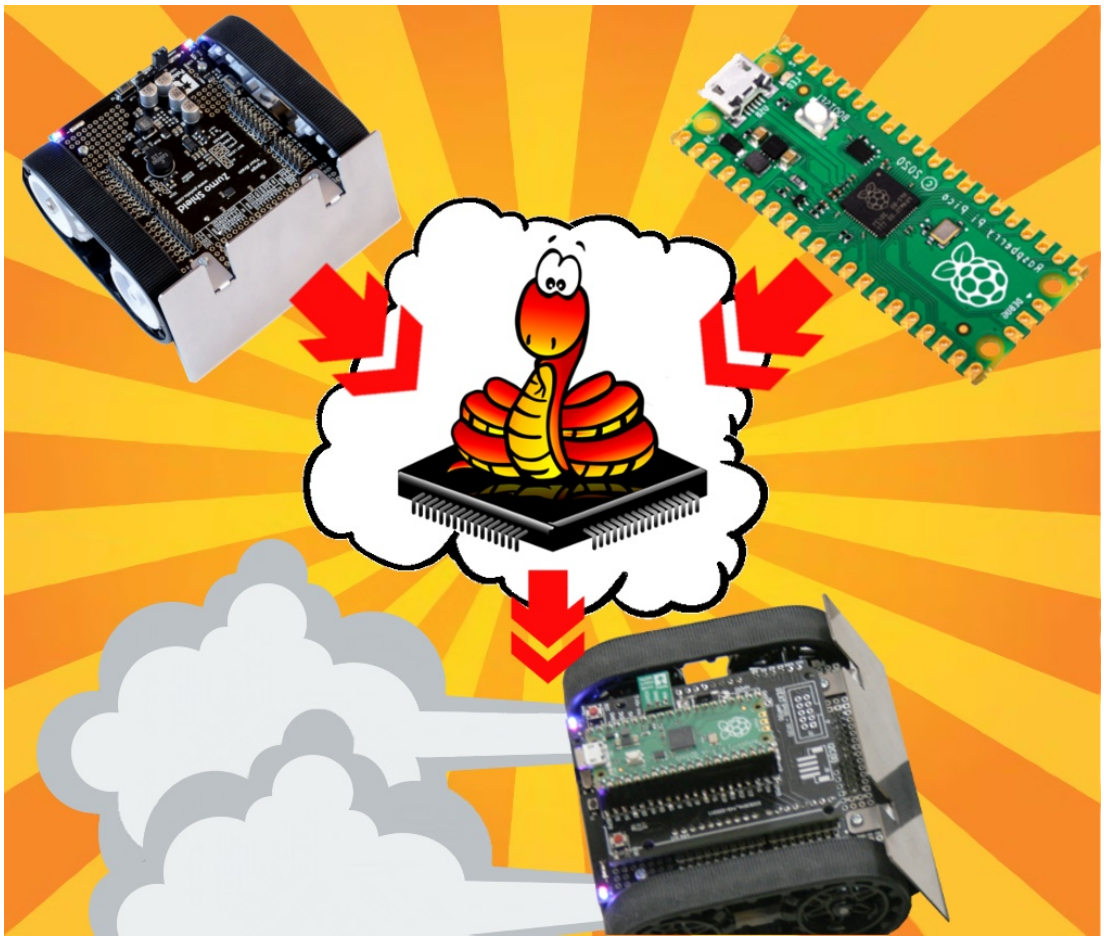
Pico, Zumo Robot et MicroPython

Programmer le Zumo Robot avec Python pour Microcontrôleur



Bibliothèque Zumo Robot

Table des matières

1.Introduction

Après l'exploration du matériel et les détails de branchements, il faut maintenant se tourner vers les éléments logiciels permettant de prendre le contrôle de la plateforme robotique.

Cela peut se faire en suivant deux approches distinctes :

1.Développer une solution de bout en bout

2.Utiliser une bibliothèque pré-existante

Développer une solution de bout en bout

Sans s'appuyer sur du code pré-existant (une bibliothèque), il serait nécessaire de s'attarder sur tous les détails de l'implémentation logicielle.

Si la commande des moteurs seraient relativement simple à mette en oeuvre. Par contre, les capteurs de ligne infrarouges seraient déjà plus ardu à implémenter et les difficultés seraient vraiment difficiles à surmonter pour la centrale inertielle (concepts physiques et mathématiques, protocole de communication, transmission via le bus I2C). Le portage de la bibliothèque d'Arduino (C/C++) vers MicroPython n' d'ailleurs pas contrarié cette observation.

Ces détails techniques seraient très fastidieux et concentreraient, à tord, les efforts sur la création d'une bibliothèque alors qu'il serait tellement plus intéressant de se pencher sur la résolution d'un labyrinthe ou de suivit d'une ligne.

Utiliser une bibliothèque préexistante

L'intérêt immédiat d'une bibliothèque est de pouvoir se lancer plus rapidement dans l'expérimentation sans se soucier sur les détails techniques. La bibliothèque expose des fonctions de haut niveau en masquant toute la complexité du contrôle électronique de la plateforme Zumo robot.

Une bibliothèque permet de se concentrer plus rapidement sur des projets captivants (plus proche de la programmation, moins proche de l'électronique).

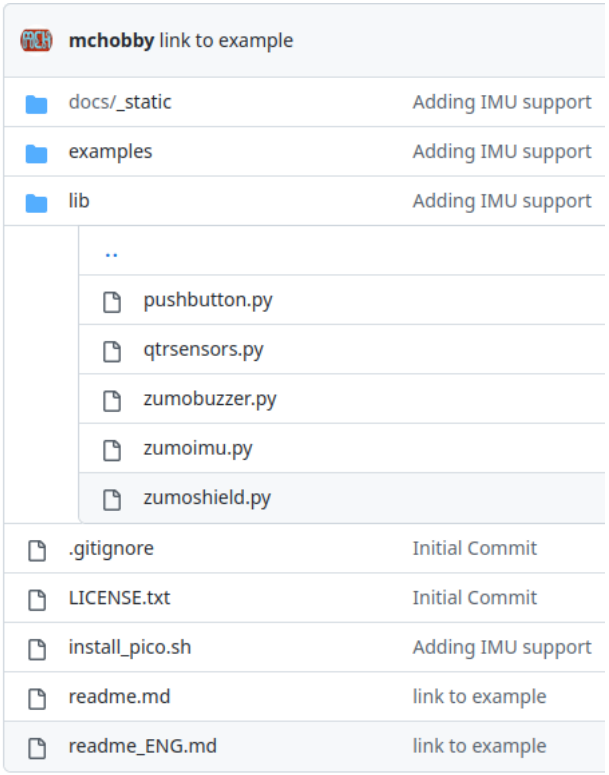
L'inconvénient d'une bibliothèque est qu'il faut néanmoins s'intéresser à son interface de programmation (l'API) pour l'exploiter au mieux.

2.Bibliothèques Zumo Robot

Le Robot Zumo de Pololu dispose d'une bibliothèque Arduino. Cette bibliothèque est **portée sous MicroPython** pour fonctionner avec les microcontrôleur Pyboard et Raspberry-Pi Pico.

La bibliothèque est disponible sur le dépôt GitHub suivant :

https://github.com/mchobby/micropython-zumo-robot



04RI45 – Contenu du dépôt micropython-zumo-robot

Le sous-répétoire `lib` contient les bibliothèques permettant de contrôler tous les périphériques du Robot Zumo.

•**Pushbutton.py** : contient la classe `Pushbutton` permettant d'utiliser un bouton et détecter ses différents états.

•**Zumobuzzer.py** : contient la classe `PololuBuzzer` permettant de produire des sons et jouer des mélodies sur le buzzer du Zumo.

•**Qtrsensors.py** : contient la classe `QTRSensors` destiné à la détection de surface claire/sombre à l'aide de capteurs infrarouges. Cette bibliothèque supporte également une méthode de calibration.

•**Zumoimu.py** : contient les classes `ZumoIMU` et `Vector` prenant en charge la centrale inertielle du Zumo Robot v1.3 .

•**Zumoshield.py** : bibliothèque contenant les classes `ZumoMotor`, `ZumoReflectanceSensorArray` et `ZumoShield`. La classe **ZumoShield est le point d'accès vers toutes les fonctionnalités du Zumo Robot**.

Les différentes bibliothèques détectent automatiquement la plateforme MicroPython (Pyboard ou Pico) utilisée et adaptent automatiquement la configuration des broches.

