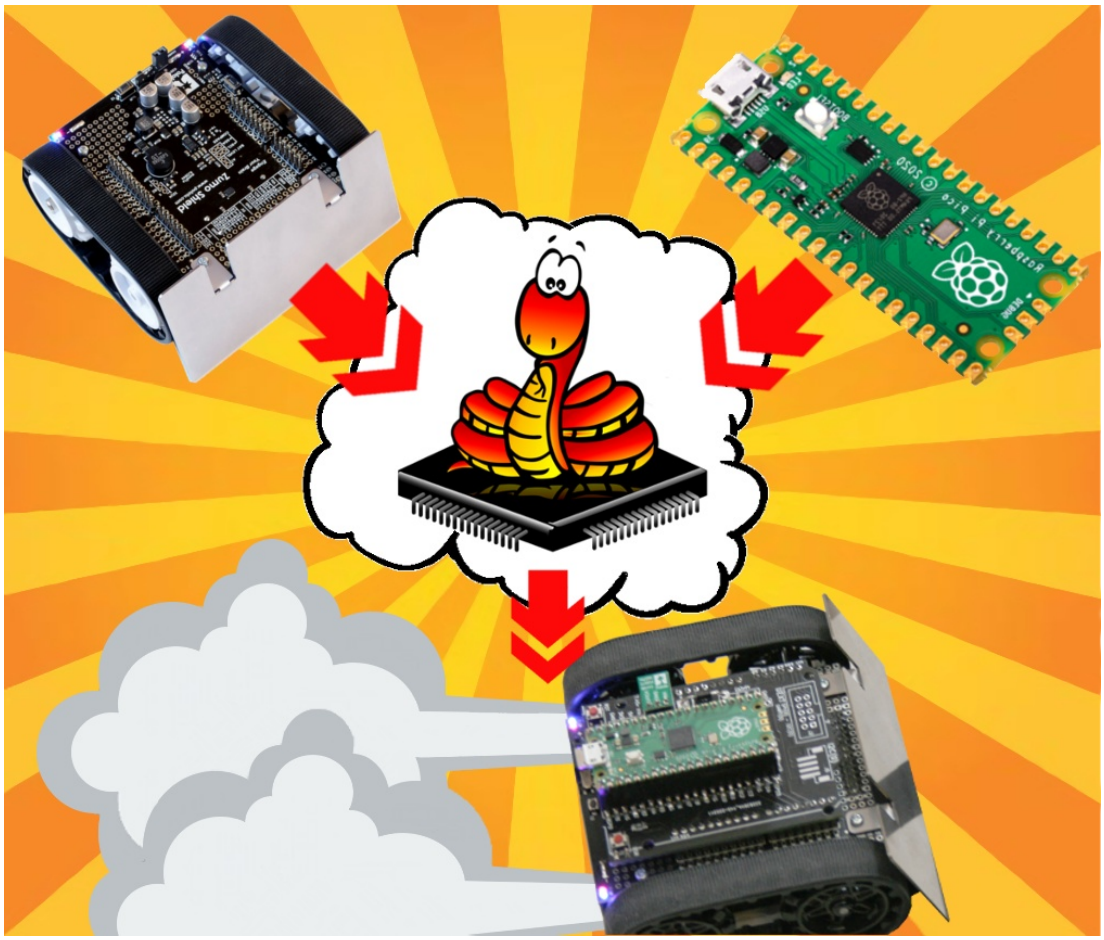
Pico, Zumo Robot et MicroPython

Programmer le Zumo Robot avec Python pour Microcontrôleur



Exemples avancés

Table des matières

1.Commande RC

Créer des robot ou objet intelligent et pouvoir les commander à distance à l'aide d'une télécommande est un rêve que tout maker aura au moins fait une fois.

Dans le monde du modélisme, on utilise un

1.1.Emeteur-Récepteur RC

Dans le monde du modélisme, le système de commande RC (Emeteur/Recepteur Radio Command) est disponible indépendamment du système contrôlé.

Que cela soit en Belgique ou en France, nombre de sites vendent des commandes RC (émetteur + récepteur inclus) indépendamment du véhicule à commander.

En Belgique, je vous recommande la société MCMGroup (www.mcmgroup.be) qui dispose d'un personnel compétent pouvant apporter leur aide au néophyte que j'étais.

Quelques critères sont à garder à l'oeil :

1.Le prix

2.Certification CE

3.Émetteur avec Récepteur.

4.Le mode de la télécommande

5.Le nombre de canaux

1.1.1.Le prix

Il faut garder à l'esprit qu'un matériel réutilisable et durable implique inévitablement une conception soignée, technologie numérique et fabrication de qualité.

Oubliez les télécommandes des jouets bons marchés. Ces télécommandes ne sont pas conçues dans l'idée du réemploi, ni d'une utilisation durable. Utilisant généralement une technologique analogique, tenter de hacker/réutiliser un tel matériel peu s'avérer source d'une grande frustration.

Il faudra compter une 75 à 150 Euro pour une télécommande de qualité honorable proposant plus de deux canaux (très utile pour des makers) et la possibilité de configurer les canaux.

A défaut, les sites en « seconde main » permettent parfois de faire des achats très intéressants dans ce domaine.

1.1.2.Certification CE

Il est tentant de faire un achat sur des grands sites de distribution comme Ama… , AliB… ou A..Express. La plupart de ces appareil ne disposent pas d'une certification CE et ne peuvent donc pas être utilisés sur le territoire.

Outre la liberté qu'un utilisateur pourrait prendre -a ses risques et péril- avec la réglementation Radio-Fréquence, le marquage CE est aussi le signe d'une conception aboutie et fabrication de qualité.

1.1.3.Émetteur avec récepteur

Il est bien loin le temps où il était nécessaire d'utiliser un cristal identique entre la télécommande et le récepteur.

De nos jours, les télécommandes fonctionnent en mode numérique avec des protocoles de corrections, algorithme de chiffrement.

Il est donc essentiel de faire l'**achat conjoint de l'émetteur + récepteur** ! En effet, ces deux éléments sont pré-appairés en usine. L'achat conjoint permet d'éviter de nombreuses difficultés inutiles.

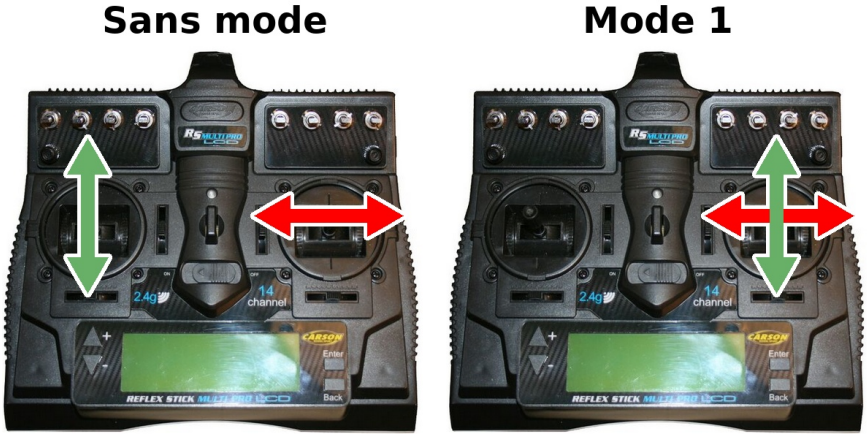
A noter que les modèles les plus coûteux permettent de réaliser un appairage avec un module récepteur de la même marque à proximité.

1.1.4.Le mode de la télécommande

Les modes de 1 à 4 concernent généralement les hélicoptères, avions, drones. Le mode associe la commande des deux joysticks à des fonctionnalités spécifiques (suivant le mode).

Les mode 1 ou « sans mode » concerne principalement véhicules roulants et bateaux.

Il est assez commun de commander les gaz (*throttle*) pour avancer/reculer avec un joystick et utiliser l'autre joystick modifier la direction (*steering*) gauche/droite.



08RI01 – Mode de commande

Les télécommandes à double joystick proposent généralement de nombreux canaux facilement accessible du bout des doigts.

Les télécommandes à volant (sans mode) sont aussi très populaire dans le monde automobile.

08RI02 – Commande à volant

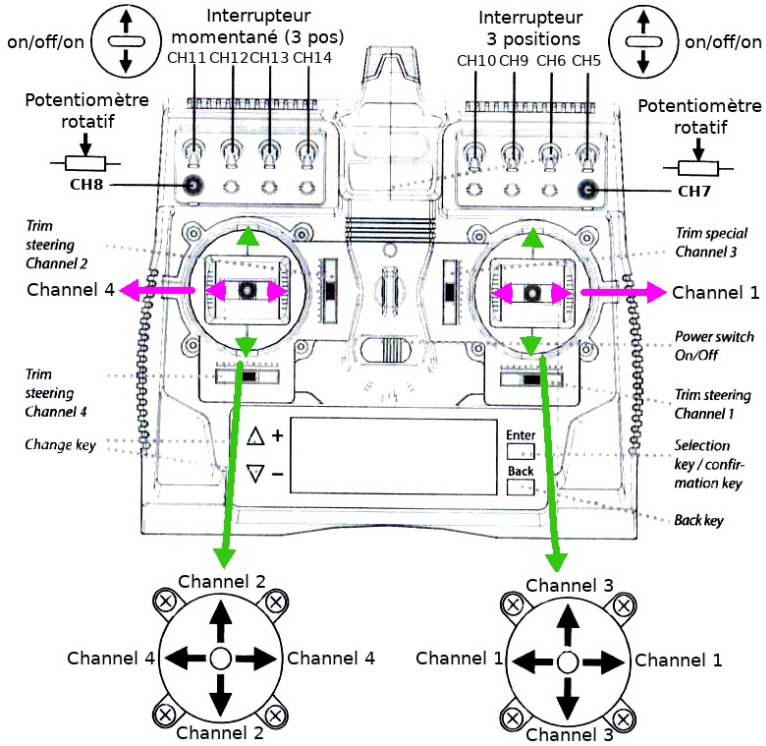
1.1.5.Le nombre de canaux

Nombre de télécommandes du mode RC disposent de multiples canaux. Ceux-ci sont identifiés par leur numéro de canal (de 1 à N)

Etant donné que la télécommande sera utilisée pour piloter un véhicule, deux canaux seront nécessaires.

Un premier canal pour commander commander la direction et un deuxième canal pour commander les gaz. A noter qu'il est possible d'opter pour d'autres mode de contrôle nettement plus originaux.

La commande Carson présentée dans ce chapitre était destinée à la commande d'un semi-remorque. Cette télécommande dispose de pas moins de 14 canaux.



08RI03 – Telecommande Carson à 14 Canaux

Dont voici le module récepteur présentant :

- un connecteur d'alimentation pour alimenter le récepteur
- les 14 connecteurs des 14 canaux.

08RI04 – Module récepteur RC

Les broches des canaux ressemblent à s'y méprendre aux connecteurs utilisés pour les servo-moteurs.

C'est normal puisque le signal utilisé sur ces connecteurs est techniquement identique à celui qui anime un servo-moteur.

08RI05 – Exemple de servo-moteur

Le +5V est au centre, la masse a gauche (brun) et signal à l'opposé (orange). L'intérêt d'une telle connectique est quelle évite la destruction des composant si elle était branchée à l'envers (le +5V est toujours au centre).

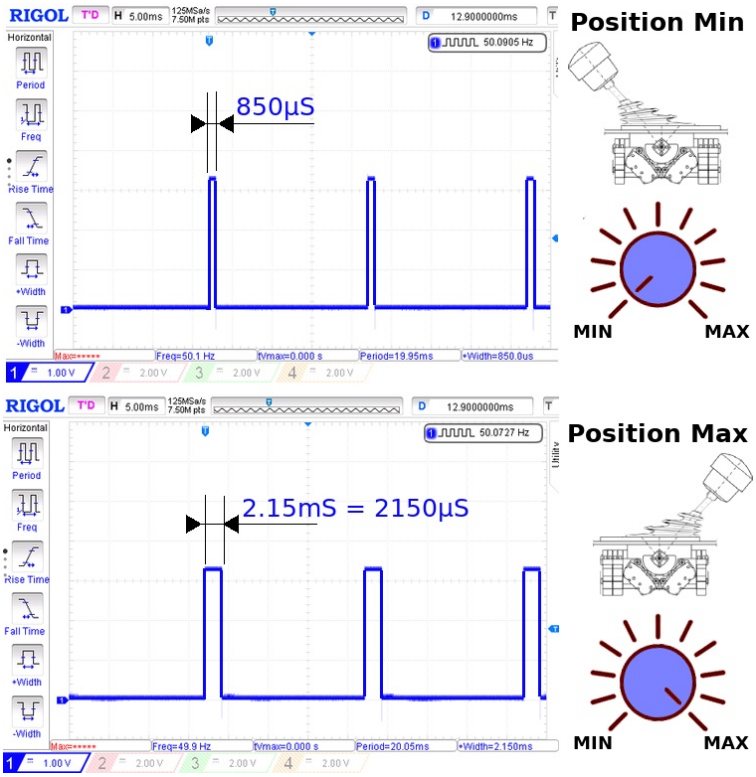
Un signal servo peut être utilisé pour commander de bien d'autres types de périphériques (vitesse moteur, direction, feux, relais, etc).

1.2.Les signaux RC

Si l'on alimente le module récepteur, un oscilloscope permet d'inspecter le signal réceptionné par le canal 7 (bouton rotatif droit de la télécommande).

08RI06 – Mesure du signal réceptionné.

Ce signal, produit sur le connecteur 3 points (broche signal, en haut), est répété toutes les 20ms. Le temps d'impulsion du signal représente l'amplitude du signal de commande (entre un minima et un maxima).



08RI07 – Signal RC entre les positions min et max.

Ce signal s'apparente à une valeur analogique qui, elle aussi, évolue entre un minimum et un maximum. Ce type de signal permet donc de contrôler la vitesse d'un moteur, la direction du véhicule, la puissance des gaz, mes ailerons d'un avion, etc.

La nature même du signal n'empêche nullement d'envoyer un état binaire (marche/ Arrêt). Dans pareil cas, la valeur « arrêt » serait un signal à ~800µS (donc largement sous la médiane de 1050µS) tandis qu'un signal « marche » serait à ~2000µS.

1.2.1.Fonction time_pulse_us()

MicroPython dispose de la fonction `time_pulse_us()` dans sa bibliothèque `machine`.

Cette fonction permet de calculer la durée, en microsecondes, d'une pulsation sur une broche.

```
from machine import time_pulse_in
usec = time_pulse_us( pin(15), pulse_level=1 )
```

Avec le paramètre `pulse_level=1`, la fonction attend que le signal passe au niveau haut puis mesure la durée jusqu'à ce que le signal repasse au niveau bas.

`time_pulse_us()` est une fonction bloquante ! Etant donné que le signal RC est généré toutes les 20ms, le temps de blocage maximum est de l'ordre de 20ms par mesure (donc aussi chaque canal mesuré).

