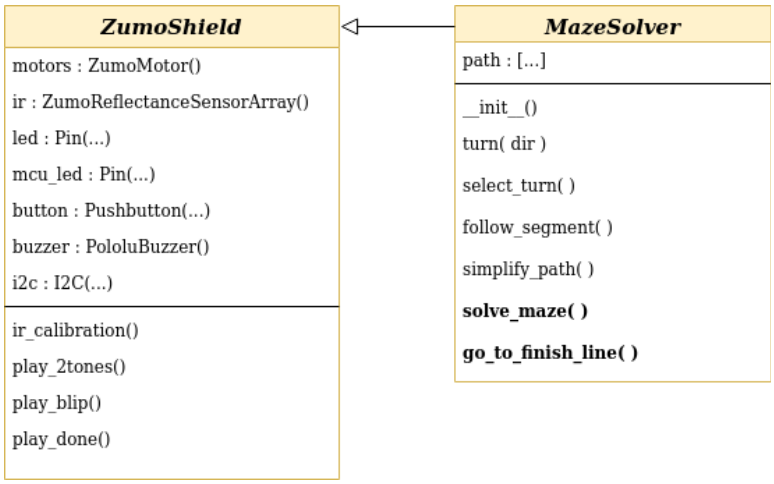
6.5.1.Classe MazeSolver

Pour commencer, voici un petit rappel des fonctionnalités de la classe ZumoShield tel que détaillé dans le chapitre relatif à la bibliothèque Zumo Robot (cf. Bibliothèque Zumo Robot – Diagramme des classes).



07RI50 – Diagramme des classes de ZumoShield

Le diagramme ci-dessous reprend les détails de la classe enfant `MazeSolver` hériter de `ZumoShield`.



07RI51 – Diagramme de ma classe `MazeSolver`.

L'intérêt de cette approche est de permettre à l'instance de `MazeSolver` de :

1.D'offrir l'accès directe aux méthodes de `ZumoShield` comme si `MazeSolver` était un `ZumoShield` (ce qui est effectivement le cas).

2.D'apporter de nouvelles fonctionnalités dans `MazeSolver`, fonctionnalités qui ne sont pas disponibles au niveau `ZumoShield` (et qui ne serait pas pertinent à ce niveau). C'est un procédé de « spécialisation ».

3.De permettre aux nouvelles fonctionnalités de `MazeSolver` d'exploiter les fonctionnalités existantes de son parent (celles de `ZumoShield`).

Constructeur MazeSolver

Lors de la création de l'instance, appel de la méthode `__init__()`, la classe `MazeSolver` crée les différentes instances des objets nécessaires à la résolution du labyrinthe.

La méthode `__init__()` de `MazeSolver` n'oublie pas de faire un appel au constructeur de la classe ancêtre (celui de `ZumoShield`).

Propriété path: list

La propriété `path` est une liste contenant les différentes actions nécessaire au suivit du chemin. Cette liste contient des occurrence de `'L'` (*left*, gauche), `'R'` (*right*, droite), `'S'` (*straight*, tout droit) ou `'B'` (*back*, demi tour) en fonction du chemin parcouru ou simplifié.

La commande suivante permet d'inspecter le chemin parcouru :

```
z = MazeSolver()
print( z.path )
```

ce qui affiche un résultat similaire à celui ci-dessous :

```
['L', 'S', 'B', 'L', 'B', 'L', 'L', 'B', 'L']
```

Propriété path_len: int

Indique le nombre d'éléments présents dans le chemin (voir propriété `path`).

Méthode turn(dir)

Prend le contrôle des moteurs du robot et du capteur de ligne pour appliquer l'ordre de rotation communiqué dans le paramètre dir (soit L, R, S B).

Cette fonction tient compte du positionnement du robot par rapport à la ligne du croisement lorsqu'il tourne à droite ou à gauche.

Cette fonction n'enregistre aucune information.

Méthode select_turn(found_left, found_straight, found_right) : str

Cette fonction est appelée après la détection des embranchements disponibles sur un carrefour. Les variable found_left, found_straight, found_right correspondent respectivement à la détection d'une ligne à gauche, devant et à droite du Zumo Robot.