2.1.Copie locale des bibliothèques

Il est nécessaire de disposer d'une copie des bibliothèques Zomu Robot sur l'ordinateur afin de pouvoir la téléverser sur la plateforme MicroPython cible (Pico ou Pico-W).

Il existe plusieurs méthodes pour obtenir une copie des bibliothèques Zumo Robot sur l'ordinateur.

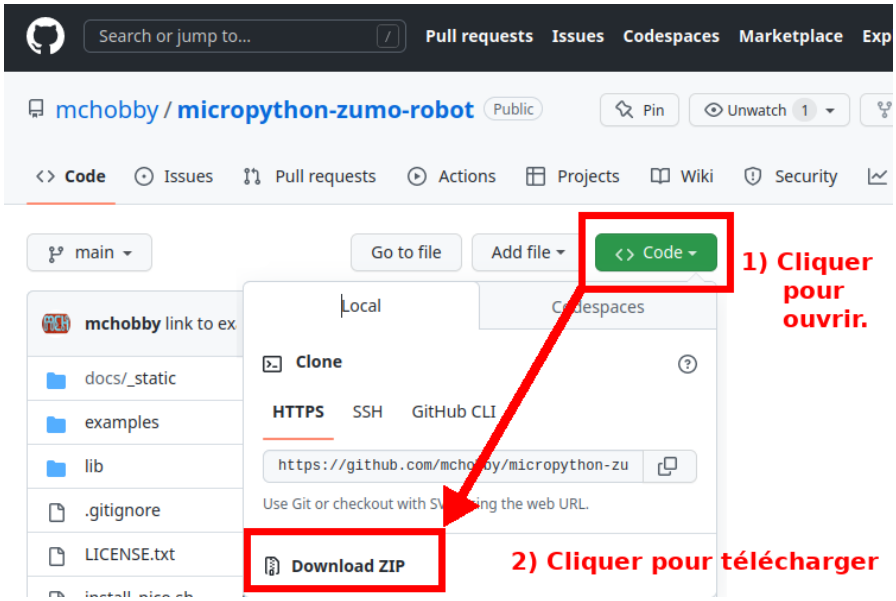
Téléchargement rattaché à l'ouvrage

Une copie du dépôt est disponible dans les éléments de télécharge associé à l'ouvrage.

Téléchargement l'archive ZIP du dépôt

GitHub propose de télécharger la dernière version du dépôt sous forme d'une archive ZIP.

Une fois l'archive extraite, l'utilisateur dispose d'une copie complète du dépôt (donc des sources) sur l'ordinateur.



04RI47 – Télécharger l'archive depuis le dépôt

Clôner le dépôt

Pour les ordinateurs disposant d'un client **Git** (utilitaire open-source, en ligne de commande), il est possible de cloner le dépôt à l'aide de la commande :

`git clone` https://github.com/mchobby/micropython-zumo-robot.git

La commande duplique le contenu du dépôt dans le répertoire courant.

3.Installation de la bibliothèque

Les bibliothèques Robot Zumo `pushbutton.py`, `zumobuzzer.py`, `qtrsensors.py`, `zumoimu.py` et `zumoshield.py` doivent être copiés sur la plateforme MicroPython.

Il est vivement recommandé de créer un sous-répertoire `lib/` sur la plateforme MicroPython pour y placer les bibliothèques. Cela évite de mélanger les scripts utilisateurs et le code source des bibliothèques.

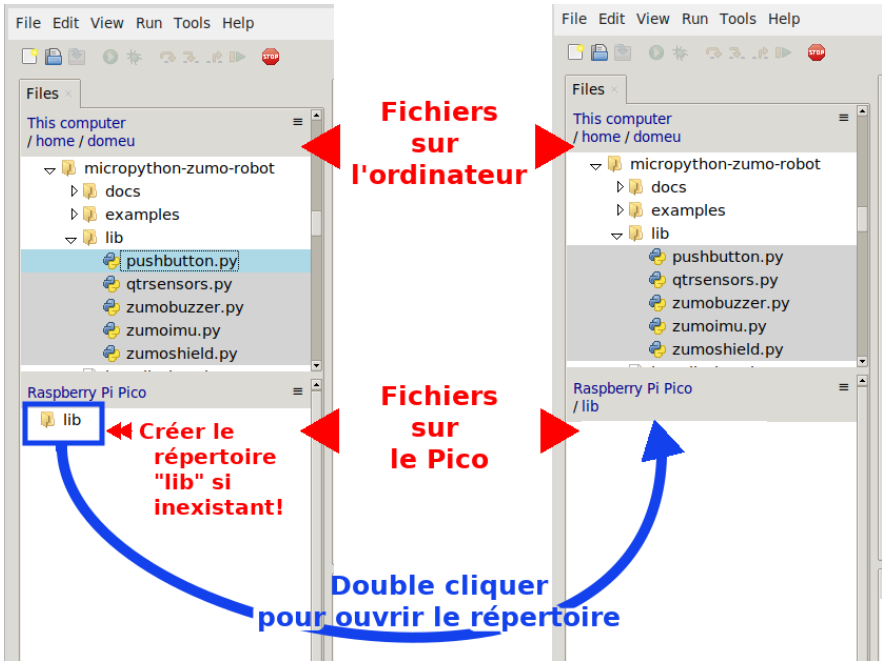
Cette tâche peut être réalisée à l'aide ThonnyIDE (environnement graphique), MPRemote (outils en ligne de commande) ou tout autre outils MicroPython permettant de manipuler le système de fichier MicroPython.

Le firmware MicroPython doit déjà être actif sur le Raspberry-Pi Pico. Une fois ThonnyIDE connecté sur la carte Pico, les bibliothèques sont copiées à l'aide du gestionnaire de fichiers de ThonnyIDE.

Dans la partie haute du gestionnaire (l'ordinateur) : sélectionner le répertoire contenant les bibliothèques Zumo Robot dupliqués sur l'ordinateur.

Dans la partie basse du gestionnaire (le pico) : créer le répertoire `lib/` dans l'arborescence si celui-ci n'existe pas. Cliquer sur l'entrée `lib` permet d'accéder au contenu du répertoire sur le Pico et de lister les fichiers qu'il contient.

La capture ci-dessous présente l'explorateur de fichiers avant et après ouverture du répertoire `lib/` .



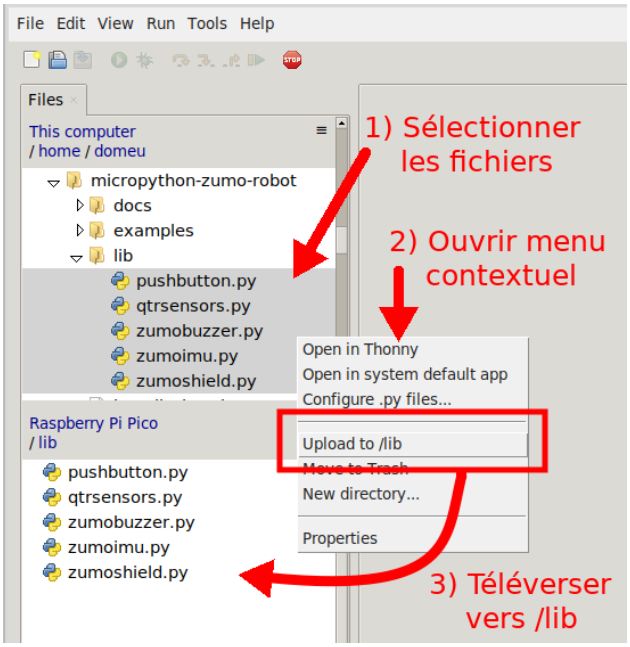
04RI50 – Explorateur de fichiers ThonnyIDE

Il est alors possible de sélectionner les fichiers sur l'ordinateur (utiliser la touche [Shift] pour sélectionner plusieurs fichiers en cliquant avec la souris).

Ensuite, afficher le menu contextuel à l'aide du bouton droit de la souris.

Le menu contextuel propose l'entrée « upload to /lib » permettant de téléverser les fichiers sur le Pico (dans le répertoire `lib`).

Une fois l'opération de copie achevée, les fichiers sont également visible dans la partie inférieure du navigateur de fichiers.