1.3.Contrôle Renault FT-14

1.3.1.Le char Renault FT-14

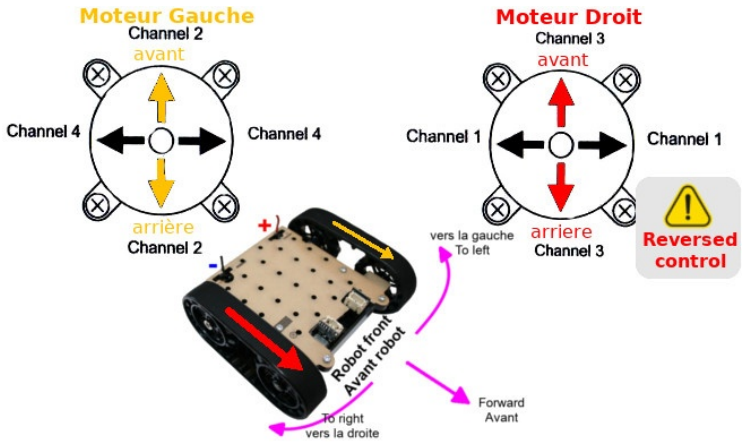
Le char Renault FT17 fût produit durant les années 1917-1918 et était capable de transporter deux personnes (le conducteur et le mitrailleur). Ces chars se pilotaient non pas à l'aide d'un volant, ni d'un manche mais bien avec deux manches.

Le conducteur avait un manche dans chaque main. Le manche droit contrôlant la chenille droite et le manche gauche contrôlant la chenille gauche.

Ce type de commande n'est pas des plus naturelles.

1.3.2.Commande Radio

Ce projet se propose de suivre le même mode de commande pour le robot Zumo.



08RI08 – Commande de type char

En utilisant les canaux 2 et 3 (spécifique à ce modèle de télécommande), il est possible d'avoir une commande similaire au char.

- La commande à droite commande la chenille droite.
- La commande à gauche commande la chenille gauche.

Il est important de rappeler qu'avoir les deux chenilles tournant dans le même sens, il faut que les deux moteurs tournent dans les sens opposés !

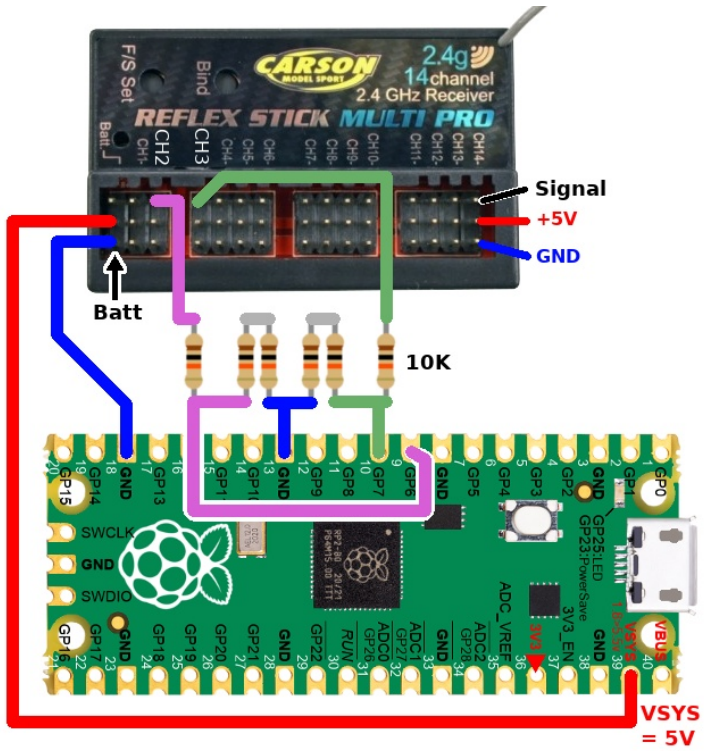
1.3.3.Brancher sur le Pico

En plus des branchements déjà réalisés entre le Robot Zumo et le Pico (cf. Brancher), le module récepteur peut être branché comme suit sur les broches encore libre du Pico.

Le canal 3 (chenille droite) est branché sur le GP6 par l'intermédiaire d'un pont diviseur de tension pour réduire la tension de 5V (du récepteur) vers 3,3V (du Pico).

Le canal 2 (chenille gauche) est branché sur le GP7 par l'intermédiaire d'un pont autre pont diviseur de tension.

Les ponts diviseurs de tension sont réalisé à l'aide de résistances de 10 KOhms.

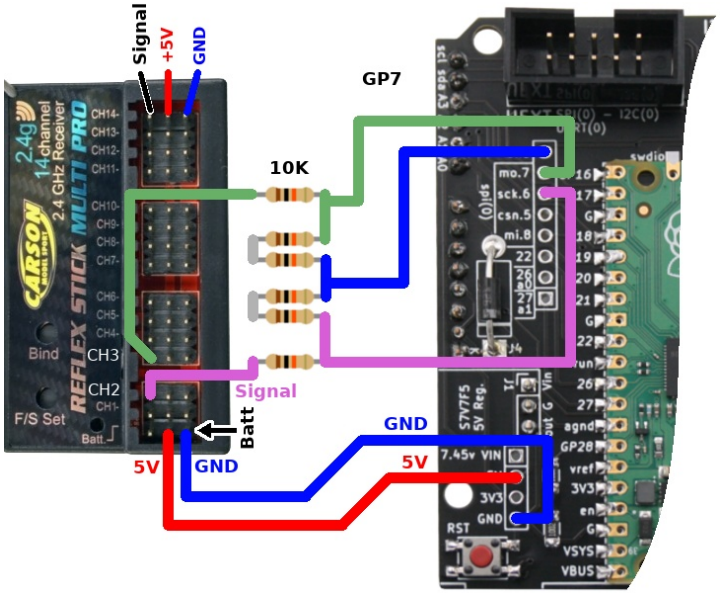


08RI10 – Brancher un récepteur radio sur le Pico

La tension sur la broche VSYS est de 5V, cette tension est générée par le régulateur 5V de Pololu (cf. Brancher – Raccordements).

1.3.4.Brancher sur l'adaptateur Pico-Zumo

L'adaptateur Pico-Zumo expose également les broches encore libre du Pico sur un connecteur d'extension. Les broches GP7 & GP6 en font également partie.



08RI11 – Brancher le récepteur radio sur l'adaptateur Pico-Zumo

1.3.5.Tester le récepteur radio

Le script `rctest.py` exploite le relevé d'impulsion sur les canaux 2 et 3 de la télécommande pour commander le Zumo Robot comme un tank.

Ce script est disponible dans le dépôt du projet :

https://github.com/mchobby/micropython-zumo-robot/blob/main/extras/rctest.py

L'exécution de `rctest.py` permet de relever la valeur de repos, valeur maximale et minimale de l'impulsion.

```
01: from zumoshield import *
02: from machine import Pin, time_pulse_us
03: import time
04:
05: ch_left  = Pin( 7, Pin.IN )
06: ch_right = Pin( 6, Pin.IN )
07:
08: z = ZumoShield()
09: print( "Left (uS)", "Right (uS)")
10: while True:
11:   us_left  = time_pulse_us( ch_left, 1 ) # pulse_level=1
12:   us_right = time_pulse_us( ch_right, 1 )
13:   print( us_left, us_right )
14:   time.sleep(1)
```

Voici les détails du fonctionnement du script :

•Lignes 1 à 3 : import des classes et bibliothèques nécessaires

•Ligne 5 : déclaration de la broche GP7 associée au moteur gauche. Ce moteur est contrôlé par le canal n° 2 du récepteur radio.

•Ligne 6 : déclaration de la broche GP6 associée au moteur droit. Ce moteur est contrôlé par le canal n° 3 du récepteur radio.

•Ligne 8 : création de l'objet `ZumoShield` (variable `z`). Pas indispensable mais assure que les moteurs soient mis à l'arrêt !

•Ligne 10 : début de la boucle infinie. Pour arrêter le script, il sera nécessaire de presser la combinaison de touche [Ctrl] + c.

•Ligne 11 : récupération de la longueur d'impulsion (en µSecondes) pour le moteur gauche. Le second paramètre indique que la fonction doit attendre le passage au niveau haut puis mesurer la durée de celui-ci (temps pour qu'il repasse au niveau bas). La fonction `time_pulse_us()` retourne une valeur négative si la mesure est impossible.

•Ligne 12 : longueur d'impulsion pour le moteur droit.

•Ligne 13 : affichage des deux informations dans la session REPL.

•Ligne 14 : mettre le script en pause pendant une seconde avant d'effectuer la mesure suivante.

Ce qui produit les résultats suivant dans la session REPL :

```
1042 1485
1042 1484
1941 1484
1941 1484
1491 1485
1491 1484
```

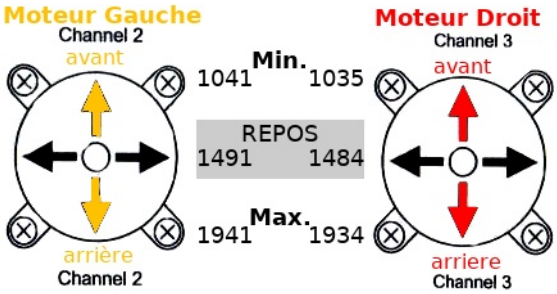
La colonne à gauche (chenille gauche) présente le temps d'impulsion en micro-secondes pour le canal radio n° 2 (broche GP7).

La valeur 1042 correspond à la valeur minimale lorsque le levier est poussé vers le haut tandis que 1491 correspond à la valeur au repos (centrale).

La colonne de droite (chenille droite) correspond au canal radio n°3 (broche GP6).

Enfin, la fonction time_pulse_us() retourne une valeur négative lorsqu'il n'y pas d'impulsion détectée (au terme d'un timeout de 1000000µSec = 1 sec).

Le test permet de faire les relevés suivants.



08RI12 – relevé des minimas, repos et maximas

Il est intéressant de noter que la valeur de REPOS ±400 permet d'obtenir une valeur très proche des minimas et maximas (dans les deux cas). Cette caractéristique sera exploitée pour faciliter l'implémentation du script de pilotage.

1.3.6.Script de pilotage rctank

Le script `rctank.py` affiche le relevé d'impulsion sur les canaux 2 et 3 de la télécommande.

Le script est disponible dans le dépôt du projet :

https://github.com/mchobby/micropython-zumo-robot/blob/main/extras/rctank.py

```
01: from zumoshield import *
02: from machine import Pin, time_pulse_us
03: import time
04:
05: LEFT_CENTER_US = 1491
06: RIGHT_CENTER_US = 1484
07: RANGE_US = 400
08:
09: ch_left  = Pin( 7, Pin.IN )
10: ch_right = Pin( 6, Pin.IN )
11:
12: def map(value, istart, istop, ostart, ostop):
13:   return ostart + (ostop - ostart) * (
               (value - istart) / (istop – istart) )
14:
15: z = ZumoShield()
16: z.play_blip()
17: while True:
18:   us_left  = time_pulse_us( ch_left, 1 )
19:   us_right = time_pulse_us( ch_right, 1 )
20:
21:   if (us_left < 0) or (us_right < 0):
22:     z.motors.stop()
23:     z.led.toggle()
24:     # no sleep required
25:     continue
26:
27:   speed_left = map( us_left, LEFT_CENTER_US-RANGE_US,
                       LEFT_CENTER_US+RANGE_US, -400, +400 )
28:   speed_right = map( us_right, RIGHT_CENTER_US-RANGE_US,
                        RIGHT_CENTER_US+RANGE_US, -400, +400 )
29:   speed_right = -1*speed_right
30:
31:   if abs(speed_left)<=25:
32:     speed_left = 0
33:   if abs(speed_right)<=25:
24:     speed_right = 0
35:   z.motors.setSpeeds( speed_left, speed_right )
36:   time.sleep_ms(100)
```

Voici les détails de fonctionnement du script :

•Lignes 1 à 3 : import des classes, fonctions et bibliothèques nécessaires.

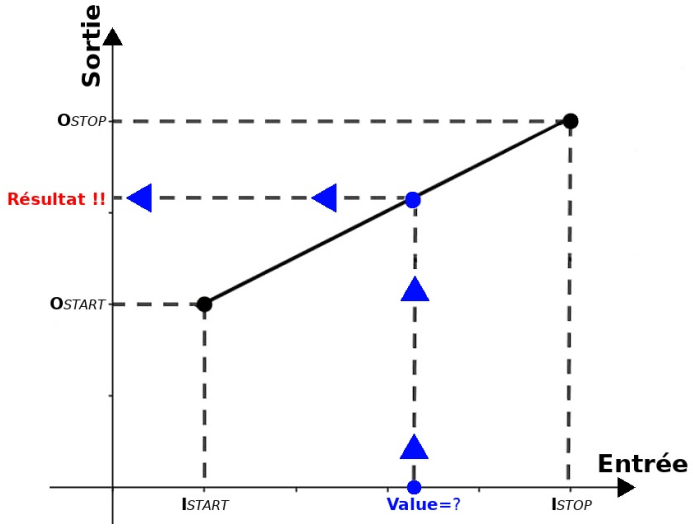
•Lignes 5 et 6 : temps d'impulsion au repos (en µSec) pour le contrôle de la chenille gauche (*Left* en anglais, canal 2) et pour la chenille droite (*right* en anglais, canal 3).

•Ligne 7 : écart max (en µSec) pour chaque canal à partir de la position de repos (ajouté ou soustrait).

•Ligne 9 : définition de la variable ch_left correspondant au canal 2 du récepteur radio (chenille gauche). Cette instance de la classe Pin est associé au GPIO 7, broche définie en entrée pour la circonstance.

•Ligne 10 : définition de la variable ch_right destiné à la chenille droite (canal 3 vers GPIO 6)

•Lignes 12 et 13 : définition de la fonction `map()` dont le fonctionnement est identique à la fonction du même nom pour Arduino. La fonction `map()` effectue une interpolation linéaire permettant de transposer une valeur d'entrée `value` de la gamme d'entrée `istart` à `istop` vers une nouvelle gamme de valeur de sortie `ostart` à `ostop`.



08RI14 – Interpolation linéaire de map()

•Ligne 14 : création de l'instance du `ZumoShield` (variable `z`), ce qui initialise le contrôleur moteur.

•Ligne 15 : production d'un signal sonore pour indiquer le démarrage effectif du contrôle radio.

•Ligne 17 : démarrage de la boucle infinie exécutant l'acquisition des temps d'impulsions des canaux 2 et 3 (GP7 et GP6), conversion linéaire en vitesse Zumo (entre -400 et +400) et, enfin, une petite pause.

•Lignes 18 et 19 : capture du temps d'impulsion `us_left` (us=microseconde) de la commande de la chenille gauche et `us_right` pour la chenille droite. Si le récepteur ne reçoit pas de signal, le *timeout* de la fonction `time_pulse_us()` est de 1 seconde.

•Lignes 21 à 25 : si l'un des canaux retourne une valeur négative alors cela signifie que la connexion avec la télécommande est perdue. Dans pareil cas, les moteurs du robot sont stoppés et la LED utilisateur du Zumo change d'état (`toggle`). L'instruction continue redémarre immédiatement la boucle `while` pour une nouvelle tentative de capture des impulsions.

La LED utilisateur clignote lorsque la connexion radio est perdue. La vitesse de clignotement est limitée par le timeout (temps de réponse) de la fonction pulse_time_us(). De fait, la LED clignote une fois toutes les 2 secondes.