La fonction `select_turn()` indique le chemin qu'il faudrait suivre L,S,R ou B à partir du croisement et de ses embranchements disponibles.

Cette fonction n'enregistre aucune information.

Méthode follow_segment( )

Prend le contrôle des moteurs et du capteur de ligne du Zumo pour suivre une ligne (un segment) jusqu'au prochain carrefour ou la sortie du labyrinthe. `follow_segment()` retourne la main au code appelant uniquement après la détection du carrefour suivant dans le parcours.

A noter qu'un cul-de-sac est bien un carrefour n'ayant aucune voie à gauche, à droite et à l'avant.

Cette fonction n'enregistre aucune information.

Méthode simplify_path(dir)

Cette méthode est utilisée par `solve_maze()` après chaque carrefour pour tenter de simplifier le parcours des trois derniers éléments stockés dans `path`.

Le fonctionnement de la méthode `simplify_path()` et son appel récurent depuis `solve_maze()` sont des optimisations issues du monde Arduino.

Le script de démonstration `maze_simplify.py` démontre qu'il y a une autre approche d'optimisation possible lorsque que Python est utilisé.

Méthode solve_maze( )

La méthode `solve_maze()` est parmi les plus importantes du script.

Son rôle est de parcourir la labyrinthe à la recherche de la sortie tout en enregistrant le chemin parcouru dans `path` .

Sous sa forme originale (convertie d'un croquis Arduino), la méthode `solve_maze()` procède à une simplification des trois derniers éléments du chemin `path` après chaque carrefour détecté dans le labyrinthe (voir méthode `simplify_path()`).

La méthode `solve_maze()` retourne la main au code appelant lorsque la sortie du labyrinthe est trouvée.

Cette fonction enregistre des informations. La propriété `path` contient une description du chemin à suivre.

Méthode go_to_finish_line( )

La deuxième méthode la plus importante du script.

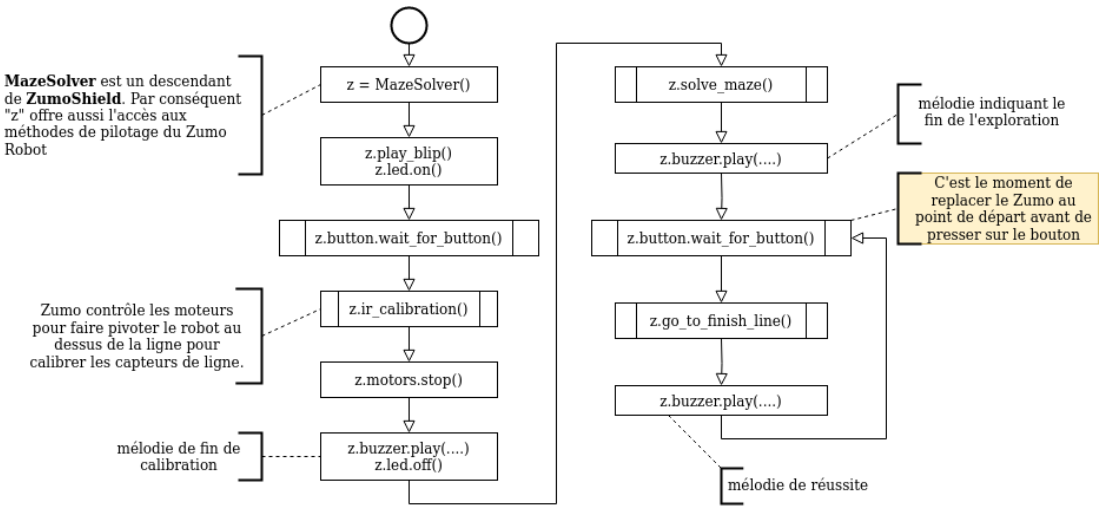
Celle-ci prend le contrôle du Robot Zumo pour parcourir les segments et rotations aux différents carrefours en fonction du chemin enregistré dans la liste `path`.

Cette méthode permet de conduire directement le robot Zumo du point de départ vers la sortie.

Cette fonction n'enregistre aucune information.

6.5.2.Détails du script

Pour commencer, le corps du script se résume à la création d'une instance de la classe `MazeSolver` (objet `z`) puis l'appel procédural d'une série de méthodes.



