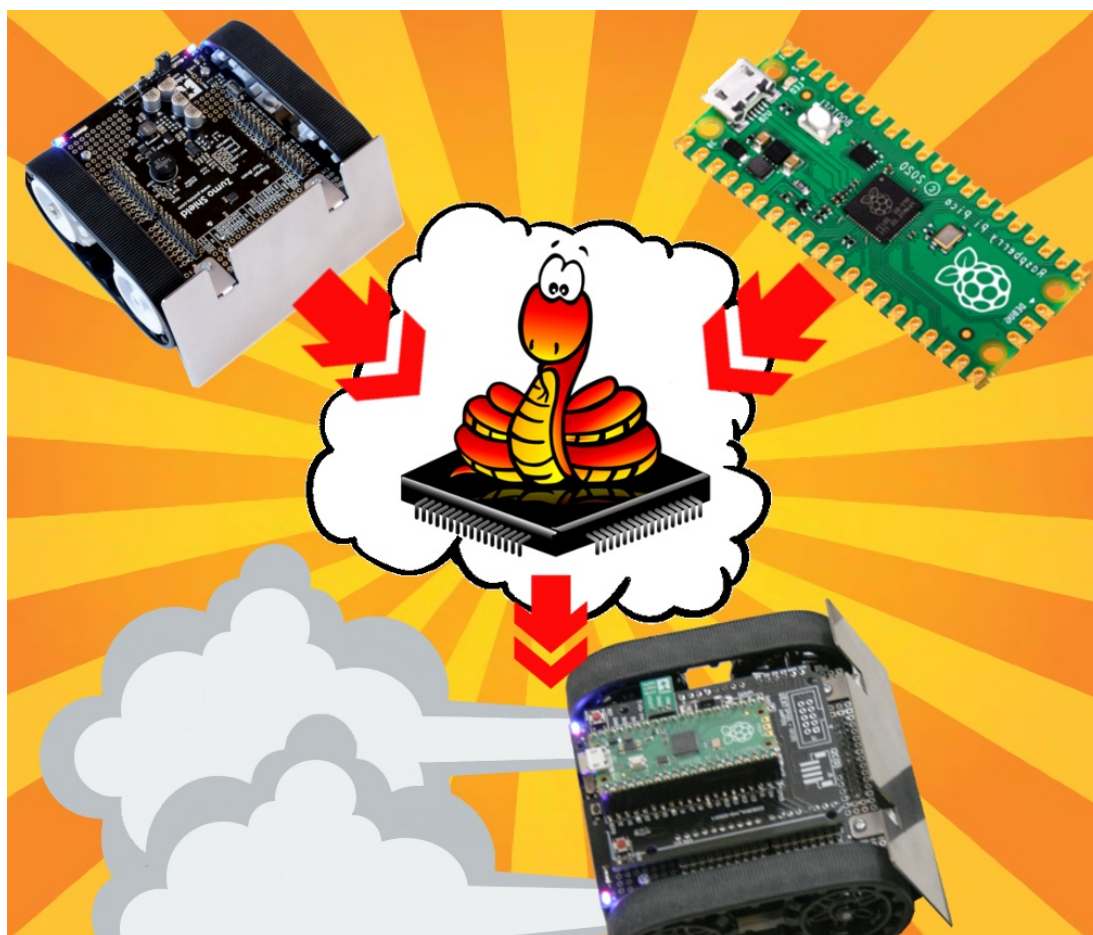


Pico, Zumo Robot et MicroPython

Programmer le Zumo Robot avec Python pour
Microcontrôleur



Tester

Table des matières

1. Introduction.....	3
2. Préparation.....	3
3. Commande moteur.....	4
3.1. Moteur droit.....	4
3.2. Moteur gauche.....	5
3.3. Commande simultanée.....	6
3.4. Rotation.....	6
3.4.1. Rotation sur place.....	6
3.4.2. Rotation en courbure.....	7
3.4.3. Inversion rapide du sens.....	8
4. Détecteur de ligne.....	9
4.1. Lecture brute.....	9
4.2. Calibration et lecture calibrée.....	11
4.2.1. Calibration manuelle.....	11
4.2.2. Calibration avec contrôle moteur.....	12
4.2.3. Lecture de données calibrées.....	13
4.2.4. Recharger les données de calibration.....	14
4.3. Position de la ligne.....	14
5. LEDs.....	15
6. Bouton utilisateur.....	16
7. Tension des piles.....	17
8. Piezo Buzzer.....	17
9. Centrale inertielle.....	18
9.1. Magnétomètre.....	18
9.1.1. Lecture brute.....	18
9.1.2. Lecture boussole.....	19
9.2. Accéléromètre.....	20
9.2.1. Lecture brute.....	21
9.2.2. Détection de choc.....	22
9.3. Gyroscope.....	27
9.3.1. Interpréter les données de rotation.....	30
9.3.2. Digital Zero Rate Level.....	30
9.3.3. Améliorer la qualité des mesures.....	30

1.Introduction

Maintenant que les détails du Robot Zumo, des raccordement sur le Pico et de la bibliothèque MicroPython sont connu, il est temps de passer à la phase de test.

Ces tests permettront de découvrir les différentes fonctionnalité du Zumo mises en relation avec les appel à la bibliothèque.

2.Préparation

Une fois les raccordements achevés (ou carte d'interface) en place, il est temps de se préparer aux différents tests :

1.Le pico exécute le firmware MicroPython et disposant déjà de la bibliothèque Zumo Robot dont l'installation est décrite dans le précédent chapitre (cf. Bibliothèque Zumo Robot – Bibliothèques Zumo Robot).

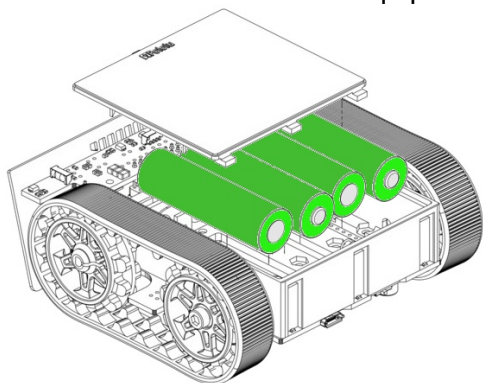
2.Le Robot Zumo configuré de façon appropriée (cf. Brancher – Adaptateur vs méthode Maker, voir point « Rappel sur le Robot Zumo » - section « Configurer le Zumo »).

Cavalier buzze en position 328P.

Cavalier du capteur de ligne en position 2 (ou pas de cavalier). Capteur accessible sous le capteur de ligne.

Cavalier en place sur « A1 - battery_level »

3.Le Robot Zumo doit être équipé avec des piles.

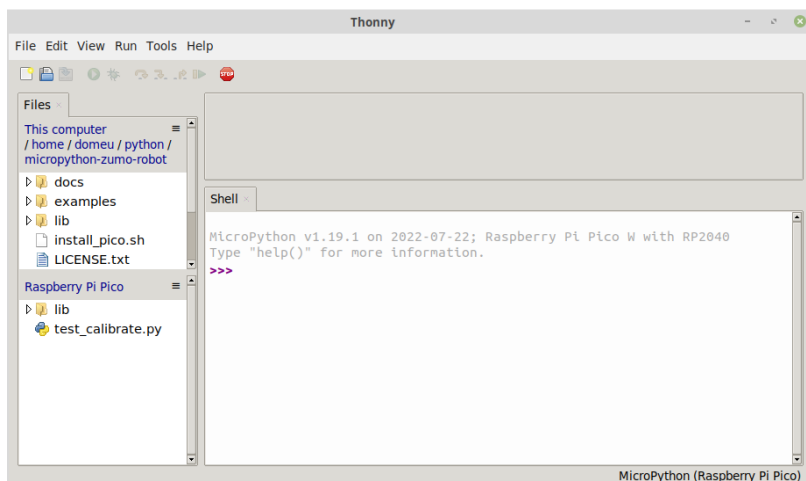


06RI01 – Position des piles dans le Robot Zumo

4.Le Robot Zumo branché sur l'ordinateur à l'aide d'un câble USB.

5.Le Robot Zumo mis sous-tension à l'aide de l'interrupteur marche/arrêt accessible à l'arrière du Zumo (les LEDs du Zumo sont allumées).

6.Démarrer votre outil REPL de prédilection ou a défaut Thonny IDE et établir une connexion REPL avec le microcontrôleur.



06RI02 – Thonny IDE avec connexion REPL (via USB) vers le Pico

3.Commande moteur

Avoir un Zumo qui se déplace avec un câble qui traîne derrière n'est pas confortable et sera source de nombreuses manipulations.

Ainsi, pour tester plus facilement la motorisation, il suffit alors de sur-élever le Zumo sur une boîte de sorte que les chenilles ne touche plus le sol.

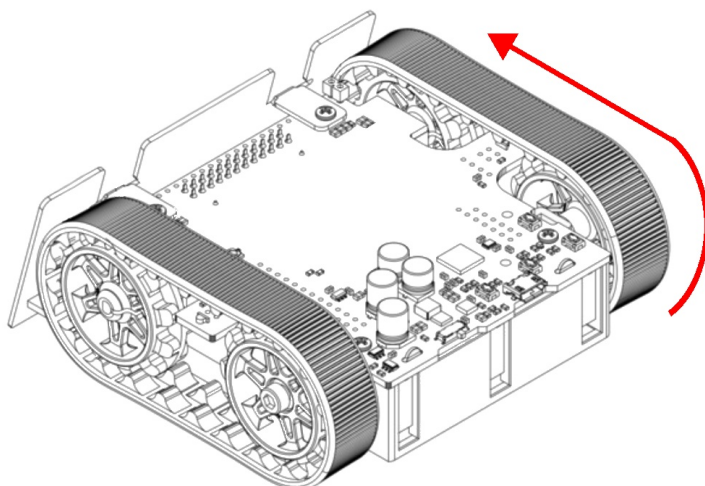


06RI03 – Placer le Zumo sur un support sur-élevé.