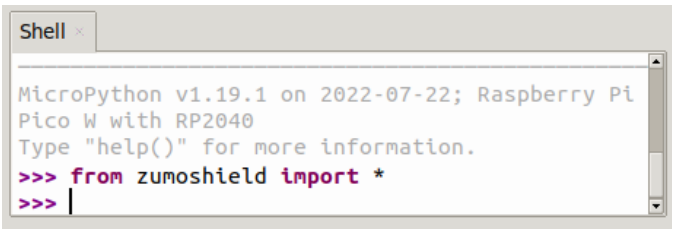


04RI51 – Copie des bibliothèques sur le Pico

Vérifier l'installation des bibliothèques

Pour vérifier l'installation de la bibliothèque, cliquer dans la section « **Shell** » de ThonnyIDE et saisir l'intruction « `from zumoshield import *` ». Cette instruction provoque le chargement et parsing (décodage) de tous les scripts Python de la bibliothèque Zumo Robot.

Cette opération doit se dérouler sans message d'erreur et présenter l'invite de commande pour saisir l'instruction suivante.



04RI52a – chargement de la bibliothèque zumoshield

4.Diagramme des classes

Le schéma suivant reprend le diagramme des différentes classes, propriétés et méthodes disponibles pour piloter le Zumo Robot.

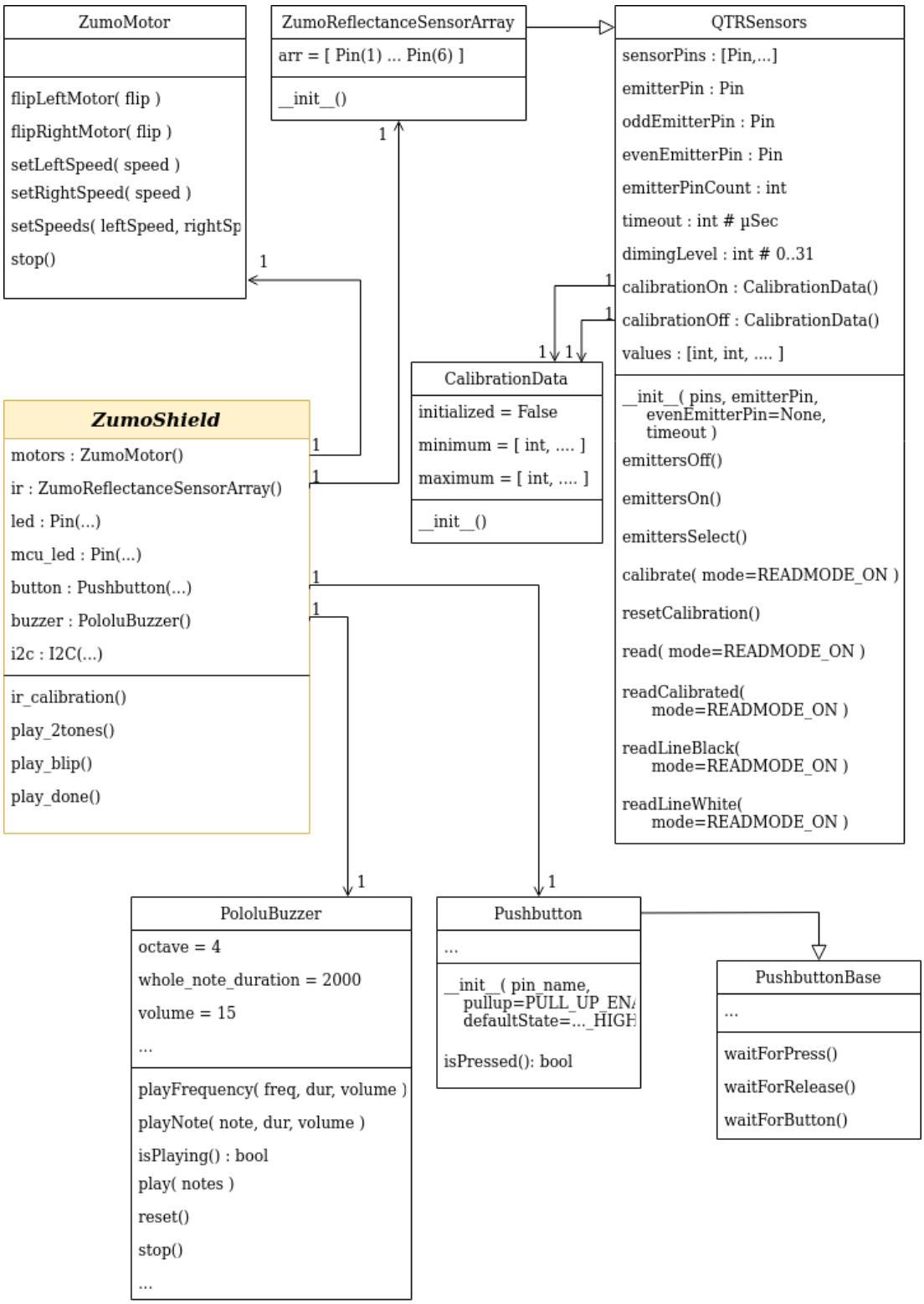
Une fois la classe ZumoShield instanciée (voir script ci-dessous), toutes les fonctionnalités du Zumo sont accessibles via cette instance.

```
from zumoshield import ZumoShield
z = ZumoShield()
```

Le graphe ci-dessous reprend les dépendances entre les différentes classes et les attributs principaux de celle-cis.

Le nom des méthodes et attributs de la bibliothèque Zumo Robot pour MicroPython préserve la nomenclature utilisée dans la bibliothèque Arduino (celle produite par Pololu Ltd). Ainsi cette nomenclature correspond au C, et

non à Python, permettant ainsi aux utilisateurs Arduino de transposer leur savoir d'un langage à l'autre.



04RI55 – Diagramme des classes (réaliser avec Draw.io)

Voici quelques astuces permettant de lire le diagramme des classes.

Chaque classe est un bloc divisé en 3 sections contenant les éléments suivants (de haut en bas) :

•Le nom de la classe

•Les propriétés de la classe. Les propriétés peuvent être vue comme des variables de l'objet.

•Les méthodes de la classe. Les méthodes peuvent être vue comme des fonctions qui agissent sur les propriétés de l'objet.

Certaines classes sont liées entre-elles par une **relation d'héritage**. Cela est indiqué à l'aide d'une **flèche fermée** comme la relation entre `Pushbutton` et `PushbuttonBase`.

•La classe parent (`PushbuttonBase`) se trouve à la pointe de la flèche.

•La classe enfant (ici `Pushbutton`) hérite des propriétés et méthodes de la classe parent (ici `PushbuttonBase`). Ainsi la classe enfant bénéficie du fonctionnement générique de la classe Parent `PushbuttonBase`.

•La classe enfant (ici `Pushbutton`) est destiné à spécialiser la classe parent.

L'**association directe** est représentée par une **flèche ouverte**. C'est le cas entre `ZumoShield` et `ZumoMotors`. Comme l'indique la flèche de `ZumoShield` vers `ZumoMotor`, cela signifie que `ZumoShield` contient une instance de `ZumoMotor`.

•La cardinalité (1) sur la flèche à côté de `ZumoMotor` signifie que `ZumoShield` contient 1 et 1 seule instance de `ZumoMotor`.

•La cardinalité (1) sur la flèche à côté de `ZumoShield` indique d'une instance `ZumoMotor` est liée à 1 et 1 seule instance de `ZumoShield`.

•Enfin, la flèche placée à côté de la propriété `motor` du `ZumoShield` suggère que l'instance `ZumoMotor` est accessible via la propriété `motor`.

5.Détail des classes

5.1.Classe ZumoShield

La classe ZumoShield permet d'accéder à toutes les fonctionnalités du Zumo Robot exception faite de la centrale inertielle (abordée séparément).

Lors de la création de l'instance, appel de la méthode `__init__()`, ZumoShield crée les différentes instances d'objets nécessaires au contrôle de la plateforme Zumo puis met les moteurs du Zumo à l'arrêt.

Propriété motors : ZumoMotor

Cette propriété permet d'accéder à l'instance de la classe ZumoMotor utilisée pour contrôler les moteurs du Zumo.

Propriété ir : `ZumoReflectanceSensorArray`

Cette propriété permet d'accéder au contrôle du détecteur de ligne et des 6 capteurs infrarouge le composant.

Propriété led : Pin

Cette propriété permet de contrôler l'état de la LED utilisateur présente sur le robot Zumo. La propriété retourne une référence vers un objet de type `Pin` dont la méthode `value()` permet de contrôler l'état de la broche (donc de la LED).