07RI52 – fonctionnement général du script

Les méthodes digne d'intérêt sont :

- `solve_maze()` : explore le labyrinthe à la recherche de la sortie et enregistre le parcours dans la list path.
- `go_to_finish_line()` : utilise le parcours `path` (simplifié) pour conduire le robot directement à la sortie.

Méthode solve_maze

Voici les détails concernant le parcours du labyrinthe.

```
01: def solve_maze( self ):
02:   while True:
03:     self.follow_segment()
04:
05:     found_left = 0
06:     found_straight = 0
07:     found_right = 0
08:
09:     self.ir.readLineBlack()
10:     if above_line(self.ir.values[0]):
11:       found_left = 1
12:     if above_line(self.ir.values[5]):
13:       found_right = 1
14:
15:     self.motors.setSpeeds(SPEED, SPEED)
16:     time.sleep_ms( overshoot(LINE_THICKNESS)//2 )
```

```
17:     self.ir.readLineBlack()
18:
19:     if above_line( self.ir.values[0] ):
20:       found_left = 1
21:     if above_line( self.ir.values[5] ):
22:       found_right = 1
23:
24:     time.sleep_ms( overshoot(LINE_THICKNESS)//2 )
25:     self.ir.readLineBlack()
26:
27:     if above_line( self.ir.values[0] ):
28:       found_left = 1
29:     if above_line( self.ir.values[5] ):
30:       found_right = 1
31:
32:     if any( [ above_line(self.ir.values[i]) for
                      i in range(1,5)] ):
33:       found_straight = 1
34:
35:     if all( [ above_line(self.ir.values[i]) for
                      i in range(1,5)] ):
36:       self.motors.stop()
37:       break
38:
39:     dir = self.select_turn( found_left,
                      found_straight, found_right )
40:     self.turn( dir )
41:     self.path.append( dir )
42:
43:     self.simplify_path()
44:   return
```

•Ligne 1 : Déclaration de la méthode de l'objet (donc rattachée à une instance). Le premier paramètre `self` est donc une référence référençant l'objet. Cette référence `self` permet d'avoir accès aux autres méthodes et attributs de l'objet.

•Lignes 2, 3, 40, 36 et 37 : Boucle infinie qui s'interrompt avec l'instruction `break` (ligne 37) lorsque la sortie est trouvée. Sinon, le principe général c'est suivre un segment jusqu'au prochain carrefour (`follow_segment` en ligne 3) et tourner (`turn` en ligne 40) en suivant la règle de la main gauche.

•Ligne 3 : Suivre un segment (`follow_segment`) en gardant la ligne au centre du capteur de ligne. La méthode `follow_segment` s'interrompt lorsque le capteur détecte une ligne transversale (plusieurs capteurs activés). A noter que `follow_segment` n'arrête pas les moteurs, la méthode `solve_maze` doit dont réagir promptement pour prendre immédiatement la décision adéquate.

•Lignes 5 à 7 : Initialisation des variables `found_left`, `found_straight`, `found_right` correspondant respectivement à la détection d'une voie à gauche, à l'avant ou à droite. L'initialisation à 0 (équivalent de `False`) indique que la détection n'a pas encore été opérée.

•Ligne 9 : Effectuer un échantillonnage sur le capteur de ligne.

•Lignes 10 à 13 : Si le premier capteur est activé alors il y a une ligne noire sur la gauche. Si le dernier capteur est activé alors il y a une signe noire sur la droite du robot. Pour s'avoir s'il y a une ligne a l'avant, il faudra que le robot se déplace au-delà du carrefour.

•Lignes 15 à 17 : Mettre les deux moteurs à la même vitesse (en effet, `follow_segment` pouvait être en correction de trajectoire et avoir les deux moteurs à des vitesses différentes!). Le robot poursuit un déplacement

correspondant à la demi-épaisseur de la ligne puis refait un échantillonnage sur le capteur de ligne.

•Lignes 19 à 22: revérifie l'état du premier et dernier capteur de ligne une seconde fois et active les variables `found_left` et `found_right`. Cela permet de couvrir le cas probable pour le robot se présente de biais sur la carrefour empêchant ainsi la détection correcte à droite ou à gauche.