Les différents exemples de cette section sont consultable dans le dépôt du projet sous le nom `examples/test_zumoshield.py`.

3.1.Moteur droit

Commençons par tester le moteur droit et vérifier qu'il fait bien avancer le Robot Zumo pour une vitesse positive.



04RI04 – Marche avant du moteur droit.

La méthode utilisée est ici `motors.setRightSpeed(vitesse)` avec une valeur entre -400 et +400.

```
>>> from zumoshield import *
>>> z = ZumoShield()
>>> z.motors.setRightSpeed( 100 )
>>> z.motors.stop()
```

La méthode `motors.stop()` permet d'arrêter les deux moteurs en une seule instruction. Il est également possible d'utiliser `motors.setRightSpeed(0)`.

Sur une plateforme Zumo Robot officielle et la carte adaptateur Pico-Zumo cela doit fonctionner comme attendu.

Dans le cadre de la réalisation d'un robot Maker il se peut que la chenille ne fonctionne pas comme attendu.

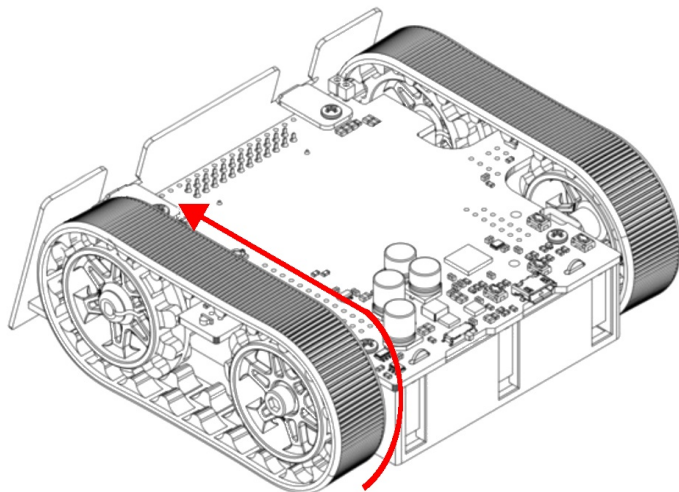
- S'il s'avère que la chenille du moteur droit tourne dans l'autre sens alors il suffit d'inverser les deux connexions électriques du moteur.
- S'il s'avère que c'est le moteur gauche qui fonctionne au lieu du moteur droit alors il suffit de connecter les fils d'alimentation du moteur droit sur le moteur gauche et vice-versa.

3.2.Moteur gauche

Ensuite, pour tester le moteur gauche, il suffit d'utiliser la méthode `motors.setLeftSpeed(vitesse)` avec une valeur entre -400 et +400 .

```
>>> from zumoshield import *
>>> z = ZumoShield()
>>> z.motors.setLeftSpeed( 100 )
>>> z.motors.stop()
```

👉 Si l'instance du `ZumoShield` existe déjà (la variable `z`) alors il n'est pas nécessaire de re-saisir les deux premières lignes.



06RI05 – Marche avant du moteur gauche

S'il s'avère que le moteur ne tourne pas dans le bon sens (sur un robot réalisé par l'utilisateur), cela signifie qu'il faut inverser les deux fils de connexion sur le moteur.

3.3. Commande simultanée

Il est possible de commander les deux moteurs en une seule opération grâce à `motors.setSpeeds(vitesse_gauche, vitesse_droite)`.

L'exemple ci-dessous présente différentes commandes moteur avec, en commentaire, le résultat attendu.

```
>>> from zumoshield import *
>>> z = ZumoShield()
>>> # Marche avant lente
>>> z.motors.setSpeeds( 100, 100 )
>>> # Marche avant rapide
>>> z.motors.setSpeeds( 400, 400 )
>>> # Marche arrière lente
>>> z.motors.setSpeeds( -100, -100 )
>>> # arrêt
>>> z.motors.setSpeeds( 0, 0 )
```

3.4. Rotation

Le robot Zumo peut tourner soit :

- **Sur place** : pratique durant se déplacer dans un labyrinthe ou éviter un objet/concurrent durant une compétition de robot.
- **En courbure** : adopter une courbure en tournant est indiqué pour le suivit d'une ligne ou l'évitement d'un objet distant.

3.4.1. Rotation sur place

Pour que le Zumo robot tourne sur place, il suffit que les moteurs tournent à la même vitesse dans des sens opposés.

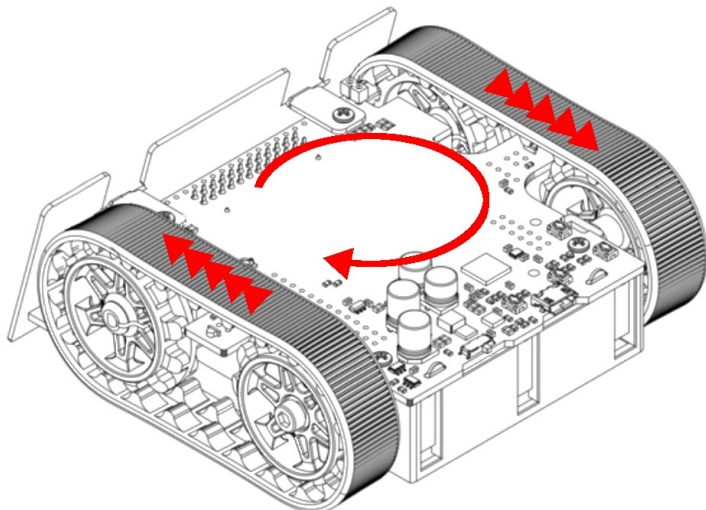
L'angle de rotation dépend de la vitesse des moteurs et du temps de rotation.

➡ *Il n'est pas aisé d'établir une relation fiable entre vitesse/temps et angle de rotation car la vitesse des moteurs diminue avec la tension des piles. Le champ magnétique terrestre peut être utilisé pour palier à cet inconvénient.*

Chapitre 6 : Tester

Pour effectuer une rotation à droite, il moteur gauche est en marche avant tandis que le moteur droit est en marche arrière.

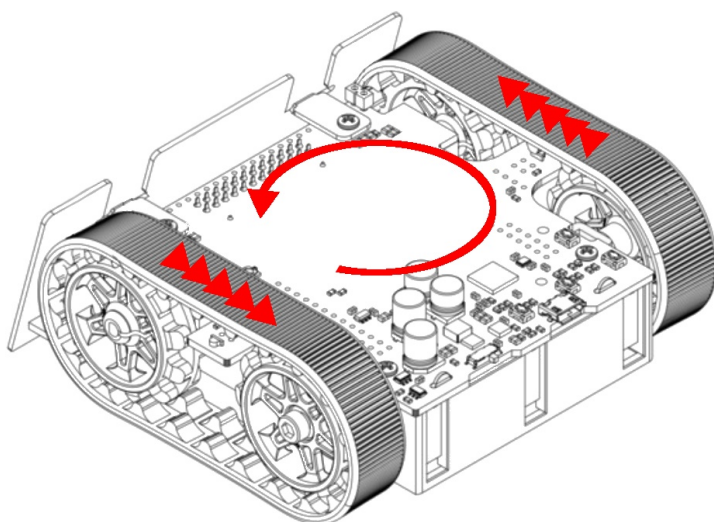
```
>>> from zumoshield import *
>>> z = ZumoShield()
>>> # Rotation à droite
>>> z.motors.setSpeeds( 200, -200 )
>>> # Arrêt
>>> z.motors.stop()
```



06RI06 – Rotation sur place à droite

En inversant la logique, le robot tourne à gauche

```
>>> from zumoshield import *
>>> z = ZumoShield()
>>> # Rotation à droite
>>> z.motors.setSpeeds( -200, 200 )
>>> # Arrêt
>>> z.motors.stop()
```



06RI07 – Rotation sur place à gauche

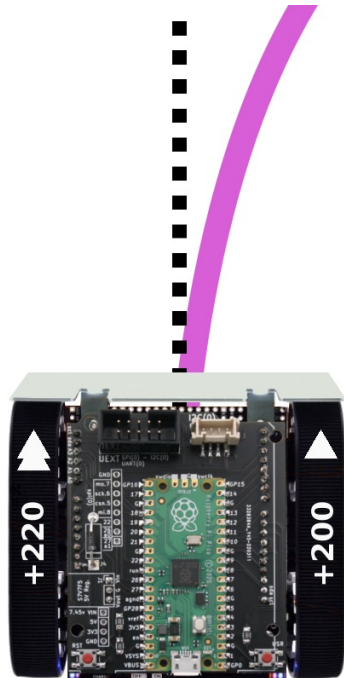
3.4.2. Rotation en courbure

Pour que le Zumo robot tourne en dessinant une courbe, les deux moteurs doivent tourner dans le même sens avec une légère différence de vitesse.

Chapitre 6 : Tester

Si le moteur gauche tourne plus vite que le moteur droit alors le robot dessinera une courbe sur la droite.

```
>>> from zumoshield import *
>>> z = ZumoShield()
>>> # Rotation avec courbure à droite
>>> z.motors.setSpeeds( 220, 200 )
```



06RI08 – Courbure sur la droite

Si le moteur droit tourne plus vite que le moteur gauche alors le robot dessinera une courbe vers la gauche.

➡ *Plus la différence de vitesse entre les deux moteurs est grande et plus le rayon de courbure sera petit (le tournant sera plus serré).*

3.4.3. Inversion rapide du sens

Les méthodes `flipLeftMotor(flip)` et `flipRightMotor(flip)` permettent d'inverser le contrôle moteur en une instruction. Cette inversion sera active dès le prochain appel à l'une des méthodes `setSpeed()`.