•Ligne 27 : conversion du temps d'impulsion de la chenille gauche (`us_left`) de la gamme 1491-400 à 1491+400 vers une vitesse moteur Zumo (donc entre -400 et +400 pour `speed_left`).

•Ligne 28 : opération identique pour le moteur droit. La vitesse `speed_right` est donc également situé entre -400 et +400.

•Ligne 29 : le sens de rotation du moteur droit doit être inversé pour que le robot avance lorsque les deux commandes sont poussées vers l'avant. Pour inverser le sens de rotation, il suffit de multiplier la vitesse par -1.

•Ligne 31 et 33 : lorsque les commandes sont au point de repos, les vitesses moteurs sont supposées être égale à zéro. Cependant, suite au différents calculs, il est fort probable que ces valeurs de vitesses soient très proche de zéro mais pas exactement égales à zéro. D'autre part une vitesse moteur très faible ne permet pas au Zumo de se déplacer. Par conséquent, si une vitesse moteur est inférieure à 25

(sur une gamme de 0 à 400) alors le script force la vitesse à 0 (l'arrêt). La fonction `abs()` permet de considérer la vitesse en valeur absolue (donc sans le signe, en effet `abs(-12)` retourne 12, `abs(15)` retourne 15).

•Ligne 35 : applique les vitesses calculée aux moteurs du Zumo.

•Ligne 36 : lorsqu'il n'y a pas de *timeout* dans les appels de pulse_time_us(), le traitement du corps de la boucle est quasi instantané. Une pause de 1/10 sec est alors ajouté dans la boucle pour ralentir la boucle de traitement. Même dans ces conditions, la vitesse des moteurs est revue une dizaine de fois par seconde, ce qui amplement suffisant pour maintenir une bonne réactivité.

1.4.Contrôle joystick analogique

Le contrôle de type « char » montre très vite les limites d'une telle approche. S'il est confortable de contrôler le Zumo lors qu'il est dos au pilote, il en va tout autrement s'il se présente de face (ou de biais) au pilote.

La gymnastique intellectuelle rend le contrôle difficile et hasardeux.

1.4.1.Commande joystick RC

Cette extension du projet se propose d'utiliser un seul joystick pour piloter le robot Zumo.

En effet, le joystick peu prendre n'importe quelle position dans le cercle permettant ainsi de contrôler la vitesse (avant/arrière) et en même temps que la direction a suivre (déporter le manche vers la droite ou la gauche).

Cette approche est beaucoup plus naturelle et facile à maîtriser.

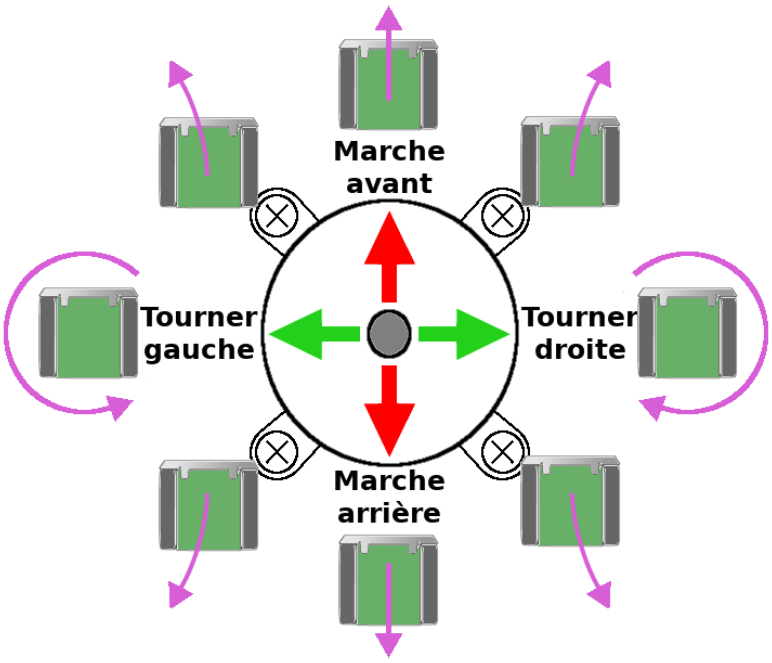
1.4.2.Principe de fonctionnement

Il n'est plus question ici de commander les deux moteurs directement avec des joysticks mais de moduler le régime de chacun des moteurs en fonction de la position du joystick.

Le schéma ci-dessous présente les différent cas de figure en fonction de la position du joystick :

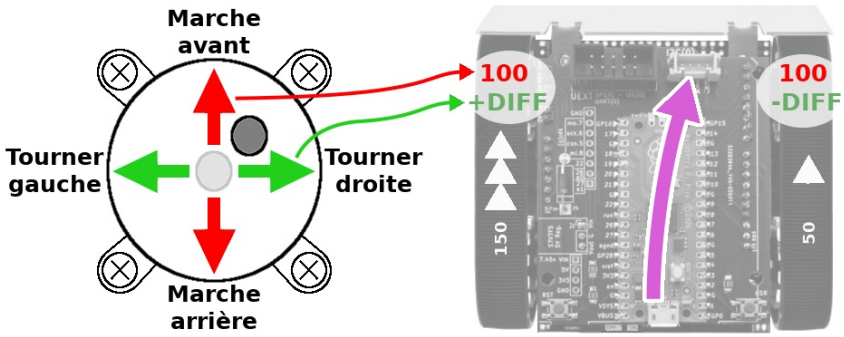
•Le canal 3 contrôle toujours la vitesse.

•Le canal 1 contrôle la direction à suivre (tout droit, à droite, à gauche)



08RI15 – contrôle du Zumo avec un joystick unique

Le principe de fonctionne est très simple, la composante direction (droite/gauche) est utilisée pour créer une différence de vitesse (positive/négative) entre les moteurs.

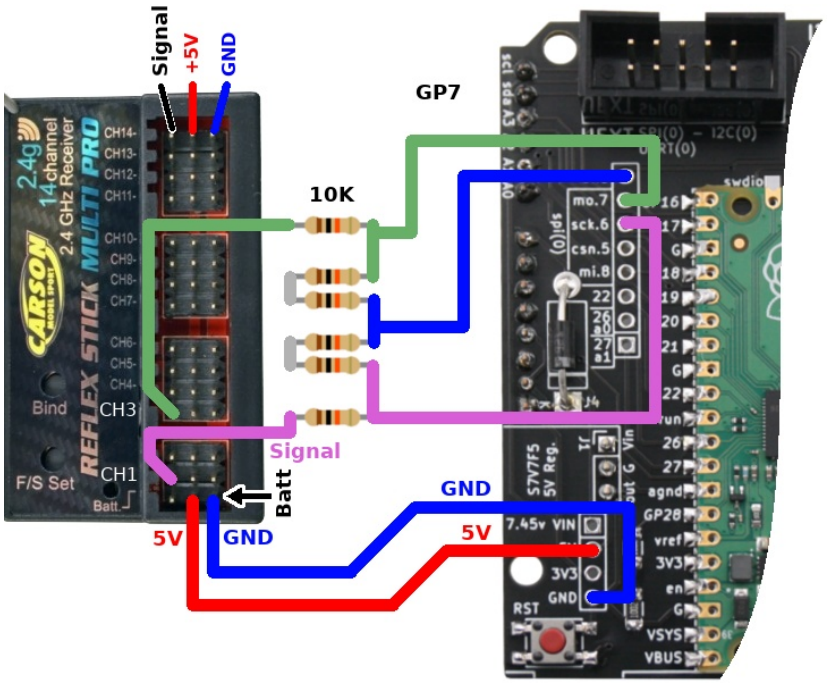


08RI16 – détail du pilotage moteur

1.4.3.Brancher sur l'adaptateur Pico-Zumo

Comme pour la précédente implémentation, le canal 3 reste branché sur la broche GP7.

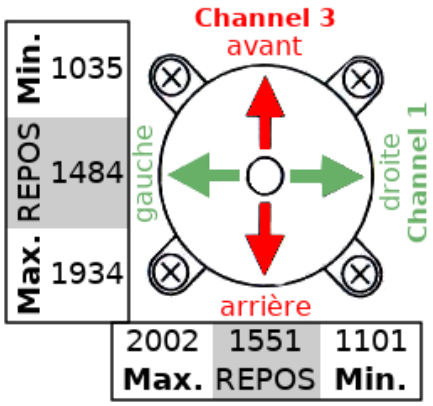
Par contre, cette fois, c'est le canal 1 qui est branché sur la broche GP6 (et non plus le canal 2).



08RI18 – Brancher les canaux 3 et 1 pour un contrôle joystick

En réutilisant une nouvelle fois le script `rctest.py`, il est possible de relever la position de repos ainsi que minima et maxima sur le canal 1 (canal de direction) nouvellement branché.

Voici les résultats obtenus :



08RI19 - relevé des minimas, repos et maximas

Comme pour la commande de type « tank » les minimas et maximas restent dans une gamme de ± 400 µS.

1.4.4.Script de pilotage rcjoy

Le script `rcjoy.py` affiche le relevé d'impulsion sur les canaux 1 et 3 de la télécommande pour piloter la robot à l'aide du joystick.

Le script est disponible dans le dépôt du projet :

https://github.com/mchobby/micropython-zumo-robot/blob/main/extras/rcjoy.py

```
01: from zumoshield import *
02: from machine import Pin, time_pulse_us
03: import time
04:
05: SPEED_CENTER_US = 1489 # ch 3
```

```
06: DIR_CENTER_US = 1551   # ch 1
07: RANGE_US = 400
08:
09: ch_speed  = Pin( 7, Pin.IN ) # ch 3
10: ch_dir = Pin( 6, Pin.IN )    # ch 1
11:
12: def map(value, istart, istop, ostart, ostop):
13:   return ostart + (ostop - ostart) * ((value - istart) /
(istop - istart))
14:
15: z = ZumoShield()
16: z.play_blip()
17: while True:
18:   us_speed  = time_pulse_us( ch_speed, 1 )
19:   us_dir = time_pulse_us( ch_dir, 1 )
20:
21:   if (us_speed < 0) or (us_dir < 0):
22:     z.motors.stop()
23:     z.led.toggle()
24:     continue
25:
26:   speed = map( us_speed, SPEED_CENTER_US-RANGE_US,
SPEED_CENTER_US+RANGE_US, +300, -300 )
27:   dir_diff = map( us_dir, DIR_CENTER_US+RANGE_US,
DIR_CENTER_US-RANGE_US, -100, +100 )
28:
29:   if abs(speed)<=25:
30:     speed = 0
31:   if abs(dir_diff)<=10:
32:     dir_diff = 0
33:   if speed<=-25:
34:     dir_diff *= -1
35:   elif speed==0:
36:      dir_diff *= 2
37:   speed_left = int(speed + dir_diff)
38:   speed_right = int(speed - dir_diff)
39:   if speed_left<-400:
40:     speed_left = -400
41:   if speed_left>400:
42:     speed_left = 400
43:   if speed_right<-400:
44:     speed_right=-400
45:   if speed_right>400:
46:     speed_right=400
47:
48:   z.motors.setSpeeds( speed_left, speed_right )
49:   time.sleep_ms(100)
```

Voici les détails de fonctionnement du script :

•Ligne 1 à 3 : import des classes, fonctions et bibliothèques nécessaires.

•Ligne 5 : La constante `SPEED_CENTER_US` reprenant valeur de la pulsation RC (en microseconde) lorsque le levier de vitesse est en position centrale. Comme mentionner dans le commentaire, le contrôle de vitesse est branché sur le canal 3 du récepteur.

•Ligne 6 : constante `DIR_CENTER_US` reprenant la valeur de la pulsation RC lorsqie le levier de direction est en position centrale (donc au repos). Le commentaire mentionne que le contrôle de direction est raccordé sur le canal 1.

•Ligne 7 : constante RANGE_US reprenant la gamme de mesure (en microseconde) autour du centre. Ainsi, pour la vitesse, la gamme de mesure ira de `SPEED_CENTER_US-RANGE_US` à `SPEED_CENTER_US+RANGE_US` .

•Lignes 9 et 10 : définition des variables `ch_speed` et `ch_dir` correspondant au broches 7 et 6 du Pico (instance de la classe `Pin`). Ces mêmes broches du Pico raccordées respectivement sur les canaux 3 et 1 du récepteur radio RF.

•Lignes 12 à 13 : définition de la fonction `map()` déjà présenté dans cette ce projet. La fonction map() permet de réaliser une interpolation linéaire entre l'entrée (signal en microseconde) et la sortie (vitesse Zumo entre -300 et +300).

•Lignes 15 à 16 : création de l'instance du `ZumoShield` (variable `z`) et appel de la méthode `play_blip()` produisant un son signalant que le robot est prêt.

•Lignes 33 et 34 : lorsqu'il y a une marche arrière, la différence de vitesse doit être inversée (passer en négatif) pour que le Zumo tourne dans le sens attendu (conformément au graphique des directions).

•Lignes 35 et 36 : La vitesse n'est pas négative mais en plus, elle est réputée nulle ! Nous sommes probablement dans le cas d'une rotation sur place. Dans pareil cas, la différence de vitesse est doublée (passe de ± 100 à ± 200), qui permet au robot de tourner plus énergiquement sur lui-même.

•Ligne 17 : début de la boucle infinie `while True` exécutant les lignes 18 à 49 (encore et encore). Il s'agit du corps du script réalisant les opérations suivantes : (1) acquisition des signaux RC sur les canaux 1 et 3, (2) transformation en vitesse et direction, (3) application de la vitesse et direction sur le ZumoShield.

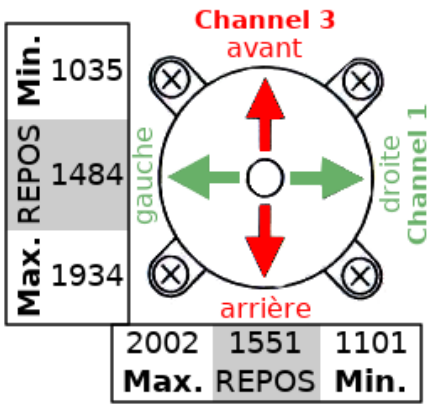
Détails de la boucle principale

•Ligne 18 : acquisition du signal RC (en microsecondes) correspondant à la position du levier de vitesse (`us_speed`). Le second paramètre `pulse_level` est à 1 pour mesurer le temps durant lequel l'impulsion reste au niveau haut.

•Ligne 19 : acquisition du signal RC (en microsecondes) correspondant à la position du levier de direction (`us_dir`).