•Lignes 24 à 25 : avance une nouvelle fois de la moitié de l'épaisseur de la ligne. Cette fois, le Zumo devrait être de l'autre côté du carrefour (ou presque).

•Lignes 27 à 30 : revérifie une troisième fois l'état du premier et dernier capteur de ligne une seconde fois et active les variables `found_left` et `found_right` (jamais trop prudent).

•Ligne 32 : le robot n'ayant jamais cessé de se déplacer, il a dépassé le carrefour. Si celui-ci est au dessus d'une voie qui se prolonge en ligne droite alors l'un des capteur au centre (sauf extrême gauche et extrême droite) doit encore être activé ! La fonction `any()` est utilisée avec une *list comprehension* pour détecter ce cas de figure. La variable `I` passe en revue les valeurs de 1 à 4 (cfr `range(1,5)`). L'expression `self.ir.values[i]` permet d'obtenir la valeur du capteur infrarouge numéro `I` (dont l'index doit être entre 0 et 6). Enfin `above_line(self.ir.values[i])` permet d'obtenir une valeur `True`/`False` en fonction que le capteur soit (ou non) au dessus d'une ligne. Au final l'expression `[ above_line(self.ir.values[i]) for i in range(1,5)]` retourne une liste `[ False, False, False, False ]` avec un `True` pour une ligne détectée sous un capteur.

> *La fonction any() retourne vraie si au moins un des éléments de la liste est vrai (ou assimilé à vrai). Ainsi any([False, True, False, False]) → True tandis que any([False, False, False, False]) → False.*

> *La list comprehension est une approche élégante pour appliquer une fonction de traitement à une série de valeurs (énumération) puis collecter les résultat dans une « nouvelle liste ». Par exemple [i*10 for i in range(4)] → [0*10, 1*10, 2*10, 3*10] → [0, 10, 20, 30] . Autrement dit : **[** expression **for** membre **in** itérable **]** → nouvelle_liste .*

•Ligne 33 : du test à la ligne précédente, si un des capteurs 1 à 4 du détecteur de ligne détecte une ligne, c'est qu'il y a une voie qui se poursuit en ligne droite au-delà du carrefour. La variable `found_straight` est donc placée a 1 (vrai).

•Ligne 35 : très similaire à la ligne 32, c'est la fonction `all()` qui vérifie si tous les capteurs de 1 à 4 se trouvent au dessus d'une zone noire. C'est le cas de la sortie du labyrinthe.

•Ligne 36 à 37 : Exécutée uniquement si la sortie du labyrinthe est trouvée. Il faut arrêter les moteurs puis interrompre la boucle infinie (ligne 2) avec la commande `break`. Après le `break`, l'exécution passe en ligne 44 qui, elle, quitte la fonction.

•Ligne 39 : la méthode `select_turn()` indique quel est la direction à suivre parmi les voies disponibles (mentionnées par les variables `found_left`, `found_straight`, `found_right`) en respectant le pincipe de la main gauche. Ainsi dir contiendra l'une des valeurs R,L,S,B pour droite, gauche, tout-droit, demi-tour.

•Lignes 40 et 41 : appliquer l'instruction de rotation sur le robot en appelant la méthode `turn( dir )`. La méthode `turn()` prendra soin d'effectuer la rotation dans les règles de l'art en utilisant le capteur de ligne pour déterminer quand celle-ci doit s'interrompre. Enfin, la direction sélectionnée est également ajoutée au chemin déjà parcouru (voir liste `path` en ligne 41).

•Ligne 43 : après tout ajout d'une nouvelle entrée dans le chemin parcouru (`path`), l'appel à la méthode `simplify_path()` permet d'applique l'algorithme de simplification du parcours.