Cela peut s'avérer intéressant pour passer d'une marche avant à une rotation sur place et inversement.

```
>>> from zumoshield import *
>>> z = ZumoShield()
>>> # Avance
>>> z.motors.setSpeeds( 200, 200 )
>>> # rotation vers la gauche
>>> z.motors.flipRightMotor( True )
>>> z.motors.setSpeeds( 200, 200 )
>>> # Rétablissement de la marche avant
>>> z.motors.flipRightMotor( False )
>>> z.motors.setSpeeds( 200, 200 )
```

➡ *L'utilité des méthodes `flipXxxMotors()` n'est pas évident de prime abord. Cependant sont existence dans la bibliothèque Arduino pour le Robot Zumo*

remonte à plusieurs années. Leurs existences doit donc présenter une utilité même si elle n'est pas évidente.

4. Détecteur de ligne

Le deuxième élément le plus important après la commande moteur c'est le détecteur de ligne constitué de 6 capteurs infrarouges.

Ce détecteur est prévu pour détecter et positionner sous le capteur des lignes noires de 15mm. Le capteur peut aussi être utilisé pour limiter une zone (comme la bordure d'un ring de combat robotique).

👉 *Bien qu'initialement conçu pour détecter une ligne noire sur fond blanc, ce détecteur peut aussi détecter des lignes blanches sur fond noir (la bibliothèque a même prévu une fonction à cet effet).*

4.1. Lecture brute

Avant d'utiliser les fonctionnalités avancées du détecteur de ligne, les 6 capteurs seront interrogés de sorte à obtenir des données brutes.

Le script ci-dessous active les LEDs infrarouges avec l'appel à `ir.emittersOn()` (des LEDs rouges sont par ailleurs activées sous le détecteur ligne pour signaler cette activation). Par la suite, le contrôle des LEDs infrarouges sera reprise par la méthode `ir.read()`.

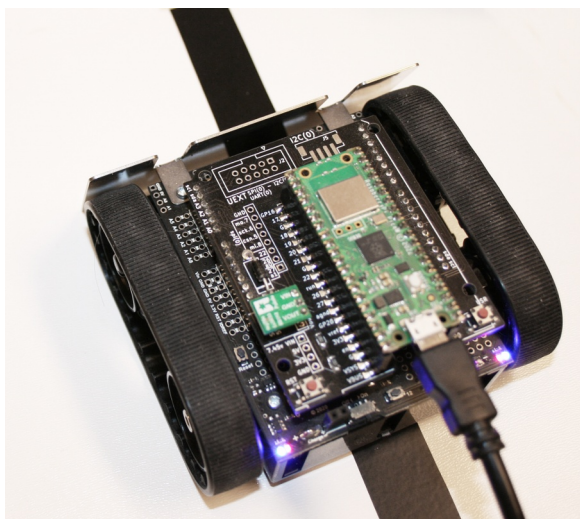
Un appel à la méthode `ir.read()` permet d'échantillonner les capteurs infrarouges du détecteur de ligne.

Les données des capteurs sont accessibles via la propriété `values` exposant les 6 éléments échantillonnées sur le détecteur de ligne (une entrée par capteur). Données qui sont affichées dans la session REPL

```
>>> from zumoshield import *
>>> import time
>>> z = ZumoShield()
>>> z.ir.emittersOn()
>>> while True:
>>>     z.ir.read()
>>>     data = z.ir.values
>>>     print( '%4i %4i %4i %4i %4i %4i ' %
>>>             (data[0],data[1],data[2],data[3],data[4],data[5]) )
>>>     time.sleep_ms(500)
```

Pour pouvoir tester cette fonctionnalité, il convient de placer le robot sur une surface blanche équipé d'une ligne noire de 15mm de large.

Dans le cas présent, la ligne noire est un peu décentrée vers la droite du robot.



O6RI10 – Zumo placé sur une ligne noire.

Une fois le script démarré, il affiche les données des capteurs en respectant l'ordre des capteurs (de gauche à droite).

Ces informations sont capturées toutes les 1/2 secondes. Il est par ailleurs possible de voir les LEDs rouges clignoter brièvement sous le détecteur de ligne.

Une fois le script en cours d'exécution, il produit 5 colonnes de données, une colonne par capteur.

298	298	298	1132	1132	441
294	294	294	1127	1127	432
296	296	296	1000	1147	296
297	297	297	1139	1139	297
313	313	313	1013	1159	313
313	313	313	1010	1160	313
310	310	310	1009	1154	310
306	306	306	1009	1156	306
296	296	296	1127	1127	436
302	302	302	1008	1152	302
305	305	305	1002	1145	305
305	305	305	1010	1153	305
304	304	304	1005	1150	304
301	301	301	997	1139	445
301	301	301	1141	1141	301
303	303	303	1004	1149	303

➤ *Les valeurs plus élevées sur les 4^{ième} et 5^{ième} colonnes correspondent bien à la position de la ligne sous le capteur. Pour rappel, moins de lumière réfléchie vers le récepteur infrarouge = un temps de décharge plus long.*

Déplacer le robot au dessus de la ligne permet de constater les variations de valeur dans les différentes colonnes.

Cet exemple peut également être utilisé pour évaluer différents matières réfléchissantes et non réfléchissantes avec le capteur de ligne.

Le script est disponible dans les sources du dépôt sous le nom `examples/test_qtrrc_rawvalues.py`.

4.2. Calibration et lecture calibrée

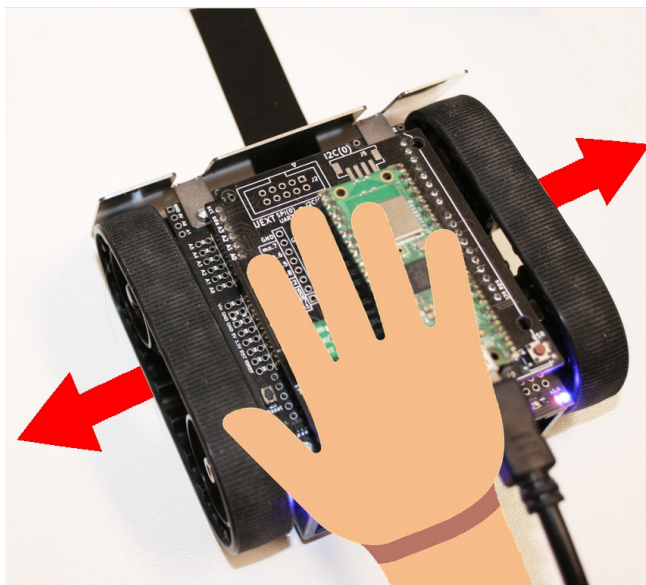
Les différents capteurs infrarouges ne sont pas strictement identiques. Par conséquent les valeurs minimales et maximales que chaque capteur est différent du capteur à côté. Ces différences peuvent être sensibles ou relativement importantes.

Créer des scripts tenant compte des spécificités pour chaque capteur peut représenter un vrai challenge.

Une solution élégante consiste à réaliser une phase de calibration relevant le minimum et le maximum pour chaque capteur infrarouge puis de réaliser une lecture en normalisant la valeur brute obtenue par rapport au maximum et au minimum connu pour ce même capteur. C'est un peu comme réaliser une lecture de données avec un résultat retourné entre « 0 et 100 % » capteur par capteur.

4.2.1. Calibration manuelle

Par calibration manuelle, il faut comprendre que le Zumo ne sera pas motorisé et qu'il faudra déplacer celui-ci au dessus d'une ligne noire.



06R113 – Déplacer le Zumo au dessus d'une ligne

Le script d'exemple ci-dessous :

1. Débuter par une phase de calibration (80 étapes) avec `z.ir_calibration(motors=False)` où il faudra déplacer progressivement le Zumo au dessus de la ligne puisque les moteurs ne sont pas activés.

2. Les données de calibration sont ensuite assignées aux variables `json_on` et `json_off`.

3. Affiche les minimas et maximas par capteur (les données de calibration).

```
>>> from zumoshield import *
>>> z = ZumoShield()
>>>
>>> # Effectue la calibration sans contrôler les moteurs
>>> z.ir_calibration( motors=False )
>>>
>>> # Affiche les données de calibration
>>> json_on = z.ir.calibrationOn.as_json()
>>> json_off = z.ir.calibrationOff.as_json()
```

```
>>>
>>> print( "calibrationOn : %s" % json_on )
calibrationOn : {"maximum": [2000, 1888, 1853, 1707, 1880, 2000],
"minimum": [282, 282, 282, 282, 282, 285], "initialized": true}
>>>
>>> print( "calibrationOff : %s" % json_off )
calibrationOff : {"maximum": null, "minimum": null, "initialized":
false}
```

La variable `json_on` contient les maximums et minimums pour chaque capteur lorsque les LEDs infrarouges sont activées. Cette structure sera initialisée puisque la calibration est réalisée en exploitant les LEDs infrarouges du capteur de ligne.