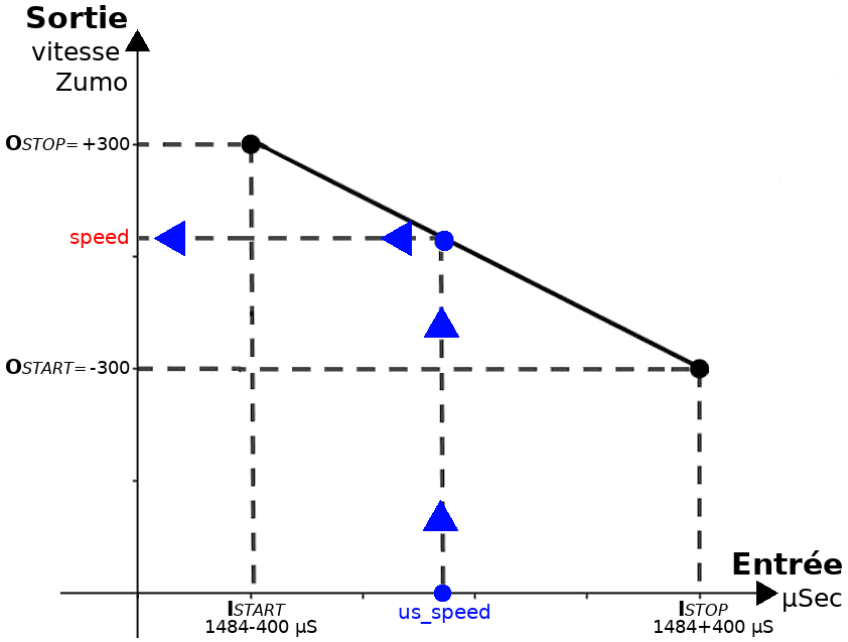
•Lignes 21 à 24 : si time_pulse_us() retourne une valeur négative, c'est qu'il n'y a plus de signal RC sur l'entrée. Que cela soit sur le canal de direction ou le canal de vitesse, il est préférable de mettre le Zumo à l'arrêt avec `z.motor.stop()` . L'instruction `z.led.toggle()` change l'état de la LED du Zumo à chaque appel pour signaler la perte du signal RC. Enfin l'instruction `continue` permet de démarrer immédiatement une nouvelle itération de la boucle. La LED clignotera aussi longtemps que la connexion RC est perdue.

• Ligne 26 : Utilisation de la fonction `map()` pour transformer la longueur d'impulsion du canal 3 de vitesse (de 1035 à 1934) en vitesse Zumo (+300 à -300). La variable `speed` contiendra donc la vitesse du `ZumoShield` .



08RI19 - relevé des minimas, repos et maximas

Le lecteur attentif remarquera l'inversion du signe dans les paramètres d'appel de la fonction `map()`. Cela est nécessaire car à commande de vitesse maximale (levier vers l'avant), le temps d'impulsion est minimale à 1035 µSec là où il faut obtenir la vitesse Zumo maximale (+300).



08RI20 – Conversion `map()` pour la vitesse du Zumo

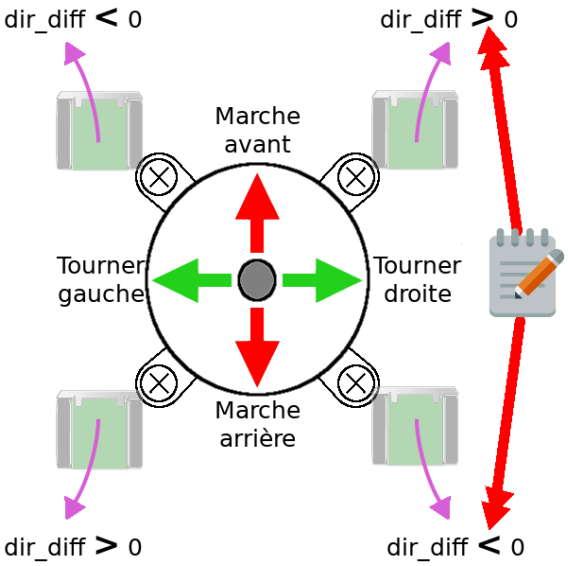
Lorsque le levier est en position centrale, la vitesse `speed` est proche de zéro.

• Ligne 27 : Utilisation de la fonction `map()` pour transformer la longueur d'impulsion du canal 1 de direction (de gauche=2002 à droite=1101) en différence de vitesse devant être appliquée entre les deux chenilles (-100 à +100). La variable `dir_diff` contiendra donc la différence de vitesse. Selon la formule, avec le levier totalement à droite, le `us_dir` tend vers 1101 µSec (ou plutôt vers `DIR_CENTER_US-RANGE_US`) de sortie que la valeur de sortie `diff_dir` tend vers `+100` . A contrario, avec le levier totalement à gauche, `us_dir` tend vers `DIR_CENTER_US+RANGE_US` (donc vers 2002 µSec) produisant ainsi une valeur de sortie `dir_diff` tendant vers -100. En position centrale, la valeur de sortie est donc proche de 0.

•Lignes 29 et 30 : si la valeur absolue (donc sans le signe) de `speed`, la vitesse, est inférieure à 25 alors la vitesse est forcée à 0. Par conséquent, lorsque le levier de vitesse est au centre (ou très proche de celui-ci) la Zumo passe directement à l'arrêt. Cela évite les grognements moteurs car dans le cas contraire les moteurs ne seraient jamais vraiment à l'arrêt. La vitesse moteur évolue donc dans la gamme -300 → -25 et +25 → +300.

•Lignes 31 et 32 : même procédé que ci-dessus pour annuler la différence de vitesse `dir_diff` si celle-ci est proche de 0 (donc levier proche du centre). Cela évitera au Zumo de modifier sensiblement sa trajectoire si le levier est proche du centre. La gamme de différence de vitesse est donc -100 → -10 et +10 → +100.

•Lignes 33 et 34 : Lors d'un virage en marche arrière (donc `speed<-25`), la différence de vitesse doit être inversée pour tourner dans la bonne direction. Cela se fait en multipliant `dir_diff` par -1 .



08RI21 – Influence de dir_diff sur la rotation

•Lignes 35 et 36 : une valeur de `dir_diff` entre 25 et 100 est parfait pour courber la trajectoire durant le déplacement. Cependant une rotation sur place (donc à `speed=0`) avec une différence de vitesse de 100 produit une rotation sur place un peu poussive ! Multiplier la valeur de `dir_diff` par deux produira une rotation plus énergique (`dir_diff = dir_diff * 2`).

•Ligne 37 et 38 : calcul de la vitesse de la chenille gauche (`speed_left`) en ajoutant la `dir_diff`, que l'on soustraira de la vitesse de la chenille droite (`speed_right`). L'utilisation de `int()` permet d'assurer un résultat entier.

•Lignes 39 à 46 : borne les valeurs de `speed_right` et `speed_left` entre -400 et +400 (les maxima de `ZumoShield.motors`)

•Ligne 48 : application des vitesses calculées sur les moteurs du ZumoShield.

•Ligne 49 : petite pause de 1/10 secondes (100 millisecondes) avant de passer à la prochaine itération de la boucle `while`.

1.5.Encore plus

Avec une télécommande disposant de plus de deux canaux, il est possible d'envisager des fonctionnalités avancées :

•Activer le suivit de ligne et laisser le robot en pilote automatique

•Créer un klaxon à l'aide du Buzzer.

•Ajouter un module audio pour utiliser des effets sonores avec le Zumo.

•Ajouter une mini catapulte déclenchable à distance.

•Activer/désactiver un mode permettant de faire des quart de tour calibrés avec le magnétomètre.

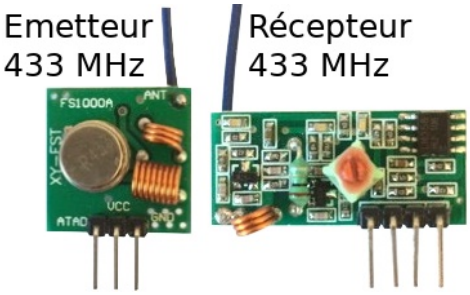
2.Commande RFM69

2.1.Les modules radios

Il existe de très nombreux modules de transmission radio allant des modules populaires (à l'usage discutable) aux modules high-tech très coûteux.

2.1.1.Les modules radios bon-marchés

Les Makers ont souvent l'occasion d'utiliser des modules RF 433 MHz rudimentaires comme le couple émetteur/récepteur ci-dessous.



08RI25 – Module de transmission 433 MHz

Si ces modules sont excessivement bons marchés, ils viennent aussi avec une série d'inconvénients majeurs :

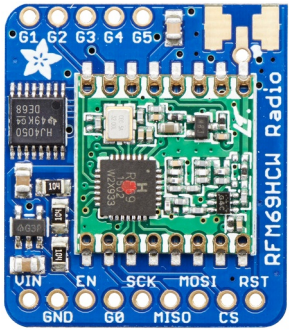
- Transmission sur des distances relativement faibles (ordre de 10 mètres) suivant la pollutation radio.
- Unidirectionnel (souvent)
- Pas de contrôle d'erreur
- Transmission des données en clair
- Tous les modules partage un unique canal de communication (collisions inévitables s'il y a plusieurs émetteurs/récepteurs).
- Fiabilité aléatoire (sensible aux bruits environnent)
- Peu de données technique et certification inexistante (généralement)

La bande de fréquence des 433 MHz est qualifiée de « fréquence libre » et utilisée par de très nombreux appareils domestiques (télécommandes, appareils domotiques, jouets télécommandés, etc). De fait la gamme des 433 MHz est aussi la plus polluées des ondes radios.

2.1.2.Le module RFM69

Entre le bas gamme et les modules radio spécialisés, il existe les module RFM69 qui fonctionnent aussi sur la gamme de fréquence des 433 MHz mais en apportant un niveau de fiabilité considérable aux communications radio.

Le breakout RFM69HCW 433MHz proposé par Adafruit Industries (Adafruit 3071) permet de raccorder facilement un module RFM69 sur des microcontrôleur en logique 5V (comme Arduino) ou logique 3,3V (comme le Pico ou le Raspberry-Pi).

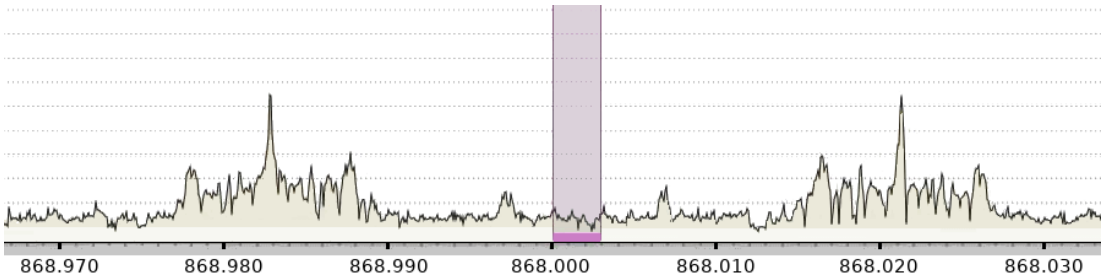


08RI26 – Module RFM69

Ce qui distincte un module RFM69 des autres modules bon marché ce sont ses caractéristiques ! Raisons pour lesquelles ce module est utilisé dans cette section.

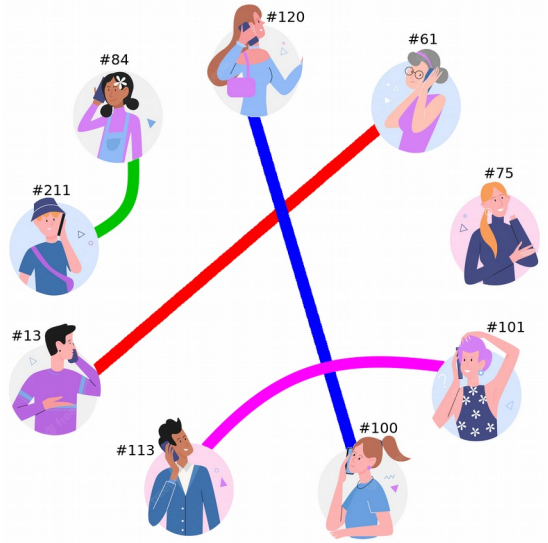
1.**Transmission de type *Packet Radio***. Une transmission par paquet de données (60 octets max) fiable, dans les **deux sens** avec utilisation de préambule, compteur de paquet, somme de contrôle, ré-envoi automatique. La possibilité d'utiliser des accusés de réception. *Packet radio* permet de réaliser des connexions point-à-point à 300 Kbits/s . En gros, c'est peu comme sur le réseau Internet filaire mais par les airs.

2.**Subdivision en « canaux »**. La fréquence de la communication est ajustable. Une communication typique se fait entre -30KHz à +30KHz autour de la fréquence de transmission. Cela permet d'établir des « canaux » de communications sur des fréquences distinctes sans risque de collision. Il est recommandé d'utiliser des écarts de 100 KHz pour éviter des recouvrement de spectre entre deux « canaux » connexe.



08RI27 – Spectre pour transmission pour 868,0 MHz (version US)

3.**Adresse de nœud** : chaque module radio peut avoir une adresse de nœud dit `NODE_ID`. Cela signifie que plusieurs cartes peuvent partager la même fréquence mais ne réceptionner que les messages qui lui sont spécifiquement adressés. Le module RFM69 permet également une communication de type *broadcast* ou le message est annoncé à tout venant sur la fréquence radio.



08RI28 – Communication entre NODE_ID

4.**Communication chiffrée**: l'algorithme de chiffrement AES est utilisé pour encrypter les données avec une clé de 128 bits. Les modules participant à un échange de données doivent donc utiliser la **même fréquence** et la **même clé de chiffrement**.

5.**Communication longue distance**. La distance de communication varie de 500m à 5 km. Celle-ci dépend de la fréquence utilisé, de la puissance d'émission mais surtout des antennes et des éléments obstruant la ligne de visée entre les modules radio.

6.**Connecteur d'antenne**. La distance de communication dépend forcement de la qualité de l'antenne. Si le module offre déjà des performances honorables avec un simple fil en guise d'antenne il faudra opter pour une connectique et une antenne appropriée pour obtenir des performances optimales.

Bien que cela paraisse évident, il faudra deux modules radio RFM69 (ou plus) pour établir une liaison radio.

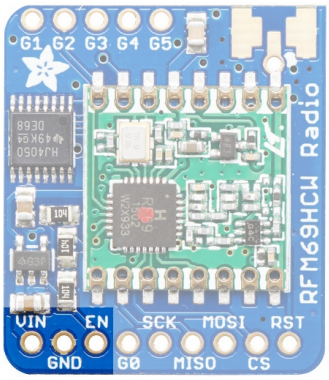
2.2.Tester le module radio

2.2.1.Détails du module radio

Voici le détail du brochage RFM69HCW 433MHz proposé par Adafruit Industries (Adafruit 3071).

Sauf broches GPIOs, toutes les broches du modules radio sont compatibles avec une logique 3,3V ou 5V. Le niveau logique est fixé par la tension d'alimentation du module (Vin).

Broches d'alimentation



08RI29 – Broches d'alimentation

•**GND:** masse commune.

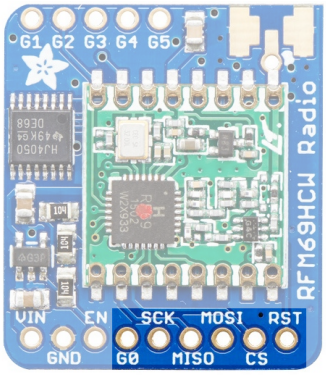
•**Vin:** alimentation du module entre 3.3 et 6V. Le régulateur de tension de la carte produit une tension de 3,3V pour l'élément radio. La consommation sur cette broche est de l'ordre de 30mA en écoute active et jusque 150mA durant une transmission.