Méthode go_to_finish_line

Voici les détails concernant la fonction permettant de reconduire le robot à destination en suivant le chemin optimisé stocké dans `path`.

```
01: def go_to_finish_line( self ):
02:   start = 0
03:   if self.path[0] == 'B':
04:     self.turn('B')
05:     start=1
06:
07:   for i in range( start, self.path_len ):
08:     self.follow_segment()
09:     self.motors.setSpeeds(SPEED, SPEED)
10:     time.sleep_ms( overshoot(LINE_THICKNESS) )
11:     self.motors.stop()
12:     self.turn( self.path[i] )
13:
14:   self.follow_segment()
15:   self.ir.readLineBlack()
16:   self.motors.stop()
17:   return
```

•Ligne 2 à 5 : Le fonctionnement normal de la fonction est « suivre_segment+tourner » (parcourir le segment puis tourner) et répéter ces opérations jusqu'à la fin de `path`. Il y a cependant une exception, si le robot démarre face à un cul-de-sac, il doit faire demi-tour avant même de commencer à parcourir le premier segment. Dans pareil cas, `path[0]` contiendra un 'B' (*Back* = demi-tour).

•Ligne 2 : Déclare la variable `start` identifiant le premier élément du chemin a utiliser. C'est normalement l'élément 0 qui contient la direction du premier tournant.

•Ligne 3 : Si le premier tournant est un demi-tour 'B' alors on sort du cadre habituel « suivre_segment+tourner ». Il faudra d'abord faire le demi-tour 'B' puis suivre le fonctionnement normal « suivre_segment+tourner » mais à partir de la position 1 du chemin contenu dans `path`.

•Lignes 4 et 5 : effectue la rotation du demi-tour puis modifie la variable `start` pour commencer les opérations « suivre_segment+tourner » à partir de la position 1 du chemin à parcourir.

•Ligne 7 : boucle `for` qui itère tous les éléments du chemin à parcourir (liste `path`) à partir de la position `start`. L'attribut `path_len` permet de connaître le nombre d'éléments dans la liste `path`. La boucle `for` exécute les « suivre_segment+tourner » (ligne 8 à 12) pour chacune des entrées de la liste `path`.

•Ligne 8 : suivre le segment sur lequel le robot se trouve (jusqu'au prochain carrefour).

•Ligne 9 : rétablir une vitesse identique sur les deux moteurs. En effet, `follow_segment()` peut terminer son exécution alors que la fonction corrigeait l'orientation du robot pour suivre la ligne (donc avec des vitesses moteurs différentes).

•Ligne 10 : calcul du dépassement nécessaire et attendre le temps adéquat pour avoir le robot correctement positionné au dessus du carrefour.

•Ligne 11 : arrêt moteur

•Ligne 12 : exécuter la rotation tel qu'enregistré dans le chemin à parcourir. Ensuite la boucle `for` démarre l'itération suivante (ligne 8).

•Ligne 14 : A ce stade, le robot vient d'effectuer la dernière rotation et est face à la sortie du labyrinthe qui se trouve au bout d'un dernier segment. l'instruction `follow_segment()` permet de parcourir cette distance.

•Lignes 15 et 16 : Dernière échantillonnage sur le capteur de ligne et arrêt moteur.

•Ligne 17 : Fin d'exécution de la méthode (et rend la main au code appelant).

6.6.Problèmes et solutions

Les différents tests ont révélé quelques exigences/contraintes concernant le labyrinthe et la configuration du script.

Constante INCHES_TO_ZUNITS

Déjà démontré, le fonctionnement correct du script (dépassement du carrefour et rotation au carrefour) est sensible à la valeur de la constante `INCHES_TO_ZUNITS` .

Ainsi, si le robot n'arrive pas à négocier correctement les virages ou reste a cheval sur deux direction après une rotation, cela signifie qu'il faudra adapter la constante INCHES_TO_ZUNITS .

Le script `test_itoz.py`, également décrit dans cette section, permet de tester facilement différentes valeurs pour la constante.

Constante LINE_THICKNESS

Durant la résolution d'un labyrinthe, le dépassement au carrefour est basé sur l'épaisseur de la ligne (en pouce). En effet, le carrefour est détecté dès que le capteur de ligne se présente au dessus de celle-ci.

Il faut se déplacer de l'autre côté de cette ligne (son épaisseur) puis dépasser celle-ci.

L'épaisseur de la ligne à donc son importance. Une épaisseur de 0,78 pouces correspond à 2cm.

Longueur de segment

Les segments trop courts empêchent la détection correcte des carrefours successifs. Ainsi les longueurs de segment inférieures à 6cm sont à exclure !