La variable `json_off` contient les maximums et minimums pour chaque capteur lorsque les LEDs infrarouge ne sont pas activées. Cette structure n'est pas initialisée puisque

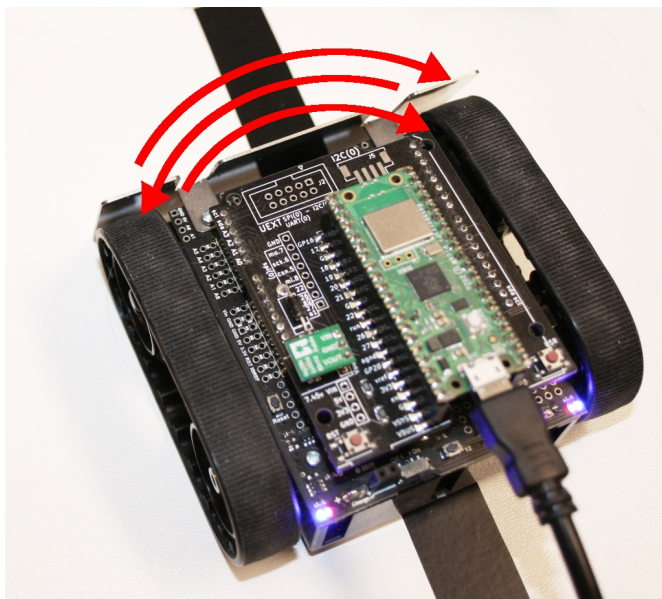
Ces données peuvent être récupérées au format JSon dans une chaîne de caractères. Cela permet, le cas échéant, de stocker les paramètres de calibration directement le script ou dans un fichier texte. Lorsque les données de calibration sont connues, il est possible de les recharger, de sorte qu'il n'est plus nécessaire de réaliser la phase de calibration.

4.2.2. Calibration avec contrôle moteur

Certes, un robot qui se met en mouvement pour faire une calibration cela est surprenant. C'est pourtant l'approche la plus adéquate pour aboutir à une calibration optimale.

Appeler la méthode `ir.calibration(motors=True)` ou sans paramètre fait osciller le zumo au dessus de la ligne servant d'étalonnage. Les moteurs sont commandés avec des vitesses de +100 et -100, donc pas trop rapide pour une calibration correcte.

En fin de calibration motorisée, la méthode `ir.calibration()` stoppe les moteurs.



06RI12 – Calibration avec commande moteur

4.2.3. Lecture de données calibrées

Une fois la calibration terminée, les données des capteurs peuvent être obtenues à l'aide de la méthode `ir.readCalibrated()` qui est alors capable de retourner une valeur normalisée 0 et 1000 pour chacun des capteurs.

Le minima et maxima de chaque capteur étant connu grâce à la calibration, une simple règle de trois permettra de ramener la valeur lue sur un capteur entre 0 et 1000.

Dans l'exemple ci-dessous, le Zumo effectue une calibration motorisée **ensuite le Robot sera déplacé à la main au dessus de la ligne** tandis que différentes itérations de `readCalibrated()` sont effectuées. En affichant les valeurs des différents capteurs, il est possible de repérer la position de la ligne sous le capteur en inspectant la (ou les) valeur(s) proche(nt) de 1000.

Les données se lisent de gauche à droite (exactement comme sont organisés les capteurs infrarouges sur le détecteur de ligne).

```
>>> from zumoshield import *
>>> z = ZumoShield()
>>>
>>> # Effectue la calibration AVEC contrôler les moteurs
>>> z.ir_calibration( motors=True )
>>>
>>> z.ir.readCalibrated()
>>> data = z.ir.values
>>> print( '%4i %4i %4i %4i %4i %4i ' %
          (data[0],data[1],data[2],data[3],data[4],data[5]) )
          14    14    16    18    954    104
>>>
>>> z.ir.readCalibrated()
>>> print( '%4i %4i %4i %4i %4i %4i ' %
          (data[0],data[1],data[2],data[3],data[4],data[5]) )
          9     9    656    617    11     9
>>>
>>> z.ir.readCalibrated()
>>> print( '%4i %4i %4i %4i %4i %4i ' %
          (data[0],data[1],data[2],data[3],data[4],data[5]) )
          254   671    3     4     3     2
>>>
```

Après la calibration réalisée à l'aide de `z.ir_calibration(motors=True)`, la méthode `readCalibrated()` permet d'obtenir une lecture du détecteur de ligne avec une valeur bornée entre 0 et 1000 pour chaque capteur infrarouge.

L'expression `data = z.ir.values` permet d'accéder rapidement à la liste des valeurs (propriété `values`) avec la variable `data`. Cette expression ne duplique pas les valeurs de `values` mais propose un raccourci pour y accéder (ce que l'on appelle une référence dans le monde de la programmation).

Enfin l'instruction `print()` permet d'afficher les 6 valeurs (index de 0 à 5) du capteur de ligne, données affichées immédiatement après l'instruction `print()`.

Les différents éléments de ce script sont disponibles dans le dépôt du projet sous le nom `examples/test_readline1.py`.

4.2.4.Recharger les données de calibration

En tout début de section, la calibration manuelle permettait d'afficher les informations de calibration dans la session REPL. Cette information est donc accessible sous forme de deux chaînes de caractères.

👉 *Les données de calibration sont propres à un détecteur de ligne donné ainsi qu'à son environnement ! Si le revêtement réfléchissant (ou la ligne) sont modifiés (matière, couleur) alors les données de calibrations sont altérés. Une nouvelle calibration sera nécessaire.*

Soit la définition des variables `calib_on` et `calib_off` contenant ces chaînes de caractères avec les informations de calibration.

```
calib_on = '{"maximum": [2000, 1888, 1853, 1707, 1880, 2000],  
"minimum": [282, 282, 282, 282, 282, 285], "initialized": true}'  
calib_off = '{"maximum": null, "minimum": null, "initialized":  
false}'
```

Le script ci-dessous instancie un `ZumoShield` et réinitialise les données de calibrations à partir des deux chaînes de caractères `calib_on` et `calib_off`.

```
>>> from zumoshield import *  
>>> # Donnees de calibration  
>>> calib_on = '{"maximum": [2000, 1888, 1853, 1707, 1880, 2000],  
"minimum": [282, 282, 282, 282, 282, 285], "initialized": true}'  
>>> calib_off = '{"maximum": null, "minimum": null, "initialized":  
false}'  
>>>  
>>> z = ZumoShield()  
>>> # recharger la calibration  
>>> z.ir.calibrationOn.load_json( calib_on )  
>>> z.ir.calibrationOff.load_json( calib_off )  
>>>  
>>> # Lecture et affichage calibré  
>>> z.ir.readCalibrated()  
>>> print( z.ir.values )  
[0, 0, 0, 776, 170, 75]
```

Les différentes informations du script sont disponibles dans le dépôt du projet sous le nom `examples/test_calibrate.py`.

4.3.Position de la ligne

Lorsque le détecteur de ligne est calibré, il est possible d'utiliser la méthode spécialisée `readLineBlack()` qui permet d'identifier la position de la ligne sous le détecteur.

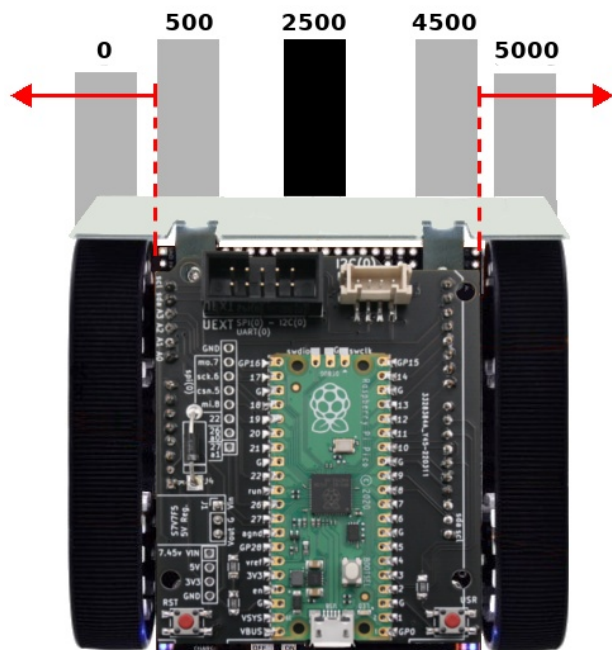
La méthode `readLineBlack()` retourne une valeur numérique permettant de déterminer la position de la ligne.

La valeur est bornée entre 500 et 4500 lorsque la ligne est effectivement sous le détecteur.

Lorsque la valeur descend et atteint 0, cela signifie que la ligne sort sur la gauche du robot.

Lorsque la valeur augmente et atteint 5000, cela signifie que la ligne sort sur la droite du robot.

Dans ces deux cas, il faut impérativement prendre des dispositions pour ramener la ligne sous le capteur.



04RI14 – position de la ligne sous le capteur

```
>>> from zumoshield import *
>>> import time
>>>
>>> z = ZumoShield()
>>>
>>> # Effectue la calibration
>>> z.ir_calibration( motors=True )
>>> while True :
>>>     pos = z.ir.readLineBlack()
>>>     print( pos )
>>>     time.sleep( 1 )
```

Le script de cet exemple est disponible dans le dépôt du projet sous le nom `examples/test_readline2.py`.

5.LEDs

LED du Zumo

Le Zumo robot est équipé d'une LED utilisateur qui peut-être contrôlé par l'intermédiaire de la propriété `led` du `ZumoShield`.

```
>>> from zumoshield import *
>>> import time
>>>
>>> z = ZumoShield()
>>> z.led.on() # allume la LED
>>> z.led.off() # éteindre la LED
```

A noter qu'il est également possible d'utiliser la notation `z.led.value(boolean)` pour contrôler l'état de la LED à l'aide d'une expression booléenne.

LED du Pico

Le Raspberry-Pi Pico (tout comme le Pico Wireless) dispose aussi d'une LED utilisateur.

Chapitre 6 : Tester

Cette LED du microcontrôleur peut-être contrôlée à l'aide de la propriété `mcu_led` du `ZumoShield`.

```
>>> from zumoshield import *
>>> import time
>>>
>>> z = ZumoShield()
>>> z.mcu_led.on() # allume la LED
>>> z.mcu_led.off() # éteindre la LED
```

Comme pour l'autre LED, il est également possible d'utiliser la méthode `value(boolean)` pour contrôler l'état de cette LED.

6. Bouton utilisateur

Le bouton utilisateur présent sur le Zumo est également répliqué sur l'adaptateur Pico-Zumo.