Propriété mcu_led : Pin

Cette propriété permet de contrôler l'état de la LED utilisateur du microcontrôleur. La propriété retourne une référence vers un objet pouvant être contrôler comme un objet de type `Pin`.

Sur un **Raspberry-Pi Pico**, la LED utilisateur est contrôlée directement par le GPIO 25.

Sur un Raspberry-Pi **Pico Wireless**, cette LED utilisateur est contrôlée par le GPIO0 du module sans fil. Dans ce second cas, la modification d'état de la LED est une opération nettement plus coûteuse du point de vue traitement.

Propriété button : PushButton

Cette propriété permet d'exploiter le bouton utilisateur présent sur le Zumo Robot (et répliqué sur l'adaptateur Pico-Zumo). La classe `PushButton` permet de détecter l'état du bouton mais aussi de surveiller les changements d'état de celui-ci.

Il est ainsi possible d'attendre que le bouton soit enfoncé ou relâché ou encore enfoncé puis relâché.

Propriété buzzer : PololuBuzzer

Cette propriété permet d'accéder aux services relatif au piezo buzzer du Zumo. Au delà de l'opportunité de jouer un son (une fréquence donnée), la classe `PololuBuzzer` permet aussi de jouer des notes ou une mélodie encodée dans une chaîne de caractères.

Propriété i2c : I2C

Cette propriété permet d'accéder au bus I2C présent sur le robot Zumo. Cette propriété retourne une instance de `machine.I2C` , instance créée lors du premier appel de la propriété `i2c`.

Ce bus I2C est essentiel pour communiquer avec la centrale inertielle à l'aide des classes ad-hoc.

Méthodes play_2tones(), play_blip(), play_done()

Ces méthodes contrôlent l'objet `buzzer` pour jouer des alertes sonores rudimentaires.

La méthode `play_2tones()` produit une notification en deux tons, tandis que `play_blip()` produira une notification d'un seul ton.

La méthode `play_done()` joue une joyeuse petite mélodie de quelques notes. Elle sert a notifier la réussite d'une opération.

Ces méthodes permettent d'implémenter facilement des notifications à destination de l'utilisateur sans avoir à se pencher sur l'encodage des mélodies dans une chaîne de caratères.

Méthode ir_calibration()

La méthode `ir_calibration()` fait partie des méthodes les plus importantes de la bibliothèque. Elle permet de calculer les données de calibration pour les 6 capteurs infrarouges constituant le suiveur de ligne.

Ainsi, il sera possible de connaître la valeur maximale (surface sombre) et minimale (surface claire) pour chacun des capteurs infrarouges. Les capteurs infrarouges étant différent les uns des autres, les maximas et minimas seront donc sensiblement différents d'un capteur à l'autre.

La calibration est un élément important permettant d'améliorer significativement l'efficacité du capteur de ligne.

Le robot Zumo doit être placé au dessus d'une ligne noir de 15mm de large avant de lancer la procédure de calibration.

04RI57 – Zumo Robot au dessus d'une ligne noire

Durant la procédure de calibration, la méthode `ir_calibration()` prend le contrôle des objets `motors` et `ir` . Le Zumo fera des rotations de gauche à droite afin de placer, tour à tour, les différents capteurs infrarouges au dessus de la ligne. L'échantillonnage actif durant cette phase de mouvements permettra de déterminer les différents minimas et maximas.

A noter que les données de calibration sont stockées dans l'instance de la classe `QTRSensors` :

- voir `QTRSensors.calibrationOn.minimum`
- voir `QTRSensors.calibrationOn.maximum`

5.2.Classe ZumoMotor

La classe `ZumoMotor` est la classe la plus importante après `ZumoShield`. Comme son nom le laisse entendre, elle permet de commander les moteurs du Zumo.

Son implémentation est relativement simple et ne propose que quelques méthodes.

Méthode setLeftSpeed(speed)

Permet de fixer la vitesse du moteur gauche à l'aide d'un entier compris entre 400 (marche avant pleine vitesse) et -400 (marche arrière pleine vitesse).

La valeur 0 correspond à un moteur à l'arrêt.

En fonction du signe de la valeur `speed`, la broche MxDIR du DRV8835 sera soit au niveau haut, soit au niveau bas.

En fonction de la valeur absolu de `speed`, le cycle utile de la broche MxPWM ira de 0 % (`speed=0`) à 100 % (`speed=400`).

Méthode setRightSpeed(speed)

Identique à la méthode `setLeftSpeed()` à l'exception que cela s'applique au moteur droit.

Méthode setSpeeds(leftSpeed, rightSpeed)

Cette méthode permet de fixer la vitesse des deux moteurs (gauche et droit) en une seule opération.

Ainsi, `setSpeeds( 100, 100 )` fera avancer le Zumo tandis que `setSpeeds( -100, -100 )` provoquera une marche arrière.

Enfin `setSpeeds( 100, -100 )` fera pivoter le Zumo à droite sur lui-même.

Méthode stop()

Comme son nom le laisse entendre, cette méthode stoppe immédiatement le Zumo.

Méthodes flipLeftMotor(flip), flipRightMotor(flip)

Cette méthode de configuration d'inverser la commande de rotation du moteur gauche ou du moteur droit. Le paramètre `flip` peut avoir la valeur `True` (ou `False` pour rétablir la configuration originale).

Cette configuration est effective dès le premier appel à `setSpeeds()`, `setLeftSpeed()`, `setRightSpeed()` .

5.3.Classes ZumoReflectanceSensorArray et QTRSensors

La classe `ZumoReflectanceSensorArray` est utilisée pour initialiser la définition des capteurs infrarouges sur la plateforme matérielle. La propriété `arr` est une liste reprenant les 6 broches numériques associées aux 6 capteurs infrarouges.

La propriété `arr` contient donc 6 instances de la classe `Pin` créés dans le constructeur (méthode `__init__()`) avant d'être utilisé pour initialiser la classe ancêtre `QTRSensors`.

Les différents services sont assurés exclusivement par la classe ancêtre `QTRSensors`.

Propriété sensorPins :

maintient une référence vers les broches contrôlant les capteurs infrarouges.

Propriété values :

Liste contenant les valeurs obtenues sur les différents capteurs infrarouges. Ces données sont initialisées via les méthodes de lecture `read()`, `readCalibrated()`.

La longueur de cette liste correspond au nombre de capteur infrarouge (donc au nombre de broches rattachées aux capteurs, information communiquée au constructeur de la classe).

Propriété emitterPin :

maintient une référence vers la broche de commande des LEDs infrarouge. Cette référence est utilisée lorsqu'il n'y a pas de distinction entre LEDs paires et LED impaires. C'est le cas du robot Zumo.

Propriété oddEmitterPin, evenEmitterPin :

maintient une référence vers les broches contrôlant les LEDs impaires et les LEDs paires. Ce n'est pas le cas du robot Zumo et ne concerne que les détecteurs de lignes équipés de très nombreux capteurs (30 et plus).

Propriété timeout :

Permet de modifier le `timeout`, temps maximal, utilisé lors de la détection de réflectance par les capteurs infrarouges. Cette valeur est également utilisée lors du processus de calibrage du capteur de ligne.

La valeur par défaut est de 2500 µSec (0,0025 seconde). La gamme utilisable varie entre 0 et 5000 µSec.

Propriété dimmingLevel :

Permet de contrôler la luminosité des LEDs infrarouge durant l'activation du détecteur de ligne. Cette valeur évolue entre 1 et 31 et représente le nombre d'impulsions de 1µSec utilisées pour éclairer la surface dont on mesure la réflectance.

Une valeur dimmingLevel à 0 (valeur par défaut) désactive la fonctionnalité de contrôle de luminosité.

Méthode emittersOff(emitters=EMITTERS_ALL) :

Permet d'éteindre les LEDs infrarouges du détecteur de ligne. Cette méthode prend en charge les combinaisons de LEDs paires, impaires, voir aussi toutes les LEDs.

Pour le robot Zumo la valeur par défaut `EMITTERS_ALL` convient parfairement. Pour d'autres types de capteurs, il est également possible d'utiliser les constantes `EMITTERS_ODD` ou `EMITTERS_EVEN`.

Méthode emittersOn(emitters=EMITTERS_ALL) :

A l'opposé d'`emittersOff()`, cette méthode est utilisée pour allumer les LEDs infrarouges du détecteur de ligne. Les détails du fonctionnement sont identiques à `emittersOff()` .