Une longueur de segment supérieur à 6cm est vivement recommandé.

Plus le robot se déplace rapidement dans le labyrinthe et plus les segments doivent être longs.

Il s'est avéré qu'avec une vitesse de déplacement de 200, le robot était incapable, sous certaines circonstances, de détecter un deuxième carrefour éloigné de 6cm du

premier. Réduire la vitesse de déplacement à 100 à permis de contourner ce problème.

Vitesse du Zumo

Plus le zumo se déplace vite et plus les écarts de trajectoires sont importants. Le temps de traitement des instructions limite forcement le temps de réaction du script pour corriger une trajectoire erronée ou détecter le passage d'un carrefour (si l'épaisseur de la ligne est trop faible).

La vitesse par défaut de 200 convient à la résolution de parcours simples. Cette vitesse est cependant moins adaptée aux parcours accidentés (avec nombreux croisements) ou parcours avec segments courts (de 6cm de long).

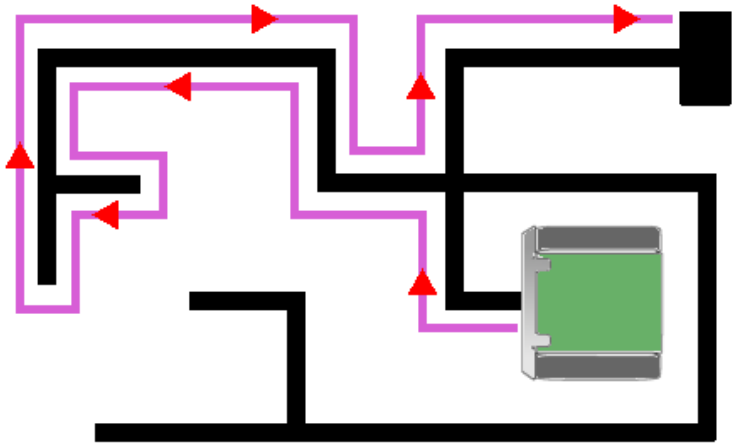
6.7.Encore Plus

6.7.1.Une meilleure simplification

Les labyrinthes les plus complexes démontre facilement les limites du procédé de simplification originel (tel qu'utilisé dans Arduino).

Le parcours ci-dessous et son exploration produit le chemin suivant :

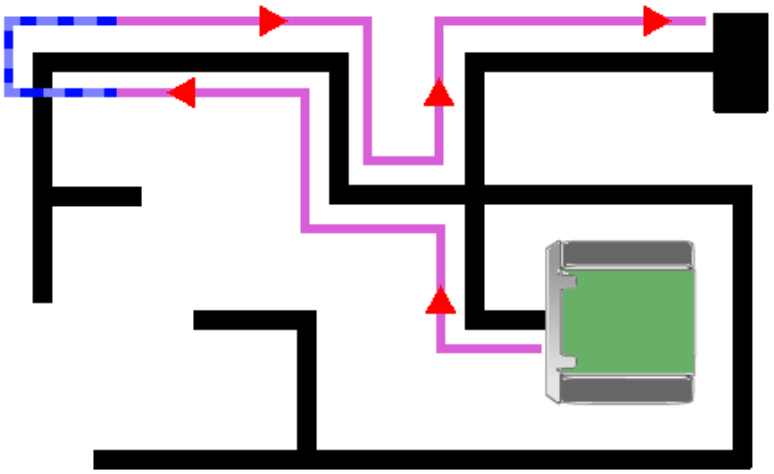
`RLRLLLBLBSRRLLR`



07RI55 – Parcours de test

La simplification Arduino originel produit la simplification suivante :