L'état du bouton poussoir est contrôlé par une machine à état fini ce qui permet de déterminer avec précision la moment où le bouton est pressé ou relâché.

La détermination précise de ces états très utile sur un robot en mouvement.

Bouton enfoncé

Le robot peut être arrêté lorsque **le bouton est enfoncé**, donc le plus tôt possible sur l'action du bouton.

Le script ci-dessous détecte lorsque le bouton utilisateur est enfoncé (aussitôt qu'il est pressé et avant même qu'il soit relâché) puis affiche l'invite REPL pour saisir la ligne suivante.


```
>>> from zumoshield import *
>>> z = ZumoShield()
>>> z.button.waitForPress()
```

Bouton relâché

Le robot peut être activé **lorsque le bouton est relâché** (pas au moment où le bouton est enfoncé car le doigt est encore sur le bouton).

Le script ci-dessous détecte lorsque le bouton utilisateur est relâché puis affiche l'invite REPL pour saisir la ligne suivante. Le bouton doit donc être enfoncé lorsque la méthode `waitForRelease()` est appelée.

```
>>> from zumoshield import *
>>> z = ZumoShield()
>>> z.button.waitForRelease()
```

 *Si le bouton n'est pas enfoncé (donc déjà relâché) alors la méthode `waitForRelease()` est exécuté instantanément.*

Cette méthode permet aussi d'interrompre l'exécution d'une boucle de traitement pendant que le bouton utilisateur est enfoncé.

Bouton pressé et relâché

Chapitre 6 : Tester

Si le point précédent présente un intérêt limité, la combinaison de `waitForPressed()` et `waitForReleased()` permet d'attendre que le bouton utilisateur soit pressé puis relâché avant de rendre la main au script appelant.

C'est exactement ce que fait la méthode `waitForButton()`.

```
>>> from zumoshield import *
>>> z = ZumoShield()
>>> z.button.waitForButton()
```

L'invite de commande REPL est à nouveau disponible après `waitForButton()` après que le bouton soit pressé et relâché.

7.Tension des piles

La tension des piles s'obtient par l'intermédiaire de l'entrée analogique branchée sur le GPIO 28 (A0 du Pico) si le cavalier est bien en place.

Pour rappel, une entrée analogique accepte une tension maximale de 3,3V. La tension du bloc pile est divisé par deux à l'aide d'un pont-diviseur de tension avant d'être présenté sur GPIO 28.

Les quelques lignes ci-dessous indique comment acquérir la tension analogique sur GPIO28 puis la multiplier par deux pour obtenir la tension du bloc pile.

```
>>> from machine import ADC
>>> a = ADC(28)
>>> # tension ADC en millivolts
>>> mv = 3300 * a.read_u16() / 65535
>>> mv
2393.413
>>> # tension des piles en millivolts
>>> vbat = mv * 2
>>> vbat
4786.826
```

La tension sur l'entrée analogique est de 2393 mV (2,393 V). La tension du bloc pile est obtenue en multipliant cette tension par deux.

Ainsi la tension du bloc pile est finalement de 4,786 Volts (4786 mV).

Il est possible de retrouver cet script dans le dépôt du projet sous le nom `examples/test_vbat.py`.

8.Piezo Buzzer

Le piezo buzzer est utilisé pour produire des mélodies et des alertes sonores. Mélodies et alertes sont encodées dans une chaîne de caractères tel que décrit dans le précédent chapitre (cf. Bibliothèque Zumo Robot, Détail des classes. Voir point « Classe PololuBuzzer »).

L'exemple de script ci-dessous reprend les alertes sonores disponibles directement sur la classe `ZumoShield`.

```
>>> from zumoshield import *
>>> z = ZumoShield()
>>> z.play_blip()
>>> z.play_2tones()
>>> z.play_done()
```

Chapitre 6 : Tester

L'exemple ci-dessous permet de produire une série de notes de Do à Do (sur l'octave suivant).

```
>>> from zumoshield import *
>>> z = ZumoShield()
>>> z.buzzer.play( "18 cdefgab>c" )
```

Le script dessous reproduit quelques mesures d'une fugue.

```
>>> from zumoshield import *
>>> z = ZumoShield()
>>> fugue = "! 05 L16 agafaea dac+adaea fa<aa<bac#a dac#adaea f"
>>> fugue += "06 dcd<b-d<ad<g d<f+d<gd<ad<b- d<dd<ed<f+d<g
d<f+d<gd<ad"
>>> z.buzzer.play( fugue )
```

Il est également possible de consulter l'exemple suivant disponible dans le dépôt du projet : `examples/test_play.py`.

9.Centrale inertielle

La bibliothèque de la centrale inertielle n'est pas intégrée au coeur du `ZumoShield` mais accessible en tant que pilote séparé.

Le `ZumoShield` expose le bus I2C qui sera passé en paramètre à la classe communiquant avec la centrale inertielle.

```
>>> from zumoshield import *
>>> from zumoimu import *
>>> z = ZumoShield()
>>> imu = ZumoIMU( z.i2c )
```

Par la suite, il sera possible d'appeler les différentes méthodes

9.1.Magnétomètre

Le magnétomètre permet de détecter l'intensité du champs magnétique selon les 3 axes X, Y , Z.

9.1.1.Lecture brute

La lecture du magnétomètre passe par la méthode `read_mag()` qui rapatrie les données dans le propriété `m`.

```
>>> from zumoshield import *
>>> from zumoimu import *
>>> z = ZumoShield()
>>> imu = ZumoIMU( z.i2c )
>>> imu.read_mag()
>>> print( imu.m )
<Vector 1024,337,-15637>
>>> imu.read_mag()
>>> print( imu.m )
<Vector 2157,712,-7968>
```

la propriété `m` contient un vecteur enregistrant les informations pour les axes `x`, `y` et `z`. Il est ainsi possible de récupérer directement les valeurs par l'intermédiaire des propriétés `x`, `y` et `z`.

```
>>> imu.m.x
2157
```

```
>>> imu.m.y
712
>>> imu.m.z
-7968
```

9.1.2. Lecture boussole

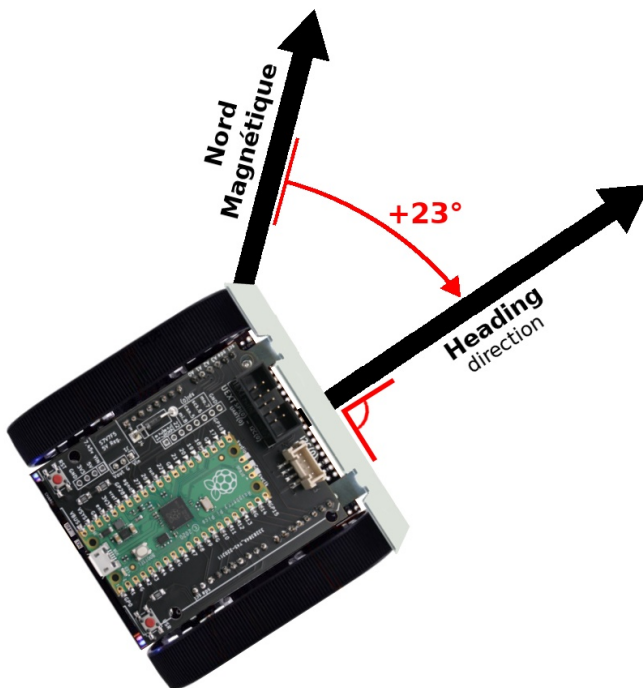
Le magnétomètre permet également de détecter le champs magnétique terrestre, ce qui permet d'utiliser le Zumo Robot comme une boussole.

Si le robot ne ferait pas une boussole très pratique à l'usage, par contre, cela permet au robot de s'orienter dans une direction relativement précise (ou de tourner selon un angle donné).

La classe `Compass` s'appuie sur la centrale inertielle qu'elle interroge afin d'obtenir les données nécessaires à sa calibration et la détection du champs magnétique.

👉 Comme pour le capteur de ligne, il est nécessaire de détecter les maximas et minimas du champs magnétique dans les 3 axes de l'espace. Cette phase de calibration effectue une série de mesures (70 par défaut) pendant que le robot tourne sur lui-même.

Le script ci-dessous met en œuvre la classe `Compass` (également disponible dans `zumoiimu.py`) pour détecter l'angle du Zumo Robot par rapport au nord magnétique.



06RI20 – Heading : orientation par rapport au nord magnétique

L'exemple ci-dessous est disponible dans le dépôt du projet sous le nom `examples/test_compass.py`.

```
from zumoshield import *
from zumoiimu import *
import time
```

```
SPEED = 100

z = ZumoShield()
imu = ZumoIMU( z.i2c ) # Centrale inertielle
compass = Compass( imu ) # Boussole

print("Presser bouton pour calibrer!")
z.play_blip()
z.button.waitForButton()

# Tourner en rond + Calibration
print("Calibration...")
z.motors.setSpeeds( SPEED, -SPEED )
compass.calibrate()

# calibration terminée
z.motors.stop()
z.play_blip()
print("calibration fait !")


# minima et maxima de la calibration
print( 'Boussole min: %s ' % compass.min )
print( 'Boussole max: %s ' % compass.max )

while True:
    # Angle par rapport au nord.
    # moyenne sur 10 lectures
    heading = compass.average_heading()
    print( 'orientation : %s degrees' % heading )
    # Attendre 1/2 seconde
    time.sleep( 0.5 )
```

Etant donné que le script utilise une boucle infinie, il faudra presser [CTRL] + C pour arrêter celui-ci.