•**EN:** broche *Enable* permettant de désactiver le régulateur et l'élément radio. La broche EN est maintenue au niveau haut à l'aide d'une résistance pull-up. Placer la broche au niveau bas pour désactiver le module.

Interface SPI

Le module communique avec le microcontrôleur par l'intermédiaire du bus SPI (donc via les signaux MISO, MOSI, CLK et ChipSelect).

Le module expose également la broche RESET et G0 (INTERRUPT) offrant un meilleur contrôle de la communication entre le microcontrôleur et le module radio.



08RI30 – Broches de communication

•**MISO:** acronyme de *Master In Slave Out*, cette broche permet au module radio d'envoyer des données vers le microcontrôleur.

•**MOSI:** acronyme de *Master Out Slave In*, cette broche permet au microcontrôleur d'envoyer des données vers le module radio.

•**SCK:** signal d'horloge permettant de synchroniser l'échange des bits entre le microcontrôleur et le module radio.

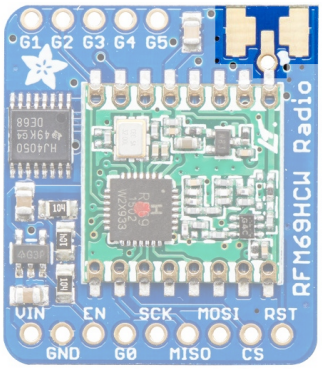
•**CS:** acronyme de *Chip Select*, placer la broche au niveau bas permet d'activer la communication avec le module radio et débuter une nouvelle transaction avec le module radio. Lorsque plusieurs modules SPI sont connectés sur un même bus SPI, les signaux CS permettent d'activer un seul module à la fois (ex: un module radio + un afficheur TFT).

•**RST:** cette broche permet de réinitialiser le module radio en plaçant la broche au niveau.

•**G0:** broche GPIO 0 du module radio. Cette broche peut également être configurée comme broche d'interruption (**IRQ**, Interrupt Request). Le module radio peut manipuler cette broche pour notifier le microcontrôleur. Cette broche est en logique 3.3V uniquement.

Connecteur d'antenne

Le connecteur d'antenne se trouve dans le coin supérieur droit du module RFM.



08RI31a – Connecteur d'antenne

Il existe trois façons de connecter une antenne sur la carte :

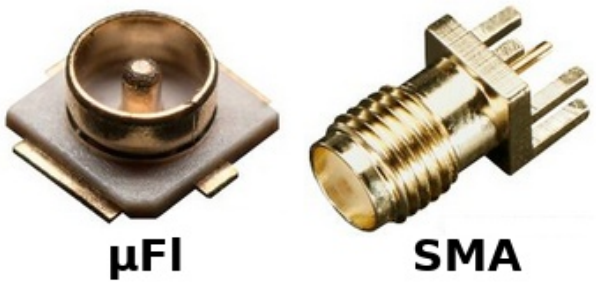
•Un fil soudé dans le trou d'antenne (aussi appelé « dipôle filaire »). C'est la méthode la plus simple et la plus économique pour équiper le module d'une antenne. Si la section du fil n'est pas spécialement important, sa longueur doit être de 17,32cm pour réaliser une antenne quart d'onde optimale.

•Un connecteur d'antenne µFl pour y brancher une antenne. Les connecteurs µFl sont généralement utilisé sur les périphériques d'ordinateur portable.

•Un connecteur d'antenne SMA pour carte. Ce connecteur universel prend généralement place sur le bord de carte et permet d'y brancher de nombreux type d'antenne. Il s'agit de la meilleure option de raccordement.

Il est impératif d'utiliser une antenne avec le module radio. Aucune transmission, même à 10 cm, n'est envisageable sans antenne.

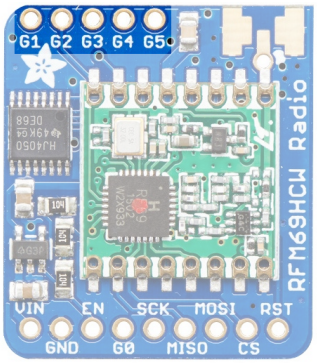
*Une antenne de longueur adéquate (**17,32cm** pour 1/4 d'onde) permet d'émettre un maximum du signal dans les airs tout en limitant la réflexion du signal (en bout d'antenne) vers le module émetteur. Lorsque l'antenne est manquante, 100 % du signal est renvoyé vers le module.*



08RI31b – connecteurs d'antenne

GPIOs du module radio

Le module radio propose 5 GPIOs supplémentaires (de G1 à G5) pour activer des interruption sur des conditions spécifiques.



08RI32 – GPIO supplémentaires

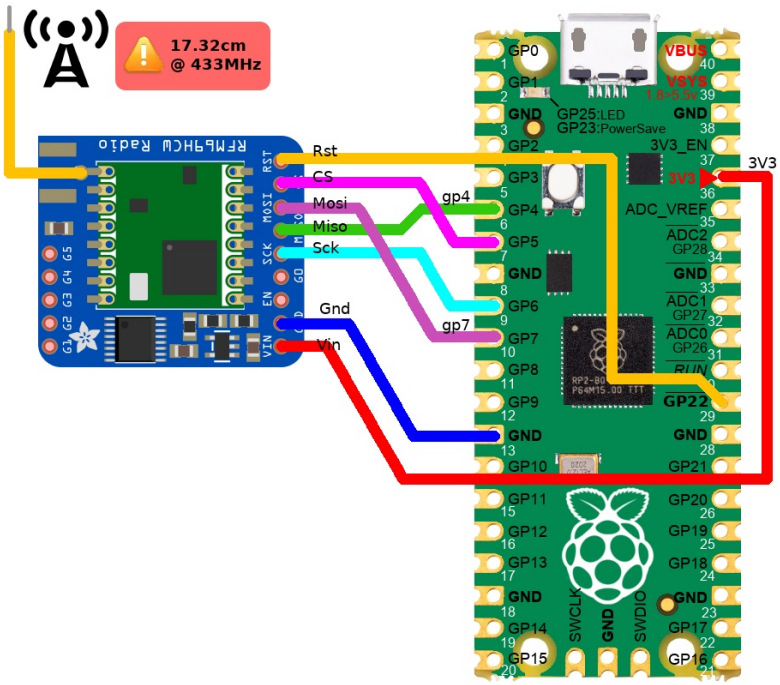
Ces GPIOs n'étant généralement pas utilisés pour créer des notification (ou fonction radio), ils peuvent être réassignés à un usage quelconque dans un projet.

A noter que ces broches sont en logique 3.3V uniquement.

2.2.2.Brancher sur un Pico

Le module radio utilise le bus SPI pour communiquer avec le microcontrôler, voici les liaison qui doivent être réalisée avec un Raspberry-Pi Pico.

Les broches utilisées ici sont les mêmes que celles disponibles sur l'adaptateur Pico-Zumo, le montage pourra être facilement transposé dans les raccordement du Zumo Robot.



03RI35 – Raccordement d'un RFM69 sur un Pico

RFM69HCW	PICO

RST	GP22
CS	GP5 (Slave Select)
MOSI	GP7 (Mosi)
MISO	GP4 (Miso)
SCK	GP6 (Clock)
GND	GND
VIN	3,3V

2.2.3.Installer la bibliothèque

Avant de pourvoir tester un module radio, il est nécessaire d'installer la bibliothèque `rfm69.py` sur la plateforme MicroPython.

La bibliothèque est disponible dans le dépôt GitHUb `esp8266-upy/rfm69/` .

•https://github.com/mchobby/esp8266-upy/tree/master/rfm69/lib

L'utilitaire MicroPython `mpremote` permet également d'installer une bibliothèque plus facilement grâce à l'option `mip` :

```
$ mpremote mip install github:mchobby/esp8266-upy/rfm69
```

2.2.4.Tester la connectique

Malgré des performances nettement supérieures du bus SPI, celui-ci souffre d'un manquement gênant ! Il s'agit de la signalisation d'un état d'erreur sur le bus SPI.

Il n'y a donc aucun moyen de savoir si une requête sur le bus SPI est bien reçu et interprétée par un composant SPI nouvellement raccordé.

Il n'y a donc aucun moyen d'être averti si le composant est mal raccordé sur le bus ou défaillant !

Il est utile d'avoir une procédure fiable permettant de vérifier la connectique. Cela se fait à l'aide d'un petite programme/script permettant d'interroger le périphérique cible et d'afficher des informations qu'il contient (ex : paramètres par dafaut).

Le script `rfmtest_config.py` permet de lire la configuration du module RFM et d'en afficher le contenu.

Le script de test est également disponible dans le dépôt GitHub du projet

•https://github.com/mchobby/micropython-zumo-robot/blob/main/extras/RFM69/rfmtest_config.py

```
from machine import SPI, Pin
from rfm69 import RFM69

def dbm_to_mw(dBm):
  return 10**((dBm)/10.)

spi = SPI(0, baudrate=50000, polarity=0, phase=0,
          firstbit=SPI.MSB)
nss = Pin( 5, Pin.OUT, value=True )
rst = Pin( 22, Pin.OUT, value=False )
```

```
rfm = RFM69( spi=spi, nss=nss, reset=rst )
rfm.frequency_mhz = 433.1
rfm.bitrate = 250000 # 250 Kbs
rfm.frequency_deviation = 250000 # 250 KHz
rfm.tx_power = 13  # 13 dBm = 20mW
rfm.encryption_key = ( b"\x01\x02\x03\x04\x05\x06\x07\x08\x01\x02\
x03\x04\x05\x06\x07\x08" )

print( 'RFM version     :', rfm.version )
print( 'Freq            :', rfm.frequency_mhz )
print( 'Freq. deviation :', rfm.frequency_deviation, 'Hz' )
print( 'bitrate         :', rfm.bitrate, 'bits/sec' )
print( 'tx power        :', rfm.tx_power, 'dBm' )
print( 'tx power        :', dbm_to_mw(rfm.tx_power), 'mW' )
print( 'Temperature     :', rfm.temperature, 'Celsius' )
print( 'Sync on         :', 'yes' if rfm.sync_on else 'no' )
print( 'Sync size       :', rfm.sync_size )
print( 'Sync Word Length:', rfm.sync_size+1, "(Sync size+1)" )
print( 'Sync Word       :', rfm.sync_word )
print( 'CRC on          :', rfm.crc_on )

print( 'Preamble Lenght :', rfm.preamble_length )
print( 'aes on          :', rfm.aes_on )
print( 'Encryption Key  :', rfm.encryption_key )
```

Exécuté dans une session REPL, le script produit le contenu suivant lorsque le module RFM est interrogé.

```
RFM version     : 36
Freq            : 433.1
Freq. deviation : 250000.0 Hz
bitrate         : 250000.0 bits/sec
tx power        : 13 dBm
tx power        : 19.95262 mW
Temperature     : 22 Celsius
Sync on         : yes
Sync size       : 1
Sync Word Length: 2 (Sync size+1)
Sync Word       : bytearray(b'-\xd4')
CRC on          : 1
Preamble Lenght : 4
aes on          : 1
Encryption Key  : bytearray(b'\x01\x02\x03\x04\x05\x06\x07\x08\
x01\x02\x03\x04\x05\x06\x07\x08')
```

Le fait de pouvoir initialiser les paramètres et relire la configuration du module RFM confirme que celui est bien raccordé.

En cas de montage incorrecte, le script `rfmtest_config.py` produit le résultat suivant :

```
Traceback (most recent call last):
  File "<stdin>", line 20, in <module>
  File "/lib/rfm69.py", line 209, in __init__
  File "/lib/rfm69.py", line 442, in __idle
  File "/lib/rfm69.py", line 250, in set_mode
RuntimeError: Change mode timeout!
```

2.2.5.Tester la communication

Avant de se lancer dans des montages plus complexe, le plus simple reste encore de tester la communication entre deux RFM69+Pico.

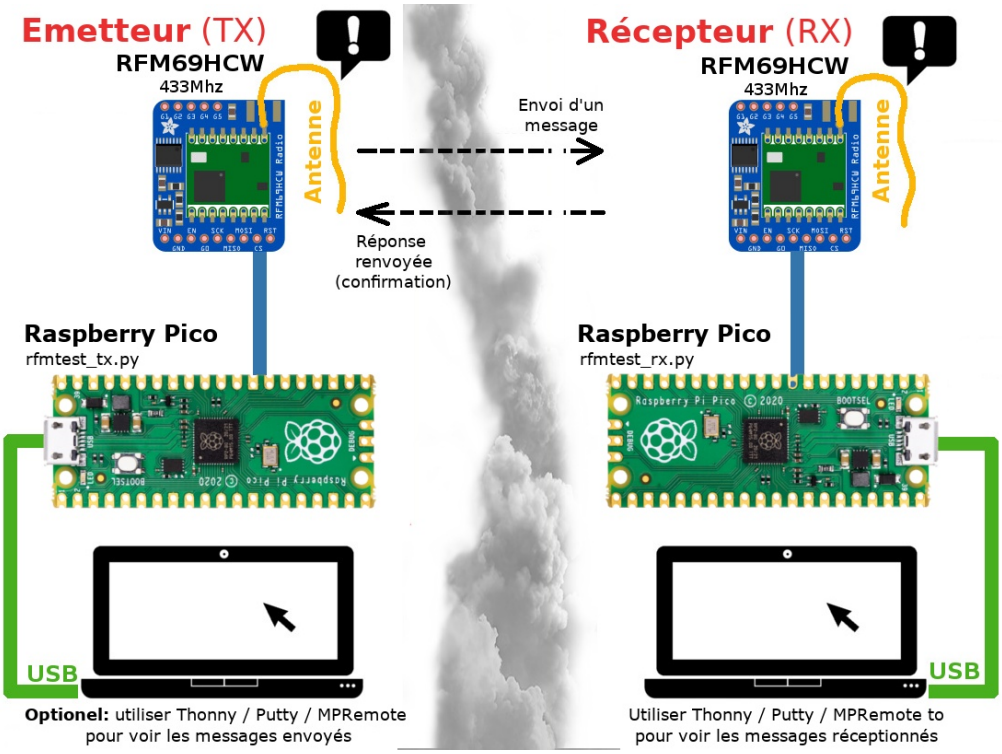
Pour faciliter la mise en œuvre du test :

1.chaque RFM69+Pico sera connecté sur un ordinateur différent

2.le premier RFM69 sera considéré comme émetteur de message

3.le second RFM69 sera considéré comme récepteur et renverra une confirmation.

De la sorte, il sera possible de survoler la plupart des fonctionnalités du module RF. Il va de soit que les rôles émetteur/récepteur sont volontairement fixé faciliter la compréhension. Un module RFM69 peut assumer les deux rôles.



08RI36 – Test émetteur / récepteur RFM69

Récepteur : rfmtest_rx.py

Le script `rfmtest_rx.py` permet de réceptionner les données de l'émetteur et de fournir un accusé de réception pour chaque message reçu.

Le script `rfmtest_rx.py` est également disponible dans les sources du projet micropython-zumo-robot .