Méthode emittersSelect(emitters) :

Permet, en une seule opération d'activer les LEDs infrarouges paires (ou impaires) et de désactiver les LEDs impaires (ou paire). Cette méthode est utilisée avec les détecteurs de ligne équipés de nombreux capteurs. Le robot Zumo n'exploite pas cette dernière.

Méthode calibrate(mode=READMODE_ON) :

Effectue une calibration des capteurs infrarouges et stocke les données dans l'instance de `CalibrationData` associé au mode. Le mode par défaut `READMODE_ON` active les LEDs infrarouges durant les mesures et stocke les données dans la propriété `calibrationOn`.

Il est également possible d'utiliser le mode `READMODE_OFF` auquel cas, les LEDs infrarouges sont désactivées et les données stockées dans `calibrationOff`.

Il existe autres modes de calibration mais `READMODE_ON` est celui utilisé avec le capteur de ligne du Zumo.

Durant la calibration, il est important de déplacer le capteur de ligne au dessus d'une ligne noire (15mm) pour pour exposer les différents capteurs infrarouges à des minimas et des maximas durant la qualibration.

Méthode resetCalibration() :

Reinitialise les données contenues dans les propriétés `calibrationOff` et `calibrationOn` .

Méthode read(mode=READMODE_ON) :

Cette méthode effectue une **lecture brute** des 6 capteurs de réflectance présents sur le détecteur de ligne.

Les données sont stockées dans la liste accessible via la propriété `values` . Cette liste contient 6 emplacements destinés à recevoir les données acquises sur les capteurs. Les données sont initialisées dans le même ordre que les capteurs sur le détecteur de ligne.

Le paramètre mode indique à la fonction comment activer les LEDs infrarouges durant la lecture. `READMODE_ON` ou `READMODE_ON_AND_OFF` active automatiquement les LEDs infrarouges durant l'évaluation de la réflectance puis les éteints. Ce paramètre par défaut permet au robot Zumo d'évaluer la réflectance sous le détecteur de ligne.

En lisant l'implémentation de la méthode `read()` dans le script `qtrsensors.py`, il est possible d'y détecter les modes :

- `READMODE_OFF` : lecture en désactivant les LEDs infrarouges.
- `READMODE_ON, READMODE_ON_AND_OFF` : lecture en activant les LEDs Infrarouges
- `READMODE_ODD_EVEN,READMODE_ODD_EVEN_AND_OFF` : activation tour-à-tour des LEDs infrarouges impaires puis paire
- `READMODE_MANUAL` : effectue une lecture sans modifier l'état des LEDs infrarouge. Dans ce cas, c'est la routine appelante qui active les LEDs infrarouge en fonction des besoins spécifiques du projet.

Méthode readCalibrated(mode=READMODE_ON) :

Lorsque les données de calibration sont disponibles (`calibrationOn`, `calibrationOff`), la méthode `readCalibrated()` peut être utilisé à la place de `read()`.

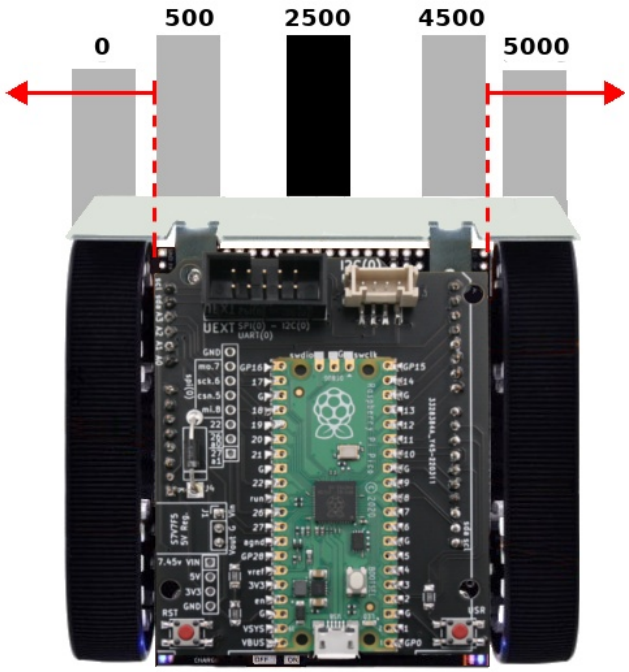
La méthode `readCalibrated()` effectue une opération de normalisation sur les valeurs de réflectance et borne les résultats pour chaque capteur entre 0 et 1000. Ces informations sont accessibles via la propriété `values` .

Méthode readLineBlack(mode=READMODE_ON) :

Cette méthode retourne un entier entre 0 et 5000 permettant d'indiquer la position de la ligne noire (15mm de large sur fond blanc) sous le détecteur de ligne.

Cette méthode propose également les données normalisée des différents capteurs via la propriété `values` .

Il faut donc disposer des données de calibration pour pourvoir utiliser `readLineBlack()` puisque celle ci s'appuie sur `readCalibrated()`.



04RI60 – valeur retournée par readLineBlack()

Voici comment interpréter la valeur numérique retournée par `readLineBlack()` :

•Une ligne noire en dessous du capteur centrale produit une valeur typique de 2500.

•Lorsque la ligne noire se déplace vers la gauche, la valeur tend progressivement vers 500.

•Lorsque la ligne noire se déplace vers la droite du capteur alors la valeur tend vers 4500.

•Lorsque le ligne noire tend à sortie par la droite du Zumo (ou est sortie par la droite) alors la valeur retournée tend vers 5000 qui est la valeur plafond.

•Lorsque la ligne noire se tend à sortir par la gauche du Zumo alors la valeur retournée tend vers 0 (valeur planché).

Méthode readLineWhite(mode=READMODE_ON) :

Cette méthode fonctionne à l'identique de readLineBlack() à l'exception que la ligne est ici blanche (15mm de large) sur fond sombre.

5.4.Classe CalibrationData

La classe `CalibrationData` permet de stocker les données de calibration des différents capteurs infrarouges composant le capteur de ligne.

Propriété initialized

Boolean indiquant si la structure contient des données de calibration. Si c'est le cas, `minimum` et `maximum` contient une liste de valeur (une valeur par capteur). Dans le cas contraire les propriétés `minimum` et `maximum` sont à `None`.

Propriété minimum

Contient une liste des valeurs minimums (en microsecondes) pour chaque capteurs infrarouge. Les temps minimums correspondent à la décharge rapide du capteur de infrarouge (donc la présence d'une surface plus réfléchissante).

Cette liste contient une entrée par capteur (de gauche à droite sur le détecteur de ligne).

Propriété maximum

A l'identique de la propriété `minimum`, cette liste contient les valeurs maximums pour chaque capteur de ligne. Ces valeurs correspondent donc a une surface la moins réfléchissante.

Méthode as_json() et load_json()

Non indiqué sur le diagramme des classes, la classe `CalibrationData` propose les méthodes `as_json()` et `load_json()` permettant respectivement :

•De sauver les informations de configuration au format json dans une chaîne de caractères.

•De recharger la configuration depuis une chaîne de caractères (au format json).

5.5.Classes Pushbutton et PushbuttonBase

Le constructeur `__init_( pin_name, pullup=PULL_UP_EN, defaultState=DEFAULT_STATE_HIGH )` de la classe `Pushbutton` permet d'initialiser la détection des états du bouton en fonction de la configuration matérielle du bouton.

En effet, le bouton utilise la résistance pull-up interne et la broche est à la masse lorsque le bouton est pressé. Par conséquent, l'état par défaut sur la broche du microcontrôleur est un niveau haut (résistance pull-up oblige) et la résistance pull-up doit être activée lors de la configuration de la broche.

Les services de base sont offerts par la classe ancêtre `PushbuttonBase`. Elle prend en charge la machine à état permettant de détecter et mémoriser l'état actuel du bouton (pressé ou non pressé).

Méthode waitForPress()

Méthode **bloquante** qui attend que le bouton soit enfoncé.

Méthode waitForRelease()

Méthode **bloquante** qui attend que le bouton soit relâché.

Méthode waitForButton()

Méthode **bloquante** qui attend que le bouton soit enfoncé puis relâché. Cette méthode permet, par exemple, d'attendre que le bouton utilisateur du Zumo soit pressé pour démarrer le programme utilisateur.

Méthode isPressed()

Cette méthode, implémentée dans classe `Pushbutton`, permet de vérifier si le bouton est actuellement pressé.

Comme la méthode est **non bloquante** et peut être utilisée dans une boucle de traitement pour agir sur son fonctionnement.