Voici les informations que le script affiche dans la session REPL :

```
...
calibration fait !
Compass min: <Vector 371,273,32767>
Compass max: <Vector 1933,2195,-32767>
Heading 328.7318 degrees
Heading 332.3544 degrees
Heading 330.4054 degrees
Heading 348.2336 degrees
Heading 0.8806099 degrees
Heading 6.805615 degrees
Heading 7.23198 degrees
Heading 9.20027 degrees
```

 Avec le Zumo au repos sur le bureau, il est possible de constater une certaine variabilité dans les mesures du Nord Magnétique. La valeur relevée peut ainsi osciller de $\pm 2^\circ$ à $\pm 3^\circ$ en fonction de l'environnement (alimentations, pièces métalliques, appareils générant des champs magnétiques locaux, aimant parasites dans l'environnement).

9.2.Accéléromètre

L'accéléromètre permet de détecter des accélérations et décélérations. Il permettra aussi de détecter des chocs lorsque le robot rencontre un objet (forte décélération) ou lorsqu'il est percuté par un autre robot (brusque accélération).

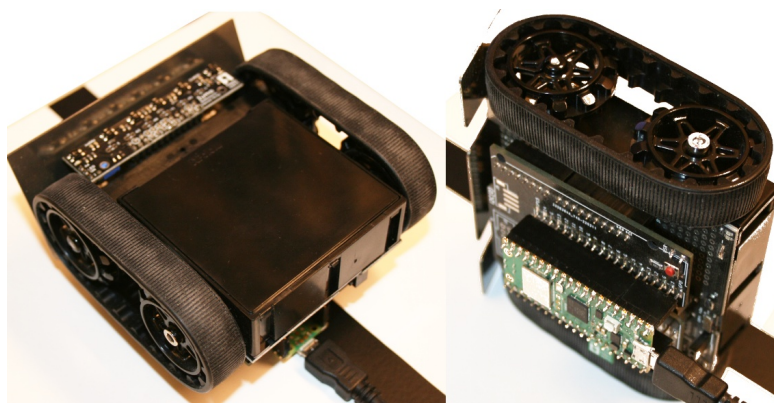
9.2.1. Lecture brute

La lecture de l'accéléromètre passe par la méthode `read_acc()` qui rapatrie les données dans le propriété `a`. La propriété `a` est un vecteur (classe `Vector`) reprenant les données selon les 3 axes X, Y et Z.

L'accélération terrestre produit une donnée d'accélération de l'ordre de 16000.

```
>>> from zumoshield import *
>>> from zumoimu import *
>>> z = ZumoShield()
>>> imu = ZumoIMU( z.i2c )
>>> # Zumo sur ses chenilles
>>> imu.read_acc()
>>> imu.a
<Vector 13,-180,16587>
>>>
>>> # Zumo sur le dos
>>> imu.read_acc()
>>> imu.a
<Vector -541,538,-16140>
>>>
>>> # Zumo sur la chenille gauche
>>> imu.read_acc()
>>> imu.a
<Vector 86,-16374,7>
```

Lorsque le Zumo Robot est à plat sur le robot, le vecteur d'accélération retourné est semblable à `<Vector 13,-180,16587>` où les valeurs sur l'axe X et Y (deux premières valeurs) sont presque nulle. L'axe Z en troisième position reprend la valeur correspondant à l'accélération terrestre (g).



06RI21 – Positions de test de l'accéléromètre

Lorsque le Zumo est placé sur le dos, l'accélération terrestre traverse l'accéléromètre toujours selon l'axe Z mais dans l'autre sens. En conséquence, le signe en position Z est inversé en préservant des valeurs similaires, ce qui est identifier sur le vecteur `<Vector -541,538,-16140>`.

Enfin, en couchant le Zumo sur sa chenille gauche, l'accélération terrestre est reporté sur l'axe Y de l'accéléromètre. Cela produit donc un vecteur `<Vector 86,-16374,7>` avec la 2^{ème} position à -16000 tandis que les axes X et Y sont presque nulle (étant donné que l'attraction terrestre n'influence pas ces axes).

Interpréter les valeurs d'accélération

L'information contenu dans le vecteur correspond au résultat d'un convertisseur numérique, donc sans unité !

L'accéléromètre est configuré par la bibliothèque pour réaliser des mesures dans la gamme +2g à -2g. Le composant retourne une valeur binaire sur 16 bits dans lequel est codé un entier signé.

Un bit est utilisé pour coder le signe et les 15 autres bits concernent la valeur entière. Sur 15 bits, la valeur maximale encodable est de 32767 pour une gamme 0 à 2g (ou encore 0 à $2 * 9,81\text{m/s}^2$). Le 16bits étant utilisé pour le signe.

Sur le vecteur `<Vector 13, -180, 16587>` l'axe Z est à 16587. Avec une simple règle de trois, il est possible de trouver le nombre de g auquel cela correspond.

Accélération = $(2g / 32767) * 16587 = 1,012g$ soit 1 fois l'accélération terrestre, ce qui est parfaitement normal puisque le Robot Zumo repose de manière statique.

Pour connaître la valeur de l'accélération, il faut remplacer la variable g par $9,81\text{m/s}^2$. Le résultat mathématique final serait donc $1,012 * 9,81 = 9,93 \text{ m/s}^2$.



Cette légère variation du vecteur d'accélération est provoqué par l'imprécision sur les calculs mathématiques car l'accélération terrestre reste constante .

En cas de choc (ou arrêt brutal) un autre vecteur d'accélération viendra se combiner au vecteur « g ». Dans pareil cas, le vecteur d'accélération final aura inévitablement des composantes significativement plus élevés sur les autres axes X et Y. Z pourrait aussi être impacté si le robot décolle du sol ou s'il s'y enfonce.

9.2.2.Détection de choc

L'accéléromètre permet également de détecter un choc et sa provenance.

Le principe de détection est simple :

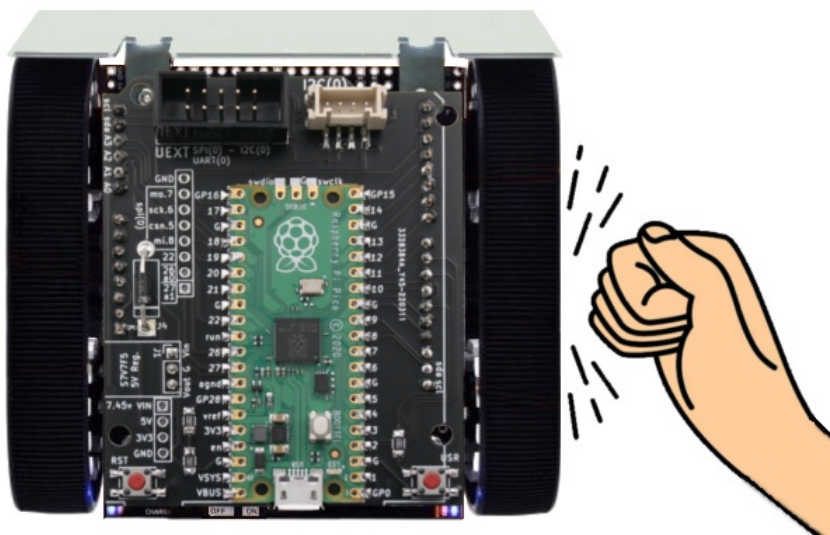
1. Une série de lecture est effectuée sur les axes X et Y de l'accéléromètre pour calculer une valeur moyenne au repos (le Zumo Robot ne doit pas bouger).

2. Une valeur seuil est déterminée (dit *Threshold* en anglais) de sorte que **si** l'accéléromètre retourne une valeur supérieure à la valeur moyenne + le seuil **alors** c'est que le Zumo Robot a reçu un choc (une accélération importune) .

3. Pour éviter les faux positifs ou influence d'une mise en marche du robot, la valeur de l'accélération est amortie en calculant l'accélération moyenne sur les dix derniers échantillons.

4. Enfin, en cas de débordement, il suffit d'inspecter les composantes X et Y du « vecteur_d'accélération_moyen – valeur_de_repos » résultant pour déterminer l'origine du choc sur les axes x et y.

L'exemple ci-dessous est disponible dans le dépôt du projet sous le nom `examples/test_knock.py` .



06R123 – Détection de choc

```

01: from zumoshield import *
02: from zumoimu import *
03: import time
04:
05: z = ZumoShield()
06: imu = ZumoIMU( z.i2c )
07:
08: class MeanEval:
09:     def __init__( self, th_x = 20, th_y = 20 ):
10:         self.pos = 0
11:         self.x_val = [0,0,0,0,0,0,0,0,0,0]
12:         self.y_val = [0,0,0,0,0,0,0,0,0,0]
13:         self.th_x = th_x
14:         self.th_y = th_y
15:
16:     def push( self, x, y ):
17:         self.x_val[self.pos] = x
18:         self.y_val[self.pos] = y
19:         self.pos += 1
20:         if self.pos > 9:
21:             self.pos = 0
22:
23:     def mean( self ):
24:         return ( sum(self.x_val)/10, sum(self.y_val)/10 )
25:
26:     def is_knock( self, x, y ):
27:         x_mean, y_mean = self.mean()
28:         x_delta, y_delta = 0, 0
29:         if abs( x-x_mean )>self.th_x:
30:             x_delta = x - x_mean
31:         if abs( y-y_mean )>self.th_y:
32:             y_delta = y - y_mean
33:         return x_delta, y_delta
34:
35: print( 'Don t move while calibrating!' )
36: print("Press the button to start calibration!")
37: z.button.waitForButton()
38: print("Starting calibration...")
39: mean_eval = MeanEval( th_x=1000, th_y=1000 )
40: for i in range(10):
41:     imu.read_acc()
42:     mean_eval.push( imu.a.x, imu.a.y )
43: print("calibration done")
44:

```

Chapitre 6 : Tester


```
45:
46: print( 'Knock it' )
47: while True:
48:     imu.read_acc()
49:
50:     xdelta,ydelta = mean_eval.is_knock(imu.a.x, imu.a.y)
51:     if (xdelta==0) & (ydelta==0):
52:         mean_eval.push( imu.a.x, imu.a.y )
53:         continue
54:
55:     print( '-----' )
56:     if xdelta < 0:
57:         print( "FRONT knock" )
58:     elif xdelta > 0:
59:         print( "BACK knock" )
60:
61:     if ydelta < 0:
62:         print( "LEFT knock" )
63:     elif ydelta > 0:
64:         print( "RIGHT knock" )
65:
66:     time.sleep( 0.100 )
```

Une fois exécuté, le script qui produit le résultat suivant dans la session REPL :

```
>>> import test_knock
Don t move while calibrating!
Press the button to start calibration!
Starting calibration...
calibration done

Knock it
-----
FRONT knock
-----
FRONT knock
-----
BACK knock
-----
FRONT knock
-----
RIGHT knock
-----
FRONT knock
LEFT knock
-----
LEFT knock
```

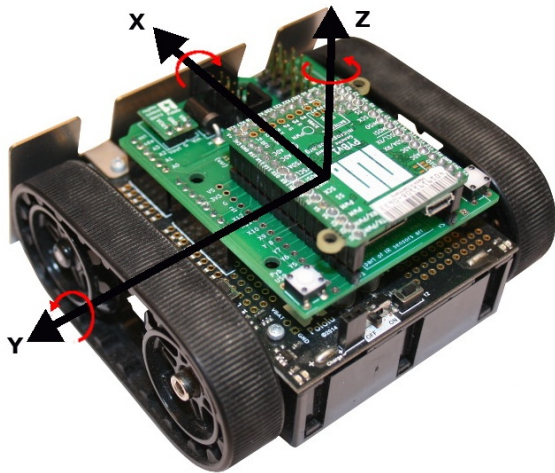
L'un des résultats annonce « FRONT knock » et « LEFT knock » dans une même section signifie simplement que le choc est venu du coin avant-gauche.

 *Frapper (ou donner une pichenette) sur le côté droit du Zumo provoquera une accélération sur l'axe Y. Il faut garder à l'esprit que le Zumo est monté sur des chenilles en caoutchouc il aura donc tendance à vouloir revenir en place (ce qui produira une accélération dans l'autre sens au retour).*

Voici quelques détails concernant le fonctionnement du script :

- Lignes 1 à 3 : import des modules nécessaires.
- Ligne 5 : création de l'instance du ZumoShield permettant de commander le robot.
- Ligne 6 : création de l'instance ZumoImu permettant d'interagir avec les composants de la centrale inertielle (via le bus I2C du ZumoShield).

- Lignes 8 à 33 : déclaration de la classe MeanVal permettant d'accumuler jusqu'à des échantillons (vecteur X, Y) et de calculer la valeur moyenne. Cette classe sera abordée plus en détails juste après le script principal.
- Lignes 34 et 35 : début du script principal avec un message indiquant qu'il ne faut pas déplacer le Zumo Robot pendant la calibration.
- Ligne 36 : attendre la pression (et relâchement) du bouton utilisateur.
- Ligne 39 : création de l'objet mean_eval permettant de calculer des valeurs moyennes sur 10 derniers échantillons de données qui y sont poussé. Le seuil utilisé pour la détection des chocs est fixé à 1000 sur les axes X et Y (soit $2 * g * 1000 / 65535 = 0,299 \text{ m/s}^2$ soit environ 1/30 de l'accélération terrestre).
- Ligne 40 à 42 : échantillonnage des 10 premières accélérations alors que le Zumo Robot est au repos. L'accélération est échantillonnée à l'aide de `imu.read_acc()` puis les données poussées dans le calculateur de moyenne à l'aide de `mean_value.push()`. Au terme de cette procédure, l'objet `mean_eval` connaît la moyenne de l'accélération au repos pour les axes X et Y. Ces valeurs devraient être proche de 0 m/s^2 sauf si le Robot Zumo n'est pas à plat (mais sur une surface en pente).
- Lignes 43 et 46 : messages indiquant la fin de la calibration et le début des mesures.
- Ligne 47 : boucle infinie qui exécute continuellement les instructions entre les lignes 48 et 66. Il s'agit là du corps du programme détectant les chocs. Presser la combinaison de touches [Ctrl]+C pour interrompre le fonctionnement du script.
- Ligne 48 : acquisition d'un échantillon de données sur l'accéléromètre.
- Ligne 49 : la fonction `mean_value.is_knock(imu.a.x, imu.a.y)` calcule la différence entre les données de l'accéléromètre passé en paramètre et la moyenne actuelle de l'accélération connue par `mean_value`. Si cette différence est supérieure au seuil (`th_x` ou `th_y` en fonction de l'axe concerné) alors la méthode `is_knock()` retourne la différence par rapport à la moyenne pour l'axe concerné. Si la différence est inférieure au seuil alors la fonction retourne 0 pour l'axe concerné. La méthode `is_knock()` **retourne deux valeurs** sous forme d'un tuple. Ces valeurs sont respectivement stockées dans les variables `xdelta`, `ydelta`.
- Ligne 51 : s'il existe pas d'écart d'accélération significatif alors `xdelta` et `ydelta` sont égales à zéro. Le test en ligne 51 **est vrai s'il n'y a pas de choc**, ce qui exécute les lignes 52 et 53 sont exécutées.
- Ligne 52 : nous ne sommes pas en situation de choc mais bien d'un mouvement normal du Zumo. Les données d'accélération sont ajoutées dans `mean_eval()` pour que la valeur moyenne tend vers la situation réelle. En effet, si le Zumo passe d'une surface horizontale vers une pente alors les composantes X et Y du vecteur d'accélération vont progressivement changer. Il faut adapter la moyenne vers cette nouvelle position.
- Ligne 53 : Puisqu'il n'y a pas de choc, l'instruction `continue` démarrer immédiatement l'itération de la boucle `while` (à la ligne 47).
- Ligne 55 : Si cette ligne est exécutée c'est qu'il y a une détection de choc sur l'axe X ou l'axe Y (voire les deux axes). L'instruction `print()` affiche une série de traits servant à indiquer une nouvelle section.



06RI24 – Orientation des axes de la centrale inertielle

- Lignes 56 et 57 : Si la valeur de `xdelta` est négative c'est que l'accélération détectée est dans le sens opposé de l'axe X. Donc de l'avant vers l'arrière. Le choc a eu lieu à l'avant.
- Lignes 58 et 59 : Sinon, si la valeur de `xdelta` est positive alors c'est que l'accélération est dans le même sens que l'axe X. Cela n'est possible que si le choc provient de l'arrière.
- Lignes 61 à 64 : détections identiques aux lignes 56 à 59 mais pour l'axe Y (en utilisant `ydelta`).
- Ligne 66 : pause de 100 millisecondes (1/10 sec) laissant le temps à l'accélération de cesser et donc de ne pas influencer le calcul de la moyenne. La boucle de traitement reprend ensuite l'itération suivante

Classe MeanVal

Le but de cette classe est de :

- 1.permettre de pousser des valeurs pour les axes X et Y avec la méthode `push()`
- 2.ne retenir que les dix dernières occurrences pour calculer la moyenne des valeurs
- 3.identifier les seuils de détection à utiliser sur chaque axe (`th_x` et `th_y`, `th` étant le diminutif du terme anglais *threshold*).
- 4.détecter le dépassement des seuils par rapport à la moyenne (sinon retourner 0) pour chaque axe grâce à la méthode `is_knock()` qui peut être traduit pas « est frappé ».

- Ligne 8: déclaration de la classe `MeanVal` .
- Lignes 9 à 14 : déclaration du constructor `__init__()` avec les paramètres de seuil pour l'axe X avec `th_x` et l'axe Y avec `th_y` .
- Ligne 10 : initialisation de la propriété `pos` à 0. Cette propriété est un index de 0 à 9 identifiant le dernier échantillon enregistré par `MeanVal`.
- Lignes 11 et 12 : déclaration des listes `x_val` et `y_val` destinées à mémoriser les 10 couples de valeur x et y. Ces listes sont par ailleurs pré-initialisées avec dix valeurs 0.
- Lignes 13 et 14 :mémorisation des paramètres de seuils dans les propriétés `th_x` et `th_y` .