- https://github.com/mchobby/micropython-zumo-robot/blob/main/extras/RFM69/rfmtest_rx.py

Voici le contenu du script récepteur :

```
01: from machine import SPI, Pin
02: from rfm69 import RFM69
03: import time
04:
05: FREQ           = 433.1
06: ENCRYPTION_KEY = b"\x01\x02\x03\x04\x05\x06\x07\x08\x01\x02\
x03\x04\x05\x06\x07\x08"
07: NODE_ID        = 100 # ID of this my node node
08:
09: spi = SPI(0, miso=Pin(4), mosi=Pin(7), sck=Pin(6), polarity=0,
phase=0, firstbit=SPI.MSB)
10: nss = Pin( 5, Pin.OUT, value=True )
11: rst = Pin( 22, Pin.OUT, value=False )
12:
13: rfm = RFM69( spi=spi, nss=nss, reset=rst )
```

```
14: rfm.frequency_mhz = FREQ
15:
16: rfm.encryption_key = ( ENCRYPTION_KEY )
17: rfm.node = NODE_ID
18:
19: print( 'Freq            :', rfm.frequency_mhz )
20: print( 'NODE            :', rfm.node )
21: print("Waiting for packets...")
22: while True:
23:   packet = rfm.receive( with_ack=True )
24:   # packet = rfm.receive(timeout=5.0)
25:   if packet is None:
26:     pass
27:   else:
28:     print( "Received (raw bytes):", packet )
29:     packet_text = str(packet, "ascii")
30:     print("Received (ASCII):", packet_text)
31:     print("-"*40)
```

Voici quelques détails concernant le fonctionnement de ce script :

•Lignes 1 à 3 : importation des modules et classes nécessaires.

•Lignes 5 et 6 : constantes définissant la fréquence et la clé de cryptage qui seront utilisées. La clé de cryptage est ici constituée d'une structure bytes de 16 octets (128 bits). **La fréquence et la clé de cryptage doit être identique pour tous les modules RFM69**.

•Ligne 6 : définition de la constante NODE_ID identifiant le récepteur. Pour contacter le récepteur, il faudra utiliser son NODE_ID qui est ici 100.

•Ligne 9 : déclaration du bus SPI permettant de communiquer avec le module RFM69. Les broches utilisées par le bus doivent être précisées durant la création, a savoir les GPIOs 4, 6 et 7. Les paramètres `polarity` et `phase` défini le mode de fonctionnement du bus, la communication sera impossible si ces éléments sont incorrectement configurés. Le paramètres `firstbit=SPI.MSB` indique que les bits transitent sur le bus en commençant par le plus signification (celui qui à la poids le plus élevé).

•Ligne 10 : définition de la broche `nss` (GPIO 5) utilisée pour activer le module RFM69 lorsqu'une transaction débute entre le microcontrôleur et le module RFM. Le module étant activé lorsque le signal est au niveau bas, la broche est créée en initialisant son état au niveau haut (`value=True`).

•Ligne 11 : définition de la broche `rst` (GPIO 22) pour réinitialiser le module RFM69. Le module étant réinitialisé lorsque le signal est au niveau haut, la création de la broche `rst` se fait avec un état initial au niveau bas (`value=False`).

•Ligne 13 : création de l'instance la classe `RFM69` dans la variable `rfm`. Le bus SPI, la broche de sélection et la broche reset sont communiqués au constructeur. La classe `RFM69` prendra en charge les détails de la communication avec le module RFM.

•Lignes 14 à 17 : fixe la fréquence de communication (433,1MHz), la clé de cryptage (de 128 bits) et l'identification du nœud (n° 100).

•Lignes 19 à 21 : affichage de quelques informations utiles dans la session REPL. Le derniers messages « Waiting for packets... » indique que le script attend l'arrivée de messages.

•Ligne 22 : Boucle infinie exécutant le corps du script (entre les lignes 23 et 31). Le corps du script en : (1) attendre l'arrivé d'un message et (2) afficher celui-ci dans la session REPL.

Corps du script

•Ligne 23 : demande au module RFM de passer en écoute active. Cet appel est bloquant et la méthode `receive()` attend de recevoir un message avant de rendre la main au script principal. Le paramètre `with_ack=True` indique à la méthode de réception de renvoyer un accusé de réception à l'émetteur lorsqu'un message est reçu. Enfin la méthode `receive()` retourne des données binaires réceptionnées (`bytearray`).

•Ligne 24 : cette ligne en commentaire (donc non exécutée) présente un appel non bloquant de la méthode `receive()`. Si aucun message est réceptionné dans l'intervalle d'un `timeout` prédéfini (ici 500ms) alors la méthode retournera `None`. Ce type d'appel permet au corps du message d'exécuter des tâches annexes à intervalle régulier.

•Lignes 25 et 26 : si aucune donnée est réceptionnée alors `packet` sera `None` (ce qui peut être le cas si la ligne 24 est utilisée). Dans pareil cas, l'instruction `pass` indique que rien ne doit être exécuté par le test `if` .

•Lignes 28 à 31 : la variable packet contient des données binaires (par exemple : `b'Message 120!'`). Si l'information est parfaitement lisible grâce au codage ASCII, il ne s'agit ni plus, ni moins, d'un `bytearray` et donc d'**information strictement binaire**.

•Ligne 29 : transformation des données binaire en chaîne de caractères (unicode) à l'aide de `str(packet, "ascii")` . Il est aussi possible de décode le contenu binaire (ASCII) avec l'instruction `packet_text = packet.decode("ASCII")`

Résultat du script récepteur

Une fois le script démarré dans une session REPL, il présente les informations suivantes et attend les messages entrant.

```
Freq            : 433.1
NODE            : 100
Waiting for packets...
```

Une fois le script émetteur `rfmtest_tx.py` démarré sur un autre ensemble Pico+RFM69 , le récepteur commence à afficher les messages réceptionnés (et renvoyer des accusés de réception).

```
Freq            : 433.1
NODE            : 100
Waiting for packets...
Received (raw bytes): bytearray(b'Message 1!')
Received (ASCII): Message 1!
----------------------------------------
Received (raw bytes): bytearray(b'Message 2!')
Received (ASCII): Message 2!
----------------------------------------
Received (raw bytes): bytearray(b'Message 3!')
Received (ASCII): Message 3!
----------------------------------------
Received (raw bytes): bytearray(b'Message 4!')
Received (ASCII): Message 4!
----------------------------------------
```

Emetteur : rfmtest_tx.py

Le script `rfmtest_tx.py` permet de d'envoyer des messages (données) vers le script récepteur et attend l'accusé de récepteuretteur et de fournir un accusé de réception pour chaque message reçu.

Le script `rfmtest_tx.py` est également disponible dans les sources du projet micropython-zumo-robot .

•https://github.com/mchobby/micropython-zumo-robot/blob/main/extras/RFM69/rfmtest_tx.py

Voici le contenu du script émetteur :

```
01: from machine import SPI, Pin
02: from rfm69 import RFM69
03: import time
04:
05: FREQ           = 433.1
06: ENCRYPTION_KEY = b"\x01\x02\x03\x04\x05\x06\x07\x08\x01\x02\
x03\x04\x05\x06\x07\x08"
07: NODE_ID        = 120
08: RECEIVER_ID    = 100
09:
10: spi = SPI(0, miso=Pin(4), mosi=Pin(7), sck=Pin(6),
baudrate=50000, polarity=0, phase=0, firstbit=SPI.MSB)
11: nss = Pin( 5, Pin.OUT, value=True )
12: rst = Pin( 22, Pin.OUT, value=False )
13:
14: rfm = RFM69( spi=spi, nss=nss, reset=rst )
15: rfm.frequency_mhz = FREQ
16: rfm.encryption_key = ( ENCRYPTION_KEY )
17: rfm.node    = NODE_ID
18:
19: print( 'Freq         :', rfm.frequency_mhz )
20: print( 'NODE         :', rfm.node )
21: print( 'Receiver NODE:', RECEIVER_ID )
22: counter = 1
23: rfm.ack_retries = 3
24: rfm.ack_wait    = 0.5
25: rfm.destination = RECEIVER_ID
26: while True:
27:   print("Send message %i!" % counter)
28:   ack = rfm.send_with_ack(bytes("Message %i!" % counter ,
"utf-8") )
29:   print("   +->", "ACK received" if ack else "ACK missing" )
30:   counter += 1
31:   time.sleep(0.1)
```

Voici quelques détails utiles concernant le script :

•Lignes 5 et 6 : déclaration des constantes `FREQ` et `ENCRYPTION_KEY` avec des valeurs identiques à celle du script émetteur.

•Ligne 7 : déclaration de la constante `NODE_ID` contenant l'identification du nœud émetteur. Le nœud émetteur porte le n° 120. Cette information est importante pour que le récepteur puisse renvoyer l'accusé de réception.

•Ligne 8 : déclaration de la constante `RECEIVER_ID`. Il s'agit de l'identification du nœud vers lequel le message doit être envoyé (100, le `NODE_ID` du script récepteur).

•Lignes 10 à 18 : ne diffèrent en rien du script émetteur.

•Lignes 19 à 21 : affichage de quelques informations utiles : fréquence, n° de nœud, n° de nœud du récepteur (celui qui sera contacté).

•Ligne 22 : déclaration de la variable `counter` qui sera incrémenté de 1 à l'envoi de chaque nouveau message.

•Ligne 23 : configure le module RFM pour faire 3 tentatives de réception de confirmation (ACK) d'envoi d'un message.

•Ligne 24 : configure le module RFM pour attendre la réception de confirmation pendant 500ms (à chaque tentative).

•Ligne 25 : mentionne la destination des messages en fixant la valeur de la propriété `destination` avec l'`ID_NODE` du récepteur (soit 100).

•Ligne 26 : début de la boucle infinie (while True) exécutant le corps du script des lignes 27 à 31, corps qui envoi les messages de test.

•Ligne 27 : informe l'utilisateur (via la session REPL) de l'envoi d'un message

•Ligne 28 : formatage et envoi du message `"Message %i!" % counter`.

□le compteur `counter` est inclus dans le message produisant ainsi les messages `"Message 1!", "Message 2!", "Message 3!"` et ainsi de suite.

□La méthode `rfm.send_with_ack()` permet d'envoyer un message et d'attendre l'accusé de réception. Cependant, `send_with_ack()` ne peut transmettre que des données binaires (des octets codés en ASCII).

□La chaîne de caractères unicode `"Message 1!"` doit donc être transformé en données binaires. Cela se fait à l'aide de `bytes( "Message 1!", "utf8")` le deuxième paramètre indiquant l'encodage de la source (donc celui de la chaîne de caractère).

□Enfin, la méthode `rfm.send_with_ack()` retourne `True` si un accusé de réception est renvoyé par le destinataire (et réceptionné).

•Ligne 29 : affichage de l'état ACK (accusé de réception) à l'aide de l'expression ternaire `"ACK received" if ack else "ACK missing"` qui retourne `"ACK received"` si `ack` est vrai (ou `"ACK missing"` si `ack` est faux).

•Ligne 30 : incrémentation du compteur `counter`. L'expression est équivalente à `counter = counter + 1`.

Résultats du script émetteur

Une fois le script `rfmtest_txt.py` démarré dans une session REPL, celui-ci affiche les message envoyés vers le récepteur.

```
Freq         : 433.1
NODE         : 120
Receiver NODE: 100
Send message 1!
   +-> ACK received
Send message 2!
   +-> ACK received
Send message 3!
   +-> ACK received
Send message 4!
   +-> ACK received
```

Le récepteur étant actif, le présent script `rfmtest_tx.py` reçoit les accusés de réceptions (ACK), ce qui est clairement visible dans les messages affichés ci-dessus.

2.3.A propos des antennes

Difficile d'aborder la transmission radio sans faire un petit crochet sur le sujet des antennes. L'antenne filaire déjà présentée au côté du RFM69 couvrira largement la plupart des besoins.

Bien qu'accessoire, un petit complément d'information sur les antennes pourrait s'avérer fort utile.

En cas de question ou de doute, le lecteur pourra se mettre en relation avec le club radio amateur local. Ces clubs regorge de membres expérimentés dans le domaine des radio fréquences.

Sauf erreur de la part de l'auteur, la législation n'autorise pas l'utilisation d'une antenne directionnelle pour l'émission à 433 MHz.

Une antenne directionnelle permet de concentrer une grande partie de la puissance d'émission dans une seule direction. A l'opposé, une antenne omnidirectionnelle irradie la puissance uniformément dans toutes les directions.

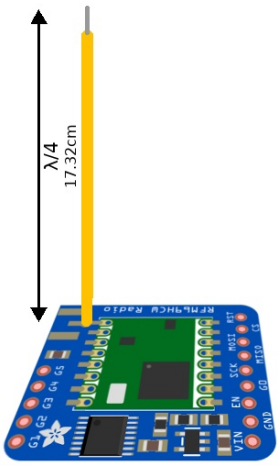
L'antenne dipôle filaire, déjà présentée dans cet ouvrage, est une antenne omnidirectionnelle.

A noter que la législation prévoit également une puissance d'émission maximal de 1mW de puissance apparente et -13dBm/10KHz de spectre (cf. IBTP Belgique).

2.3.1.Antenne filaire

L'antenne la plus simple est constituée d'un simple fil que l'on dresse à la verticale (le plus droit possible). Dans cette configuration, la masse de l'émetteur n'est pas utilisée.

Pour obtenir une réception optimal, il est important que la longueur de cette antenne soit un multiple de la longueur d'onde de la transmission.



07RI37 – Antenne filaire simple

Dans ce domaine, il est assez fréquent d'utiliser une antenne dite « quart d'onde » aussi notée λ/4 .

La longueur de l'antenne se calcule avec la formule suivante :

```
L = C / (4 * f)
```

Avec les paramètres suivants :

•C : vitesse de la lumière en m/s (vitesse de déplacement de l'onde électrique)

•f : fréquence d'utilisation en Hz

•4 : parce qu'il s'agit d'une antenne en quart d'onde.

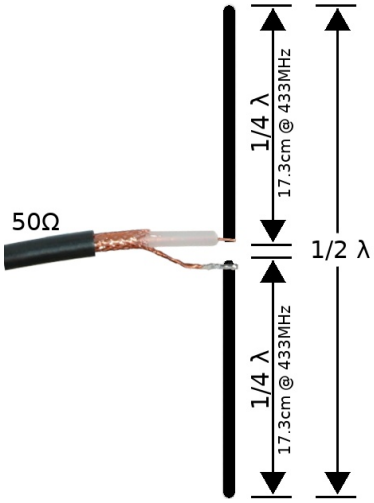
```
L = 3x10E8 / (4 x 433*10E6) = 0,1732m
```

La longueur idéal de l'antenne est de 17,32 cm

2.3.2.Antenne dipôle

Avec l'antenne dipôle, un second élément en quart de longueur d'onde participe à l'émission/réception et est raccordé sur la masse de l'émetteur.

Les deux éléments sont alignés et montés tête-bêche sur une pièce isolante située au centre de l'antenne. Il s'agit donc d'une antenne demi longueur d'onde λ/2 .