5.6.Classe PololuBuzzer

La classe PololuBuzzer est principalement utilisée pour jouer des mélodies décrites dans une chaîne de caractères mais permet également de jouer des notes simples ou une fréquence donnée.

A propos de l'encodage d'une mélodie

Les mélodies sont encodées dans des chaînes de caractères. Ci-dessous, la variable fugue contient une liste de chaîne de caractères reprenant les différents mouvement d'une fugue.

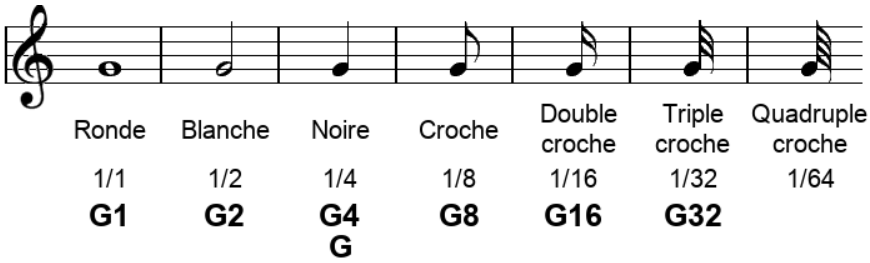
```
fugue = [   "! O5 L16 agafaea dac+adaea fa<aa<bac#a dac#adaea f",
  "O6 dcd<b-d<ad<g d<f+d<gd<ad<b- d<dd<ed<f+d<g d<f+d<gd<ad",
  "L8 MS <b-d<b-d MLe-<ge-<g MSc<ac<a ML d<fd<f O5 MS b-gb-g",
  "ML >c#e>c#e MS afaf ML gc#gc# MS fdfd ML e<b-e<b-",
  "O6 L16ragafaea dac#adaea fa<aa<bac#a dac#adaea faeadaca",
  "<b-acadg<b-g egdgcg<b-g <ag<b-gcf<af dfcf<b-f<af",
  "<gf<af<b-e<ge c#e<b-e<ae<ge <fe<ge<ad<fd",
  "O5 e>ee>ef>df>d b->c#b->c#a>df>d e>ee>ef>df>d",
  "e>d>c#>db>d>c#b >c#agaegfe f O6 dc#dfdc#<b c#4" ]
```

Les notes sont représentées les lettres C, D, E, F, G, A et B de la notation anglo-saxone dont voici la correspondance académique C=Do, D=Ré, E=Mi, F=Fa, G=Sol, A=La, B=Si.

Les notes sont, par défaut, jouées comme des noires (quart de note) d'une durée de 500 ms la noire. Cela correspond à un tempo de 120 battements par minutes (aussi appelé *beat* en anglais).

Une note blanche (=2*noire) fait donc 1000 ms tandis qu'une ronde (=2*blanches =4*noires) fait donc 2000ms.

Le graphique ci-dessous reprend différentes durées de la note Sol (G) et leur notation anglo-saxone correspondante.



04RI60a – durée des notes et notation anglo-saxone.

Il est possible de modifié la durée d'une note en ajoutant un diviseur juste derrière la note. Ainsi C8 est 1/8 de la note Do. Donc 1/8 de ronde, ce qui correspond à une simple croche. En jouant une note encodée comme « C8 », la note est deux fois plus courte qu'une note encodée comme « C » qui est jouée comme une « C4 » par défaut (une noire).

La note spéciale R est utilisée pour produire des silences (pas de son).

Les espaces sont ignorés, ce qui permet de les utiliser pour améliorer la lisibilité (en séparant des blocs de notes).

•**A** – **G** : note à jouer.

•**R** : utilisé pour produire un silence pour la durée d'une note.

•*espace* : ignoré. Peut être utilisé comme outil de formatage.

•**>** : jouer la prochaine note un octave plus haut.

•**<** : jouer la prochaine note un octave en dessous.

•**+** ou **#** : après une note augmente cette derniere d'un demi-ton (dièse).

•**-** : après une note diminue cette dernière d'un demi-ton (bémol).

•**.** : un point après une note augmente la durée de celle-ci de 50%. Chaque point additionnel augmente le temps précédent de 50 % de plus. Ainsi « A.. » correspond a 1+0,5+0,25 soit une note La d'une durée de 1,75 fois la noire.

•**O** : suivit d'un chiffre permet de fixer l'octave (O4 par défaut).

•**T** : suivi d'un nombre, il permet de fixer le tempo en battement/min (T120 par défaut).

•**L** : suivi d'un nombre, il permet de modifier la durée par défaut des notes jouées. Utiliser 4 pour une noire, 8 pour une croche, 16 pour une double croche, etc. (L4 est la durée par défaut).

•**V** : suivi d'un nombre entre 0 et 15 permet de fixer le volume sonore (V15 par défaut).

•**MS** : Permet de jouer toutes les notes suivantes en *staccato* (piqué). Chaque note est jouée pendant la moitié du temps qui lui est alloué, l'autre moitié du temps sera occupé par un silence.

•**ML** : Permet de jouer toutes les notes suivantes en *legato* (liée). Chaque note est jouée pendant la totalité du temps alloué à la note. Permet d'inverser l'effet dy *staccato*, également la configuration par défaut.

•**!** : réinitialise l'octave, le tempo, la durée par défaut des notes, le volume et l'activation du staccato aux valeurs par défaut. Etant donné que ces paramètres subsistent entre deux appels successif de `play()` , c'est une façon commode de réinitialiser les paramètres musicaux permettant ainsi de scinder la musique en plus petites sections réutilisables.

•**1-2000** : suivant une note, ce nombre permet de déterminer la durée de cette note. Par exemple, C16 indique que la note C (La) est jouée en 1/16 de note, donc une double croche.

Propriété octave

Lors du rendu d'une mélodie à partir d'une chaîne de caractères, celui-ci est rendu à partir d'un octave par défaut (octave =4).

Les piezo buzzer ne disposant pas d'une large gamme dynamique, le rendu audio peut rarement dépasser l'octave et ce, a condition d'être ni trop haut ni trop bas en fréquence.

Passer d'une octave à l'autre modifie la fréquence de la note de musique jouée.

Propriété whole_note_duration

Défini la durée d'une note ronde, par défaut, celle-ci dure 2000 millisecondes. Toutes les notes de musique jouées (blanche, noire, croche, double croches, stacato, …) sont des fractions de cette durée.

Propriété volume

Permet de module, dans une certaine mesure, le volume sonore des notes jouées. Par défaut, cette valeur est fixée au maximum de 15.

Les éléments piezo ne disposent pas de contrôle de volumes il n'est donc pas possible de moduler l'amplification du son puisque le buzzer est exclusivement un élément vibrant à une fréquence donnée !

La meilleure façon de simuler une perception du changement de volume est de moduler la durée du cycle utile. Une volume à 15 (le maximum) est calé sur un cycle utile de 30 %

La version courante de la bibliothèque (0.0.3) ne module pas cycle utile pour simuler une modification de volume.

Méthode playFrequency(freq, dur, volume)

Fait osciller le piezo buzzer à une fréquence `freq` donnée durant une durée (`dur`) de dur millisecondes. Cela aura pour effert de produire un son.

Au terme de la durée mentionné, le piezo buzzer est désactivé.

L'appel à cette méthode avec une fréquence de 0 Hertz à également pour effet de désactiver le piezo buzzer.

Méthode playNote(note, dur, volume)

Permet de jouer une note parmi à l'aide des fonctions `SILENT_NOTE`, `NOTE_C(x)`, `NOTE_C_SHARP(x)`, `NOTE_D_FLAT(x)`, `NOTE_D(x)`, `NOTE_D_SHARP(x)`, `NOTE_E_FLAT(x)`, `NOTE_E(x)`, `NOTE_F(x)`, `NOTE_F_SHARP(x)`, `NOTE_G_FLAT(x)`, `NOTE_G(x)`, `NOTE_G_SHARP(x)`, `NOTE_A_FLAT(x)`, `NOTE_A(x)`, `NOTE_A_SHARP(x)`, `NOTE_B_FLAT(x)`, `NOTE_B(x)` où `x` est l'octave de la note.

C'est la notation anglo-saxonne qui est ici utilisé pour identifier les notes avec A=La, B=Si, C=Do, D=Ré, E=Mi, F=Fa, G=Sol. Le suffixe *sharp* fait référence à la dièse # tandis que le suffixe *flat* correspond à la bémol.

Ces fonctions sont déclarées dans la bibliothèque `zumobuzzer.py` .