`RLRL~~LLBLBSR~~RLLR → RLRL**B**RLLR`

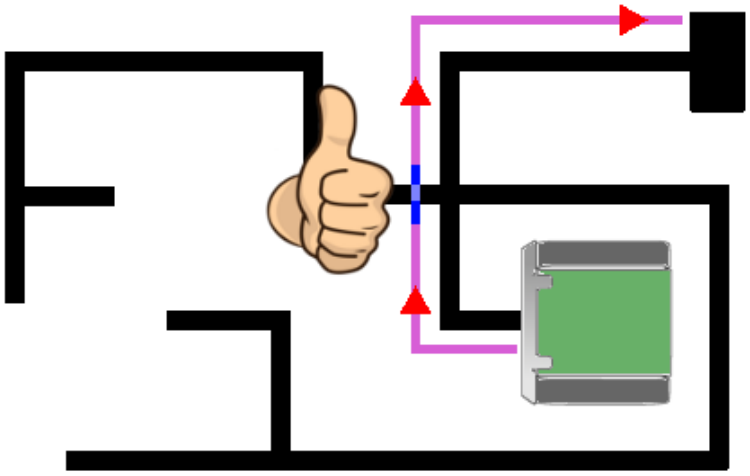


07RI56 – Simplification Arduino

C'est déjà pas mal mais il paraît évident qu'il y a un chemin encore plus direct !

Placer ce chemin dans le script `maze_simplify.py` démontre que ce dernier arrive à un meilleur résultat dans ce même cas de figure.

```
Start path: RLRLLLBLBSRRLLR
('LBS', '->', 'R', '| path=', 'RLRLLLBRRRLLR')
('LBR', '->', 'B', '| path=', 'RLRLLBRRLLR')
('LBR', '->', 'B', '| path=', 'RLRLBRLLR')
('LBR', '->', 'B', '| path=', 'RLRBLLR')
('RBL', '->', 'B', '| path=', 'RLBLR')
('LBL', '->', 'S', '| path=', 'RSR')
('Results is', ['R', 'S', 'R'])
```



07RI57 – résultat de maze_simplify.py

Il y a donc un intérêt a :

1.Retirer l'appel `simplify_path()` dans la méthode `solve_maze()`

2.Implémenter l'algorithme de simplification issue de `maze_simplify.py` dans une nouvelle méthode `simplify_path2()`

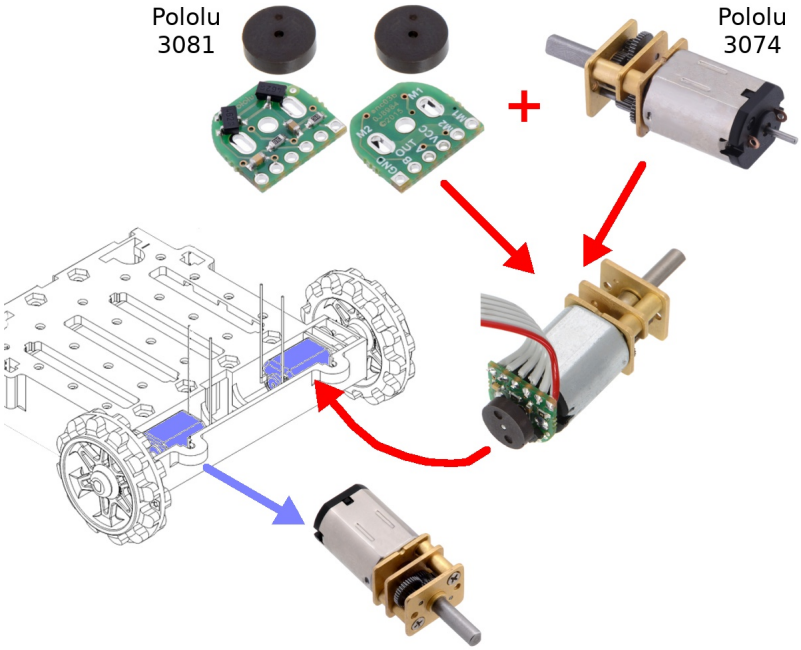
3.Appeler `simplify_path2()` une seule et unique fois à la fin de `solve_maze()`

.

6.7.2.Mesurer les distances

Être capable de mesurer précisément les distances parcourue par le robot. Cela permettrait de contrôler la distance de dépassement du carrefour.

Cela est possible en remplaçant les moteurs du zumo 75:1 HP (High Power, Pololu 3064) par des moteurs 75:1 HP double axe (Pololu 3074) + double encodeur rotatif (Pololu 3088).



07RI59 – Moteurs Pololu avec encodeurs

Les modifications non triviales tant au point de vue matériel que logiciel. Cela serait cependant une personnalisation très intéressante.