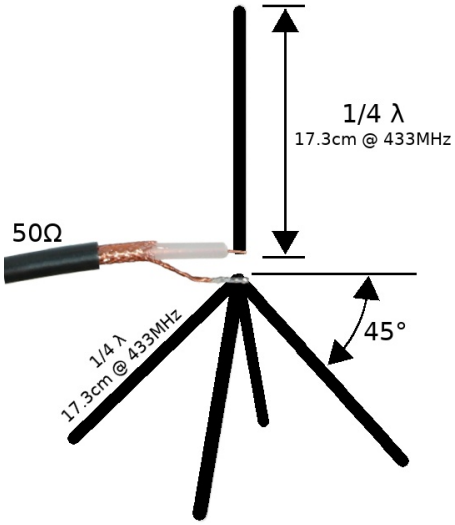


08RI38 – Antenne Dipôle

2.3.3.Antenne à plan de masse

Aussi dite « *Ground plane* », il s'agit aussi d'une antenne quart d'onde à laquelle est ajouter un plan de masse constitué de 4 radians inclinés à 45° vers le bas. Les 4 radians permettent d'établir un plan de masse de 50 Ohms.

Cette antenne omnidirectionnelle est généralement utilisée pour améliorer la réception radio. Il n'est pas rare de voir ce genre d'antenne sur les portails télécommandés.

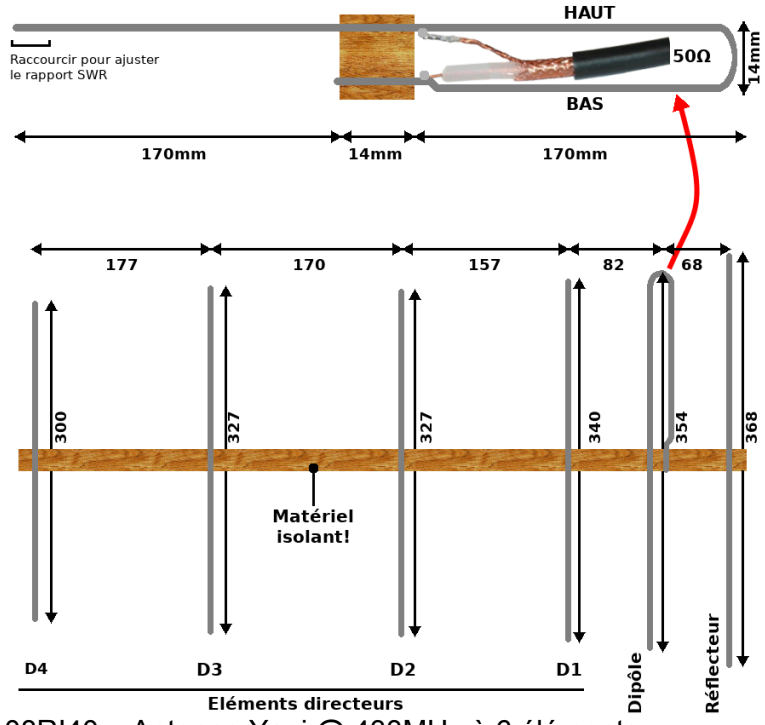


08RI39 – Antenne « *Ground Plane* »

2.3.4.Antenne Yagi

L'antenne Yagi est une antenne directionnelle (donc pour réception uniquement) permettant de concentrer la réception des signaux radio par l'intermédiaire d'éléments réflecteurs.

Les performances d'une antenne Yagi dépend du nombre d'éléments et de sa qualité de fabrication. Comme pour les antennes dipôle filaire et ground plane, les dimensions de l'antenne Yagi son calculée pour la fréquence à recevoir.



08RI40 – Antenne Yagi @ 433MHz à 6 éléments.

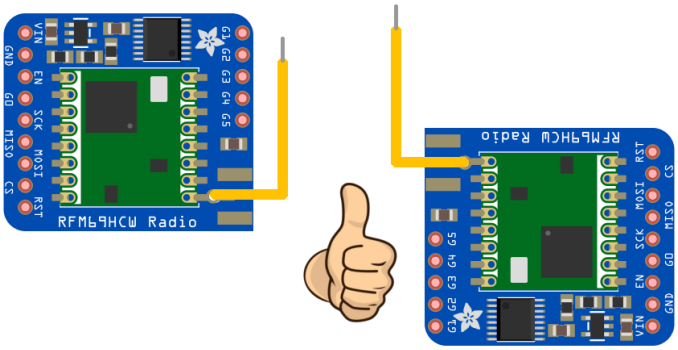
L'antenne Yagi à 6 éléments ci-dessus permet d'augmenter le gain de 11,2 dBi. Une antenne Yagi à 7 éléments offre le meilleur rapport performance/encombrement.

Pour finir, une antenne Yagi à 11 éléments doublera le gain. Le logiciel YagiMax permet de calculer des antennes Yagi optimisées.

2.3.5.Polarisation des antennes

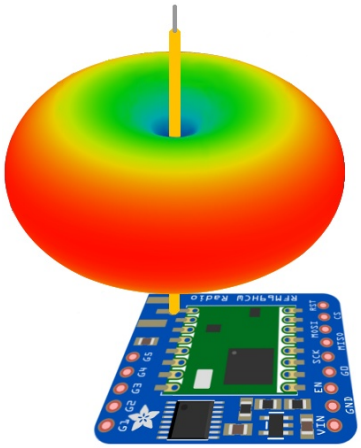
Concernant les antennes, il est également important d'aborder la polarisation des antennes.

Lorsque celle-cis sont orientées dans le même sens (**bonne polarisation**) alors la perte de puissance entre l'émetteur et le récepteur est minimale.



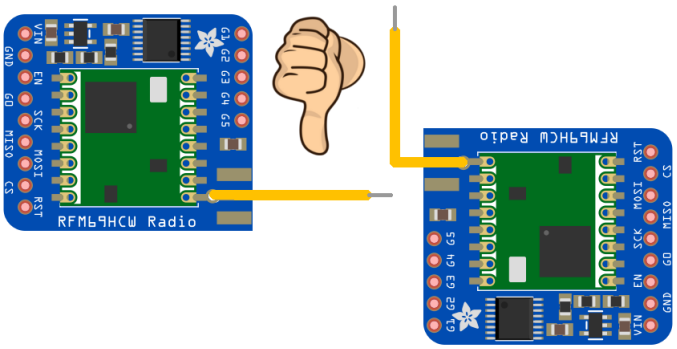
08RI41 – Bonne polarisation des antennes

C'est la conséquence du spectre de radiation de l'antenne qui ressemble à un donut. Le maximum de puissance (tout comme le maximum de sensibilité) se trouve à la périphérie du donut alors que le minimum d'émission (comme la sensibilité) se trouve au dessus de l'antenne.



08RI42 – Radiation d'une antenne filaire.

Lorsque l'orientation des antennes est inapproprié (**mauvaise polarisation**) alors presque toute la puissance est perdue entre l'émetteur et le récepteur.



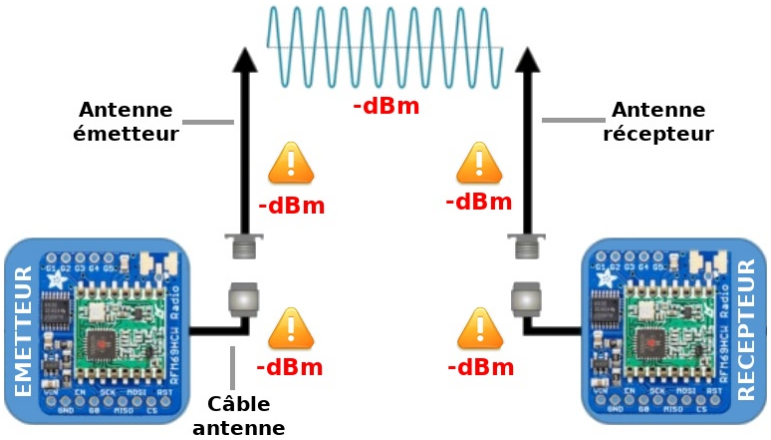
08RI43 – mauvaise polarisation des antennes

Il est assez facile de retrouver spectre des radiation d'une antenne en effectuant une recherche internet contenant les mots « antenna radiation ».

Le spectre de radiation d'une antenne dipôle est également un donut, le maximum de puissance se trouvent au niveau de la jonction isolante entre les deux éléments émetteurs.

2.3.6.Les pertes et gains

Dans le monde radio, la transmission par onde radio est confronté un ensemble de pertes tout au long du parcours de cet onde entre l'émetteur et le récepteur radio.



08RI44 – parcours de l'onde radio

Sur les 1mW de départ, il ne restera presque plus rien sur le récepteur. Cependant, ce presque rien peut varier plus ou moins fort en fonction des pertes encourue par le signal durant le trajet de l'onde.

Les pertes s'exprime en dBm = dB / mW mesuré. C'est une approche efficace pour mesurer la puissance absolue.

Ainsi, les éléments suivants seront la source de pertes :

•**Câble d'antenne** : dont la perte sera plus ou moins grande en fonction de la longueur et de la qualité du câble.

•**Connecteur cable-antenne** : donc la qualité de fabrication influe directement sur la perte de signal. Un connecteur inadapté ou de très mauvaise qualité peu réduire un signal à néant.

•**L'antenne** : L'utilisation d'une antenne inadéquate (mauvaisse gamme de fréquence), mal polarisée, ou de mauvaise qualité peut aussi absorber toute la puissance du signal sans rien émettre dans l'air ! Une antenne peut également avoir un gain, ce qui émet dans l'air un signal amplifié (attention à la législation applicable dans ce domaine).

•**L'air** : la transmission hertzienne représente la majorité de perte de signal. C'est le seul point sur lequel il n'est pas possible d'agir.

Ces points s'appliquent aussi bien du côté émetteur que du côté récepteur.

Remarques

Il serait tentant de penser que souder une antenne filaire simple serait la réponse idéale à toutes ces considérations concernant les pertes. Cela est parfaitement vrai

concernant le projet de télécommande à distance présenté ci-après. En effet, le Robot Zumo restera toujours à vue de la télécommande.

Dans le cadre de capture de données de terrain, il pourrait cependant être plus opportun de déporter l'antenne à l'extérieur du bâtiment puisque la somme des pertes accumulées serait fort probablement inférieure à la perte provoquée par la seule structure du bâtiment !

Dans le même ordre d'idée, une ligne de visée dégagée entre émetteur et récepteur (ex : bâtiment, arbre, colline, etc) est préférable pour avoir un perte de transmission aussi faible que possible dans les airs. Les ondes radios se comportant comme la lumière, celle-cis n'ont pas la possibilité d'adopter une trajectoire courbe pour éviter les obstacles !

Encore une fois, il peut être nécessaire de déplacer l'antenne en hauteur pour pouvoir profiter d'une vue plus dégagée permettant ainsi la réception d'un signal de bien meilleur qualité.

Il s'agira donc toujours de peser le pour et le contre du montage d'une antenne ou l'autre, de l'usage d'une connectique ou l'autre, de l'usage d'un câble coaxial ou autre.

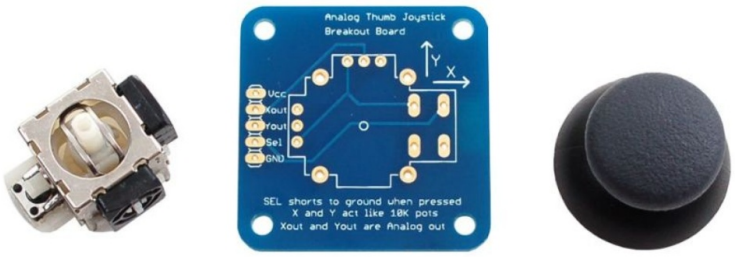
2.4.Créer une commande à distance

2.4.1.La télécommande

Joystick analogique 2 axes

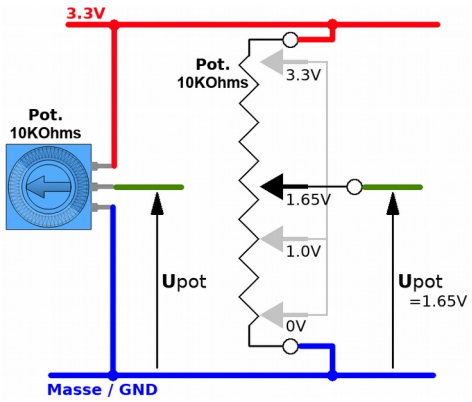
La commande sera bien entendu constituée d'un Raspberry-Pi Pico, un RFM69 et d'un joystick analogique 2 axes.

Le joystick analogique (ici Adafruit Industries 512) est constitué de deux potentiomètres de 10 KOhms permettant de relever la position horizontale (Xout) et verticale (Yout) et détecter la pression sur le joystick (Sel, placé à la masse lorsque pressé).



08RI45 – Potentiomètre analogique

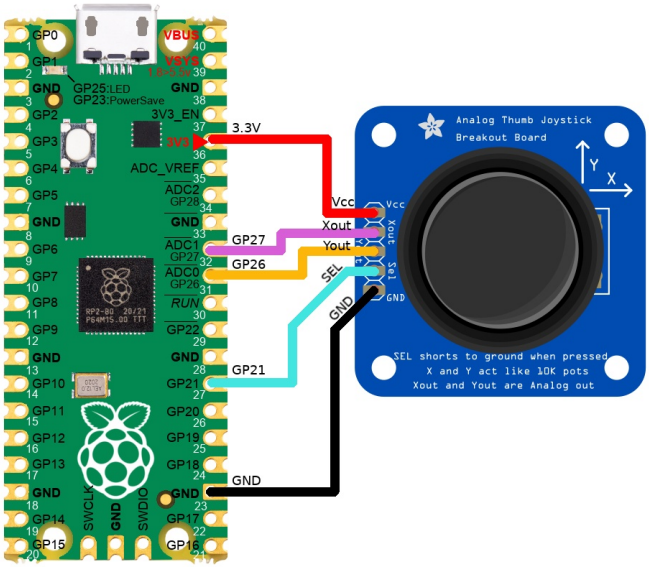
Le potentiomètre est un dispositif équipé d'un curseur se déplaçant au dessus du corps d'une résistance mis à nu (ou d'un corps résistif). Ce dispositif permet de produire une une tension de sortie proportionnelle à la position du curseur.



08RI46 – Fonctionnement interne du potentiomètre

Au repos, le joystick 2 axes est en position centrale et expose donc une tension de 1,65 V (la moitié de 3,3V) aussi bien sur la sortie Xout que Yout.

Voici une proposition de raccordement sur les broches GP26 (ADC0) pour l'axe Y et et GP27 (ADC1) pour l'axe X. Le bouton poussoir du joystick est branché sur le GP21.



08RI47 – brancher le joystick 2 axes

Le code suivant permet de lire l'état des entrées analogiques.

```
>>> from machine import Pin,ADC
>>> xpos = ADC(Pin(27))
>>> ypos = ADC(Pin(26))
>>> xpos.read_u16()
65535
>>> xpos.read_u16()
34504
>>> xpos.read_u16()
272
>>> ypos.read_u16()
65535
>>> ypos.read_u16()
32695
>>> ypos.read_u16()
272
```

L'instruction `ADC(Pin(27))` permet d'obtenir une référence vers le convertisseur analogique correspondant à la broche GPIO 27. ADC est l'acronyme de *Analog to Digital Converter*.