Méthode isPlaying( )

Cette méthode permet de savoir si le buzzer produit une note ou s'il reste des notes à jouer pour achever une mélodie. Cela est utile lorsque d'une mélodie est jouée de façon asynchrone.

Méthode stopPlaying( )

Méthode complémentaire de `isPlaying()` permettant d'interrompre le rendu d'une mélodie asynchrone.

Méthode play(notes)

Permet de jouer une mélodie encodée dans une chaîne de caractères `notes` . Une fois la méthode appelée, celle-ci ne rend la main au code appelant que lorsque la mélodie est achevée.

Méthode reset( )

Réinitialise le paramétrage de la classe PololuBuzzer aux valeurs par défaut (octave, volume, staccato, …).

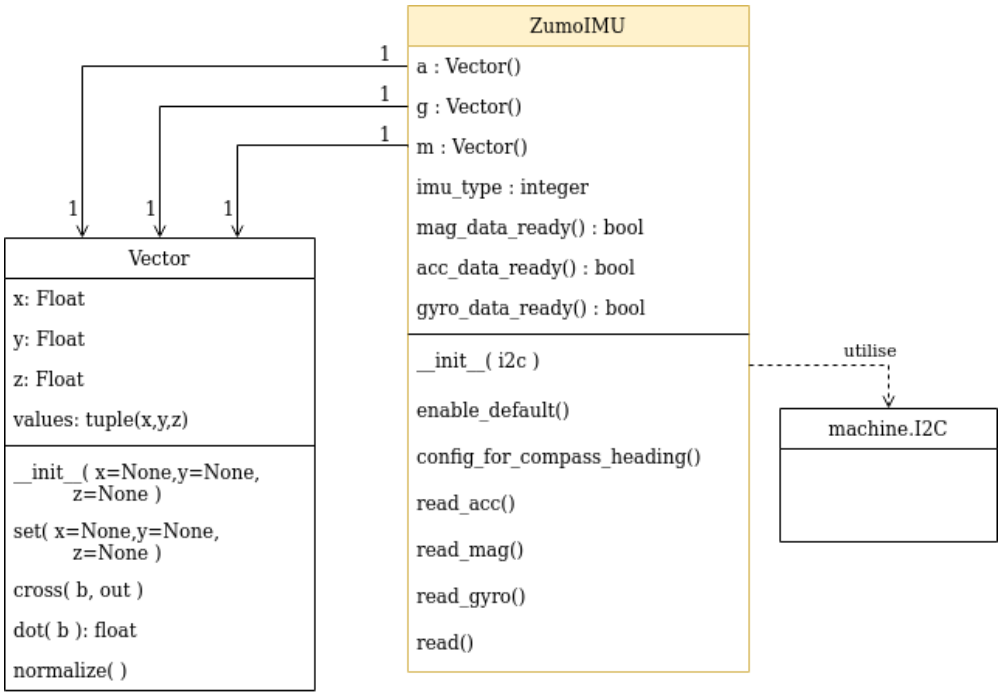
Méthode off( )

Permet de stopper le piezo buzzer.

6.Centrale inertielle

L'électronique de la centrale inertielle est susceptible de changer dans le temps. Ce fût par ailleurs le cas entre la version 1.1 et la version 1.2 du Robot Zumo.

Ainsi, la centrale inertielle ne fait pas partie du corps de la bibliothèque. Elle est disponible comme une bibliothèque annexe (`zumoimu.py`) disposant de ses propres classes utilitaires (`Vector` et `ZumoIMU`).



04RI56 – Diagramme des classe de la centrale inertielle

Communication entre ZumoIMU et ZumoShield

La seule contrainte est de permettre à la classe `ZumoIMU` de communiquer avec les composants électroniques de la centrale inertielle présente sur le Robot Zumo.

Pour rappel, les magnétomètre, accéléromètres et gyroscopes sont accessibles par l'intermédiaire d'un bus I2C.

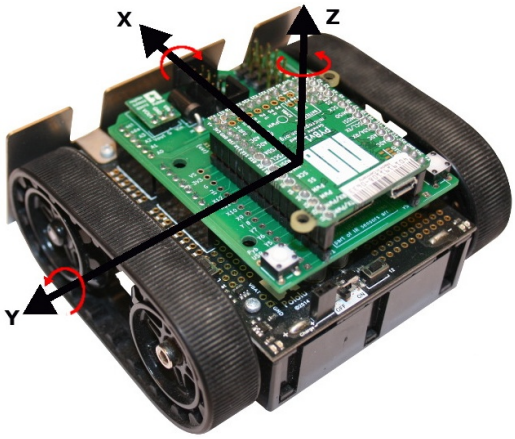
Bus I2C instancié par la classe `ZumoShield` est accessible par la propriété `i2c` . Cette classe connaît les détails électronique du Zumo Robot (donc la position du bus I2C sur le brochage), là où `ZumoIMU` est indépendante de ces détails techniques du Robot.

Ainsi l'instance du bus i2c à utiliser est communiqué en paramètre lors de la création de l'instance `ZumoIMU`.

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

6.1.Classe Vector

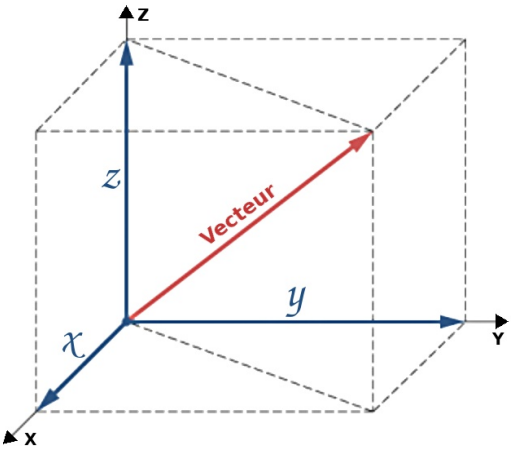
Avant d'aborder les détails concernant les vecteurs, il est important de rappeller la positions des axes sur le robot Zumo.



04RI61 – orientation des axes sur le Zumo

Les vecteurs sont utilisés pour symboliser une grandeur (et son orientation) dans l'espace. Les vecteurs s'utilisent dans les mathématiques, l'algèbre, la géométrie mais aussi pour symboliser des grandeurs physiques comme une force, une vitesse, une accélération, des champs (électriques, magnétiques), etc.

La classe `Vector` est utilisée pour créer des objets enregistrant les informations tridimensionnelles de vecteurs (selon les axes X, Y et Z).



04RI62 – Représentation d'un vecteur dans l'espace.

Les vecteurs s'utilisent aussi dans des opérations mathématiques (addition de vecteurs, produit scalaire, produit vectoriel de deux vecteurs et autres opérations), ainsi la classe `Vector` dispose de multiples méthodes ad-hoc.

La centrale inertielle permet de représenter 3 types de vecteurs distincts :

•Le champs magnétique (noté *m*) indique la direction du champs magnétique terrestre (le Nord magnétique) ou tout la présence de tout élément perturbant le champs magnétique (masse métallique, aimants disposés sur un parcours).

•La gravitation terrestre (noté *g*) peut aussi être représenté comme un vecteur. Ce vecteur d'accélération est un peu particulier puisqu'il est toujours dirigé vers le centre de la terre. Par conséquent l'orientation relatif des axes par rapport à ce vecteur permet d'avoir une idée de l'orientation du Zumo sur le terrain.

•L'accélération (noté *a*) est également représenté sous forme d'un vecteur. Le Zumo présente forcement un vecteur d'accélération pendant qu'il accélère (ou freine). Le Zumo présente aussi un vecteur d'accélération lorsqu'il reçoit un choc d'un autre robot car il est subitement accéléré dans une direction fortuite. Enfin, lorsque d'un Zumo rencontre un objet, celui-ci subit une forte décélération (ce qui est aussi un vecteur d'accélération mais dans le sens opposé).

Propriétés x, y et z

Les propriétés x, y, z contiennent des valeurs réelles (float) représentant les 3 composantes x, y et z d'un vecteur dans l'espace.

Ces valeurs sont initialisées à l'aide du constructeur `__init__()` .

Propriété values

Retourne les valeurs du vecteur sous forme d'un tuple de 3 éléments (x,y,z) . Cette représentation est très proche de la conception mathématique du vecteur.