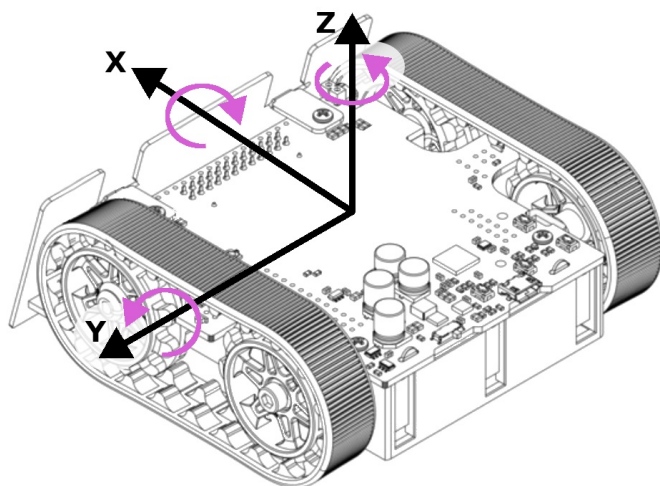
Chapitre 6 : Tester

- Lignes 16 à 21 : définition de la méthode `push(x, y)` permettant d'ajouter un nouveau couple de valeurs `x` et `y` pour le calcul de moyenne.
- Ligne 17 : stocker la valeur `x` dans la liste `x_val` à l'index indiqué par la propriété `pos` (l'index dans la liste).
- Ligne 18 : procédé identique à la ligne 17 mais pour la valeur `y`.
- Ligne 19 : Après le stockage des valeurs `x` et `y`, l'index de stockage `pos` est incrémenté pour passer à la position suivante.
- Lignes 20 et 21 : Si l'index `pos` dépasse la valeur 9 alors il est remplacé en première position (`pos = 0`).
- Lignes 23 et 24 : déclaration et implémentation de la méthode calculant les moyennes pour les listes `x_val` et `y_val`.
- Ligne 24 : la méthode retourne un tuple de deux valeur avec une instruction similaire à `return (val1, val2)` . La fonction `sum()` permet de calculer la somme d'une liste et comme celle-ci contient 10 valeurs, le résultat est divisée par 10.
- Lignes 26 à 33 : définition et implémentation de la fonction `is_knock(x, y)` qui détecte l'écart de `x` et `y` par rapport aux valeurs moyennes et retourne cette différence sous forme d'un tuple avec `return (x_delta, y_delta)` .
- Ligne 27 : calcul de la moyenne des 10 derniers échantillons et stockage des valeurs respectives dans `x_mean` et `y_mean` .
- Ligne 28 : préparation des variables `x_delta` et `y_delta` destinées à recevoir l'écart par rapport à la valeur moyenne. Ces variables sont initialisées à 0 équivalent à « pas de dépassement de seuil ».
- Lignes 29 et 30 : calcul de l'écart entre la valeur actuelle de `x` (communiqué en paramètre) et la valeur moyenne. Ce calcul se fait en valeur absolue à l'aide de la fonction `abs()` pour ne pas tenir compte du signe. Si l'écart est supérieur au seuil `th_x` alors la variable `x_delta` est initialisée et contient la différence (signée cette fois). Tout écart inférieur au seuil `th_x` ne modifie pas la variable `x_delta` qui est donc à 0 dans ce cas.
- Lignes 31 et 32 : calcul équivalent aux lignes 29 et 30 mais appliqué au variable `y`, et `th_y`.

9.3.Gyroscope

Le gyroscope permet de détecter la rotation angulaire du Zumo sur ces axes X,Y et Z. Il permettra au Zumo Robot de détecter un changement d'orientation inopiné alors même que le robot est au repos (ce qui peut arrivé lorsque le robot est percuté).

Le gyroscope n'est pas un périphérique des plus employés dans les procédés de détection de véhicule terrien. Il n'empêche qu'il s'agit d'un élément très intéressant à étudier.



06RI25 – Axes et sens giratoire dy Gyroscope

La **lecture brute** du gyroscope par la méthode `read_gyro()` qui rapatrie les données dans le propriété `g` (*gyroscope*). La propriété `g` est un vecteur (classe `Vector`) reprenant les données selon les 3 axes X, Y et Z.

```
z = ZumoShield()
imu = ZumoIMU( z.i2c )

print( "Gyroscope rotation reading" )
while True:
    imu.read_gyro()
    print( "x = %5i, y = %5i, z = %5i" %
          (imu.g.x, imu.g.y, imu.g.z) )
    time.sleep( 0.1 )
```

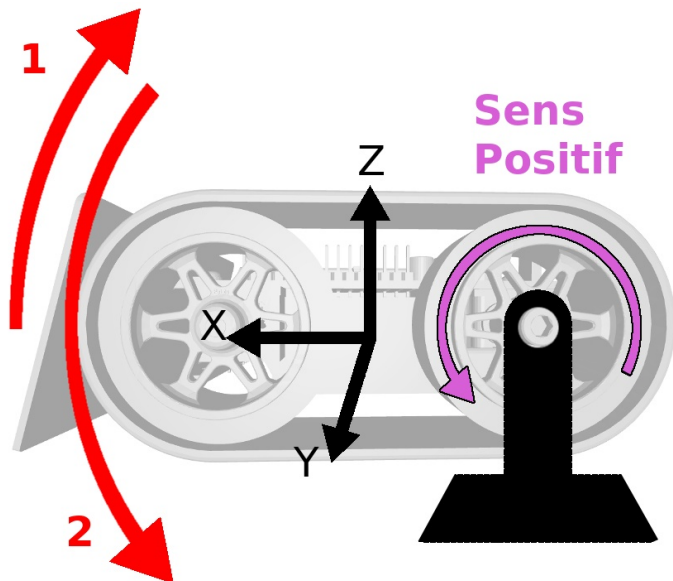
Au repos sur une table, il est possible de constater un bruit non négligeable sur les axes du gyroscope. Ce bruit s'étale de -200 à +200 allant jusqu'à -1000 à +1000 pour l'axe Y.

➡ *Ce bruit notable est relativement peu significatif par rapport aux mesures durant une rotation, ce que démontrera un deuxième exemple ci-dessous.*

```
x = 185, y = -1003, z = -437
x = 184, y = -983, z = -445
x = 181, y = -992, z = -424
x = 213, y = -995, z = -453
x = 196, y = -1005, z = -435
x = 186, y = -995, z = -437
x = 209, y = -1010, z = -445
x = 190, y = -985, z = -435
x = 202, y = -995, z = -451
x = 219, y = -1000, z = -440
x = 208, y = -982, z = -428
x = 207, y = -994, z = -453
x = 174, y = -983, z = -425
x = 196, y = -1002, z = -439
x = 215, y = -986, z = -440
x = 204, y = -998, z = -441
x = 228, y = -1010, z = -443
x = 226, y = -999, z = -437
x = 208, y = -982, z = -437
x = 201, y = -991, z = -438
x = 186, y = -983, z = -435
```

Rotation à vitesse moyenne (ni vif, ni lent) pour relever l'avant du Zumo (la lame) puis rotation pour abaisser l'avant du Zumo vers le bas.

Cela correspond à une rotation autour de l'axe Y similaire à la représentation ci-dessous



06RI26 – rotation angulaire sur l'axe Y.

Le premier mouvement produira une rotation angulaire négative sur l'axe Y tandis que la deuxième rotation se produit dans le sens positif.

```

<<< Début du mouvement 1 >>>
x = 125, y = 196, z = -478
x = -1381, y = -5407, z = 432
x = 3140, y = -22282, z = 402
x = 2623, y = -32338, z = -1335
x = 3268, y = -31856, z = -3691
x = 7512, y = -28781, z = -909
x = 255, y = -10968, z = 2913
x = 2024, y = -741, z = 152
<<< Fin mouvement 1 >>>
<<< Début mouvement 2 >>>
x = 1320, y = 6646, z = 1936
x = -745, y = 16606, z = 4326
x = 5316, y = 14221, z = -2743
x = 2082, y = 24109, z = -3994
x = 2691, y = 32765, z = -5486
x = -566, y = 14052, z = -3142
x = -356, y = 27744, z = -417
x = 4894, y = 32749, z = -85
x = 1372, y = 10083, z = -3510
<<< Fin mouvement 2 >>>
<<< Mouvement 1 (à nouveau) >>>
x = 412, y = -2510, z = -965
x = -395, y = -5698, z = 1221
x = 229, y = -14218, z = 4962
x = -1284, y = -12556, z = 4191
x = -3349, y = -6078, z = 2050
x = -1934, y = -4924, z = 971
    
```

9.3.1. Interpréter les données de rotation

L'information fournie par la bibliothèque sont des données brutes en provenance du gyroscope. Ces données sont des entiers signés encodés sur 16bits.

Un bit étant réservé pour le signe, cela laisse 15 bits pour encoder la valeur maximale (soit un maximum de 32767 aussi bien en positif qu'en négatif).

La bibliothèque configure le gyroscope pour une gamme de mesure de ± 250 dps (degrés par secondes).

Par conséquent, une mesure de 16606 sur l'axe Y correspond à $250 * 16606 / 32767 = 126$ dps (degrés par seconde).

Cela correspond à une rotation d'angle de 120° sur une seconde ou 60° sur 1/2 seconde.

9.3.2. Digital Zero Rate Level

Les mesures au repos ont démontré la présence d'un bruit. Le composant LSM6DS33 (le gyroscope) permet de configurer des mesures sur les gammes $\pm 125/\pm 250/\pm 500/\pm 1000/\pm 2000$ dps (degrés par secondes).

La fiche technique du LSM6DS33 reprend l'information suivante dans la section « caractéristiques techniques ».

G_SoDr	vs. temperature ⁽³⁾	± 1.5	%
LA_TyOff	Linear acceleration typical zero-g level onset accuracy ⁽⁴⁾	± 40	mg
G_TyOff	Angular rate typical zero-rate level ⁽⁴⁾	± 10	dps
LA_OffDr	Linear acceleration zero-g level change vs. temperature ⁽³⁾	± 0.5	mg/ $^\circ$ C
G_OffDr	Angular rate typical zero-rate level change vs. temperature ⁽³⁾	± 0.05	dps/ $^\circ$ C

06RI27 – Niveau d'erreur autour de point zéro

Le paramètre G_TyOff indique l'amplitude de l'erreur autour du point zero (donc lorsque le gyroscope et le Zumo sont au repos). Cette marge d'erreur est évaluée à ± 10 dps (degrés par seconde).

Comme précisé dans le point précédent, la gamme de mesure est de ± 250 dps pour des valeurs de -32768 à +32767 une gamme de mesure de 1 dps équivaut à $32767 / 250 = 130,7$.

Un G_TyOff de 10 dps équivaut donc à une valeur d'erreur au repos de l'ordre de 1300 sur les données produites par le gyroscope. L'erreur sera plus exactement ± 1300 puis que G_TyOff est de ± 10 dps.

L'écart (bruit) constaté lorsque le Zumo est au repos sur un bureau reste donc dans les tolérances annoncées.

9.3.3. Améliorer la qualité des mesures

Pour améliorer la qualité des mesures, il conviendrait d'effectuer un étalonnage du gyroscope.

Cela passerait par le calcul des valeurs moyennes au repos (sur les 3 axes). Ces moyennes étant ensuite soustraites aux données renvoyées par le gyroscope.