Méthode __init_(x=None, y=None, z=None)

Lors de la création d'un vecteur, la méthode `__init__()` permet d'initialiser les 3 composantes x, y et z. Les 3 composantes doivent être initialisées afin d'obtenir un vecteur valide.

```
v = Vector( 1.3, 15, -0.5 )
```

Il est également possible d'initialiser l'instance avec tuple 3 positions contenant les valeurs respectives pour x, y et z. Cela permet d'initialiser un vecteur avec le résultat d'une opération vectorielle.

```
exemple = Vector( (1, 2, 5) )
v = Vector( 1.3, 15, -0.5 )
v2 = Vector( v.values ) # copie du vecteur
```

Méthode set(x=None, y=None, z=None)

Permet de modifier une ou plusieurs composantes d'un vecteur en précisant un ou plusieurs des paramètres x, y ou z (valeur réelle).

```
v = Vector( 1.3, 15, -0.5 )
...
v.set( y=1.1, z=0 )
```

Méthode cross(b, out)

Effectue la multiplication du vecteur avec le vecteur `b` (en paramètre) et stocke les résultats dans le vecteur `out` (aussi passé en paramètre).

```
v = Vector( 0, 0, 0 )
a = Vector( 4, 5, 6 )
b = Vector( 10, 10, 10 )
# v = a * b
a.cross( b, v )
# Afficher vecteur résultant
print( v.values )
```

Méthode dot(b)

Le produit scalaire de deux est représenté par un « . ». Cette opération retourne un nombre réel (`float`), donc un scalaire.

```
a = Vector( 4, 5, 6 )
b = Vector( 10, 10, 10 )
# val = a . b
val = a.dot( b )
# Afficher scalaire résultant
print( "%5.2f" % val  )
```

Méthode normalize( )

La normalisation consiste à transformer le vecteur vers un vecteur unitaire. Pour ce faire, les différentes composantes du vecteur sont divisées par la longueur du vecteur.

Les données x, y et z du vecteur sont mise-à-jour par le processus de normalization.

```
a = Vector( 4, 5, 6 )
a.normalize( )
# Afficher vecteur résultant
print( a.values )
```

6.2.ZumoIMU

La classe ZumoIMU permet d'acquérir les différentes données des composants constituant la centrale inertielle.

Pour rappel la communication avec les différents composants passe par un bus I2C (passé en paramètre lors de l'instanciation de ZumoIMU).

Propriété a

Cette propriété de type `Vector` contient les dernières informations connue du vecteur d'accélération (en m/s²), celui associé à une mise en mouvement ou un choc.

Ce vecteur reprend également la composante du vecteur d'accélération terrestre (*g*) dirigé vers le centre de la terre. Ce dernier normalement dirigé vers le bas (une valeur Z négative).

A noter que la composante du vecteur d'accélération terrestre est négligeable (en taille) par rapport à celui d'un vecteur d'accélération issu d'un choc.

Propriété g

Cette propriété de type `Vector` permet d'accéder aux dernières informations connues du gyroscope '. Ce vecteur permet de mesurer la rotation angulaire (degrés/sec) du robot durant ses différents mouvements volontaire ou non.

Propriété m

Cette propriété de type `Vector` permet d'accéder aux dernières informations connues du magnétomètre (en µT, microTesla). Avec un bon étalonnage, il est possible de localiser le nord magnétique de la terre à condition qu'il n'y a pas d'éléments perturbateurs comme des masses métalliques ou aimant super-puissant.

Selon l'élément perturbateur, le vecteur du champ magnétique terrestre sera mal positionné OU le vecteur présentera un vecteur de champs magnétique surpuissant.

Propriété imu_type

Le zumo robot n'est pas un produit récent. Il a donc sorti en différentes réversions au cours des années. La cause principale de ce changement de version sont les composants de la centrale inertielle qui, arrivés en fin de vie, sont remplacés par une révision plus récente.

Cette propriété retourne un entier correspondant aux constantes :

- `IMU_TYPE_Unknown` : la centrale inertielle n'est pas identifiée (valeur 0).

- `IMU_TYPE_LSM303DLHC` : la détection de l'IMU à identifié le composant LSM303DLHC comme accéleromètre + magnétomètre (valeur 1)
- `IMU_TYPE_LSM303D_L3GD20H` : la détection de l'IMU à identifier le composant LSM303D comme accéléromètre + magnétomètre et le composant L3GD20H comme gyroscope (valeur 2)
- `IMU_TYPE_LSM6DS33_LIS3MDL` : la détection de l'IMU à identifié le composant LSM6DS33 comme gyroscope + accéléromètre et le composant LIS3MDL comme magnétomètre (valeur 3)

La version 1.3 du Zumo Robot pour Arduino développé dans cet ouvrage utilise le composant LSM6DS33 comme gyroscope+accéléromètre et le composant LIS3MDL comme magnétomètre. Si la propriété `imu_type` retourne donc la valeur 3 (`IMU_TYPE_LSM6DS33_LIS3MDL`).

Si la valeur retournée est différente de 3 (`IMU_TYPE_LSM6DS33_LIS3MDL`) cela signifie que la révision du robot zumo est antérieure à 1.3 . La bibliothèque supporte les anciennes révisions du robot sans pour autant avoir été testée dans pareil cas.

La détection de la centrale inertielle prend place lors de la création de l'objet. Si cette opération échoue alors une exception avec le message « *IMU detection failed* ». Dans pareil cas, le robot zumo dispose certainement d'une centrale inertielle plus récente que le présent ouvrage.

Il est possible de trouver les fiches technique (datasheet) de ces composants en saisissant la référence du composant sur un site de recherche tel que Google.

Propriété mag_data_ready

Retourne un booléen indiquant si la centrale inertielle dispose de nouvelles données magnétiques.

Propriété acc_data_ready

Retourne un booléen indiquant si la centrale inertielle dispose de nouvelles données de l'accéléromètre.

Propriété gyro_data_ready

Retourne un booléen indiquant si la centrale inertielle dispose de nouvelles données du gyroscope.

Méthode __init_(i2c)

Cette méthode est le constructeur de la classe ZumoIMU. Elle initialise tous les paramètres de l'objet, détecte le type de centrale inertielle, ce qui initialise la propriété imu_type et initialise les paramètres par défaut de la centrale inertielle (voir `enable_default()`).

Si la détection échoue, l'exception « *IMU detection failed !* » provoquera l'arrêt du script utilisateur.

Méthode enable_default( )

Cette méthode établit le paramètrage par défaut des différents composants de la centrale inertielle présente sur le Robot Zumo.

L'**accéléromètre** est configuré en mode haute performance avec un échantillonnage 52 Hz (soit 52 fois par seconde). L'échelle de mesure est fixée de -2g à +2g, ce qui permet aussi de capturer le vecteur d'accélération terrestre (ce qui ne serait plus possible sur une échelle +/- 16g).

Le **magnétomètre** est configuré en mode ultra-haute performance sur les axes X et Y. L'échantillonnage est fixé à 10 Hertz sur une échelle de -4 à +4 Gaus (-400 µT à +400µT). Avec un mode de conversion continu (un nouvel échantillonnage est démarré automatiquement à la fin du précédent).

A noter que le mode ultra-haute performance est également activé sur l'axe Z. Cependant, les informations seront fort peu utile étant donné la masse métallique importante représentée par le piles.

Le gyroscope est également configuré en mode haute performance avec un échantillonnage à 208 Hertz. La gamme de mesure permet de relever une gamme de valeur de -245 à +245 dps (degré par seconde).

Méthode config_for_compass_heading( )

Cette méthode optimise la centrale inertielle pour la détection du Nord magnétique.

Le mode ultra-haute performance est configuré sur les axes X et Y et l'échantillonnage élevé à 80 Hertz.

Méthode read( )

Méthode générale effectuant une lecture des 3 capteurs magnétomètre, accéléromètre et gyroscope.

Si la capture des données est complète, elle est aussi plus lente puisque les 3 capteurs sont interrogés à tour de rôle sur le bus I2C dont on sait que la bande passante est limitée.

Dans le cas de détection du Nord magnétique, il sera préférable d'interroger exclusivement le magnétomètre de sorte à obtenir un échantillon de donnée le plus grand possible.

Méthode read_acc( )

Acquisition des données depuis l'accéléromètre. Celle-cis sont disponibles dans le vecteur `a` (voir la propriété correspondante).

Méthode read_mag( )

Acquisition des données depuis le magnétomètre. Celle-cis sont disponibles dans le vecteur `m` (voir la propriété correspondante).

Méthode read_gyro( )

Acquisition des données du gyroscope. Celle-cis sont disponibles dans le vecteur `g` (voir la propriété correspondante).