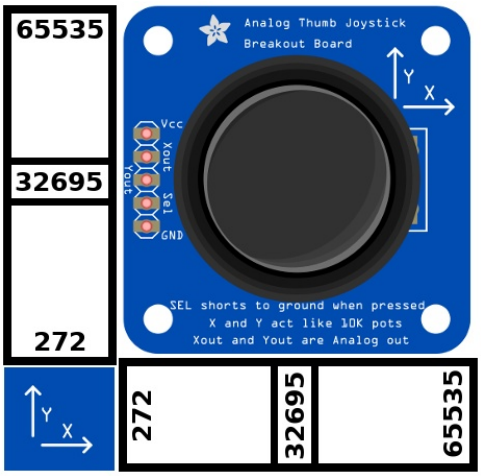
La classe ADC propose la méthode `read_u16()` qui retourne une valeur entre 0 (pour 0 volts) et 65535 (pour 3,3V).

Comme le sous-entend `read_u16()`, la valeur retournée est un *unsigned integer 16 bits* (un entier non signé 16 bits). La valeur maximale que peut contenir un entier 16 bits non signés est 65535, ce qui correspond à `0b11111111_11111111` .

Avec le joystick en position centrale (au repos), la méthode `read_u16()` retourne une valeur proche de 33000 (soit proche de 65535/2).

Avec le joystick en position minimale, la valeur est proche de 0.

Enfin, Le joystick position maximale retourne une valeur proche de 65535.



08RI48 – Minimas et maximas du joystick analogique

Bouton du joystick

Un bouton poussoir est présent sous le joystick. La broche Sel est branchée à la masse lorsque le bouton est pressé.

Pour lire l'état du bouton, la broche sera configurée en entrée en activant la résistance pull-up interne. La broche restera au niveau haut sauf si un événement extérieur (comme le bouton pressé) force le potentiel à la masse.

L'entrée sera donc au niveau haut (`True`) quand le bouton n'est pas enfoncé et passe au niveau bas (`False`) lorsque le bouton est pressé. Cette logique « inverse » peut être ré-inversée à l'aide de la classe `Signal`.

La lecture de l'état du bouton se fait donc comme suit :

```
>>> from machine import Pin, Signal
>>> btn = Signal( Pin(21, Pin.IN, Pin.PULL_UP), invert=True )
>>> btn.value() # Bouton relaché
0
>>> btn.value() # Bouton pressé
1
>>>
```

Grâce à la classe Signal, l'instance btn retourne une valeur vraie (`True`) lorsque le bouton est pressé et fausse (`False`) lorsque le bouton est relâché.

Messages via RFM69

La télécommande disposant de 60 octets pour envoyer chaque message, l'information sera envoyée en claire (sous forme de chaîne de caractères) au format suivant :

`<valeur_xout> , <valeur_yout> , <valeur_bouton>`

dont voici deux exemples :

```
"12587,65535,1"
"33254,1210,0"
```

Cette approche permettra de préserver une meilleure compréhension du script et des messages au prix d'un gaspillage de la bande passante.

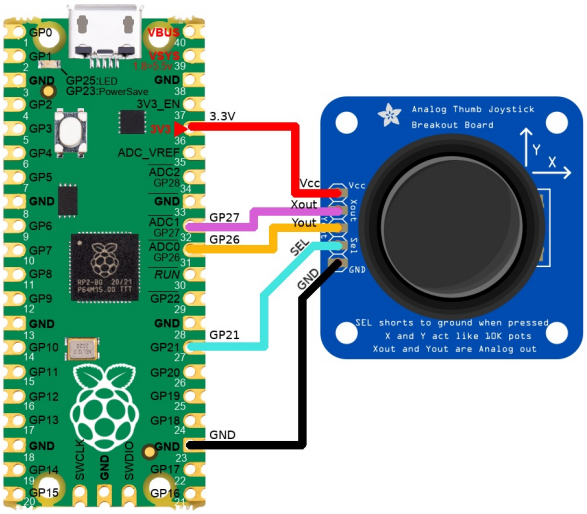
En effet, transmettre la chaîne de caractères "12587,65535,1" (soit dit en passant que c'est la plus longue) représente la transmission de 13 octets !

L'approche professionnelle, écartée dans le cas présent, utiliserait le format binaire de stockant l'entier non signé. Il faudrait alors deux octets (16 bits) pour X, deux octets pour Y et un octet pour transmettre l'état du bouton. Un total de 5 octets seraient alors suffisant.

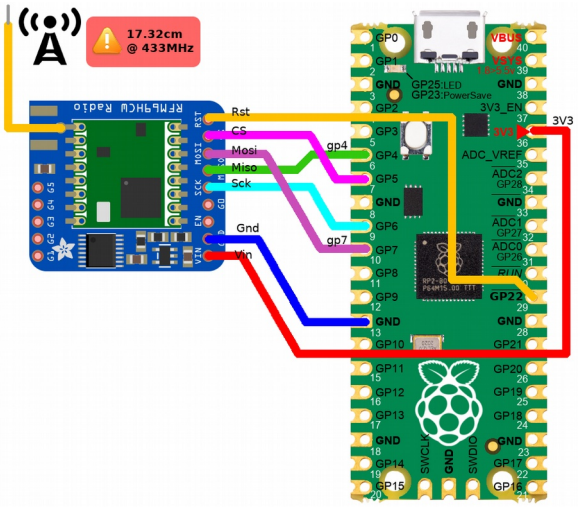
Montage télécommande RFM

La télécommande est réalisée avec un Raspberry-Pi Pico, le module RFM69 et le joystick analogique.

Le montage reprend l'assemblage du module RFM69 tel que déjà présenté et le montage du joystick analogique tels que présenté ci-dessus :



08RI47 – brancher le joystick 2 axes



03RI35 – Raccordement d'un RFM69 sur un Pico

Script télécommande RFM

Le script `rfm_rc_joy.py` transformera le montage en télécommande.

Le script est également disponible dans les dépôt du projet au lien suivant :

- https://github.com/mchobby/micropython-zumo-robot/blob/main/extras/RFM69/rfm_rc_joy.py

L'état du joystick et du bouton sera communiqué à intervalle de 1/10 sec (100ms). Aucun accusé de réception n'est attendu par cet émetteur pour éviter le temps de latence que cela représente.

```
01: from machine import SPI, Pin, ADC, Signal
02: from rfm69 import RFM69
03: import time
04:
05: FREQ           = 433.1
06: ENCRYPTION_KEY = b"\x01\x02\x03\x04\x05\x06\x07\x08\x01\x02\
x03\x04\x05\x06\x07\x08"
07: NODE_ID        = 120
08: RECEIVER_ID    = 100
09:
10: # Joystick interface
11: xpos = ADC(Pin(27))
12: ypos = ADC(Pin(26))
13: btn = Signal( Pin(21, Pin.IN, Pin.PULL_UP), invert=True )
14:
15: # RFM69 interface
16: spi = SPI(0, miso=Pin(4), mosi=Pin(7), sck=Pin(6),
                baudrate=50000, polarity=0, phase=0,
                firstbit=SPI.MSB)
17: nss = Pin( 5, Pin.OUT, value=True )
18: rst = Pin( 22, Pin.OUT, value=False )
19:
20: rfm = RFM69( spi=spi, nss=nss, reset=rst )
21: rfm.frequency_mhz = FREQ
22: rfm.encryption_key = ( ENCRYPTION_KEY )
23: rfm.node = NODE_ID
24: rfm.destination = RECEIVER_ID
25: while True:
26:   msg = "%s,%s,%s" % ( xpos.read_u16(), ypos.read_u16(),
                           btn.value() )
27:   rfm.send(  msg.encode("ASCII") )
28:   # print( "send", msg )
29:   time.sleep(0.1)
```

Les données envoyées par le script peuvent être visualisés à l'aide du script `rfm_rc_joy_rx.py` (disponible dans le dépôt aux côtés du script émetteur, fort utile pour faire du débogage).

•Lignes 1 à 3 : import des classes et module nécessaires. Les classes `ADC` et `Signal` seront utilisées avec le joystick analogique.

•Lignes 5 à 8 : définition des paramètres de l'émetteur RFM69. La télécommande portera le numéro 120 (constante `NODE_ID`) et le récepteur sur le Zumo Robot portera le numéro 100 (comme l'indique la constante `RECEIVER_ID`).

•Ligne 11 : déclaration de `xpos`, instance de l'objet `ADC` (convertisseur analogique/numérique) permettant de lire la tension sur la broche GPIO 27.

•Ligne 12 : déclaration de ypos pour acquérir la tension analogique sur la broche GPIO26.

•Ligne 13 : déclaration d'une broche en entrée (`Pin.IN`) avec activation de la résistance de rappel à 3,3V (`Pin.PULL_UP`). La tension sur l'entrée est donc de 3,3V sauf si le bouton est pressé (cas où la tension est ramenée à 0 Volts). L'instance de la broche ainsi créée est immédiatement passé en paramètre à la classe `Signal` qui en inversera l'état (pour avoir un 1 lorsque le bouton est pressé). L'instance de la classe `Signal` est accessible via la variable `btn`.

•Lignes 16 à 23 : création de l'instance du bus SPI et du module RFM69 avec initialisation des différents paramètres de communication.

•Ligne 24 : permet d'indiquer le numéro de nœud du récepteur destinataire des messages (celui présent sur le Zumo Robot, il porte le numéro 100).

•Ligne 25 : début de la boucle infinie `while True` exécutant les tâches du corps du programme. Ces tâches sont : (1) acquisition de la position du joystick analogique, (2) formatage des données (3) transmission des données.

•Ligne 26 : préparation du message à transmettre. Le message est formaté à l'aide de la syntaxe `"%s,%s,%s" % ( 111, 222, 333 )` qui produit, dans le cas présent, la chaîne de caractères `"111,222,333"` en opérant les substitutions de `%s`. 111, 222 et 333 sont respectivement remplacés par (1) la valeur du convertisseur analogique sur l'axe X, (2) par la valeur analogique sur l'axe Y et enfin (3) par l'état du bouton (1 ou 0). Pour rappel, la méthode `ADC.read_u16()` retourne une valeur numérique entre 0 et 65535. Pour finir, le message ainsi composé est stocké dans la variable `msg`.

•Ligne 27 : transformation du message en données binaires (type `bytes`) en encodant la chaîne de caractères en ASCII. Le message encodé avec `msg.encode("ASCII")` est alors envoyé au Robot Zumo avec `rfm.send()` .

Dans le cadre d'usage type « télécommande », la réception d'un accusé de réception depuis le récepteur (le Zumo Robot) est sans aucune importance. Par conséquent, autant éviter de surcharger les échanges avec une fonctionnalité non pertinente, raison de l'appel de RFM69,send() sans paramètre with_ack=True.

•Ligne 28 : ligne en commentaire. En retirant le caractère « # » en tête de ligne, cette ligne de code affiche les messages expédiés par radio (texte affiché dans la session REPL).

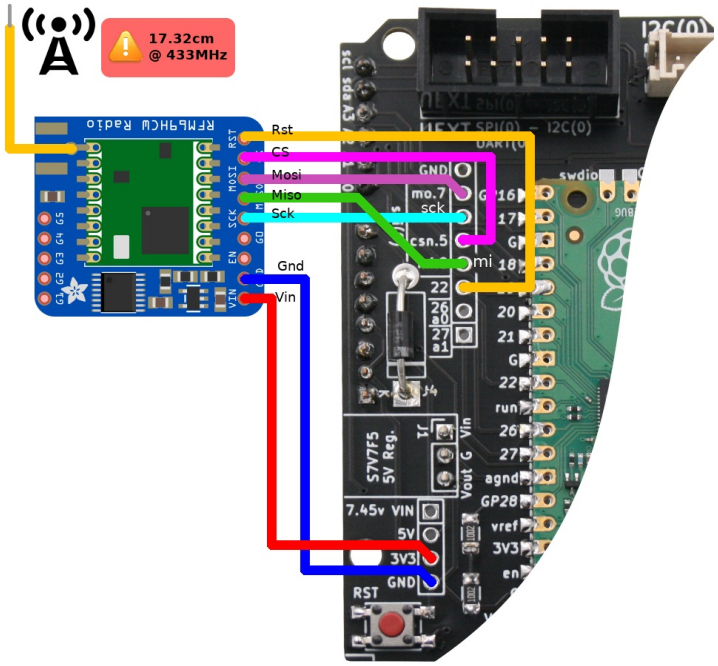
•Ligne 29 : pause de 1/10 de seconde (100 millisecondes) pour ne pas surcharger le canal de communication.

2.4.2.Le Robot Zumo

Montage du RFM sur le Zumo

Dans un montage à la mode « *Do It Yourself* », le module RFM69 se branche exactement comme sur le module émetteur (GPIOs 4,5,6,7 et 22).

Si le lecteur dispose de l'adaptateur Pico-Zumo, les GPIOs mentionnés ci-dessus restent disponibles. Les raccordements peuvent être réalisés comme suit :



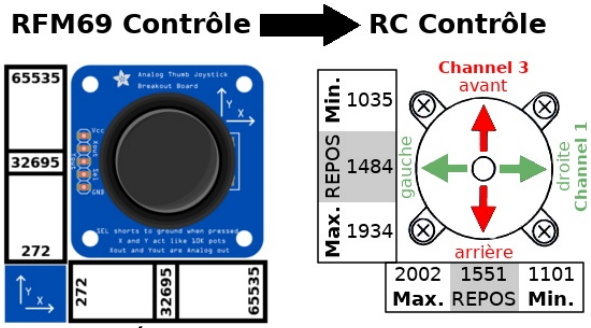
08RI51 – connexion du module RFM69 sur le Pico Zumo

RFM69HCW	PICO-Zumo
RST	GP22
CS	CSN.5
MOSI	MO.7
MISO	MI.4
SCK	SCK.6
GND	GND
VIN	3V3

Script RFM Zumo

L'idée est de réutiliser le script du robot contrôlé par la télécommande RC avec un seul joystick et de remplacer la réception RC (`time_pulse_us`) par la réception des messages réceptionnés via le RFM69 (`<valeur_xout>, <valeur_yout>, <valeur_bouton>`).

En récupération les informations x, y et bouton des messages, ces derniers peuvent être transformés en valeurs `us_speed`, `us_dir` équivalents. De la sortie, cela évite de ré-écrire un nouveau script.



08RI50 – Équivalence données RFM et implusion RC

La conversion se fait à l'aide de la fonction `map()` déjà présentée à plusieurs reprises dans cet ouvrage.

Un appuis sur le bouton du joystick provoquera un arrêt du robot Zumo (en simulant une erreur de communication)

Le script `rfm_rc_zumo.py` permet de contrôler le Zumo Robot depuis un RFM69. Il faut, bien entendu utiliser la télécommande RFM précédemment créée pour contrôler le robot à distance.

Le script `rfm_rc_zumo.py` est également disponible dans le dépôt du projet :

- https://github.com/mchobby/micropython-zumo-robot/blob/main/extras/RFM69/rfm_rc_zumo.py

```
01: from zumoshield import *
02: from machine import Pin, SPI
03: from rfm69 import RFM69
04: import time
05:
06: FREQ           = 433.1
07: ENCRYPTION_KEY = b"\x01\x02\x03\x04\x05\x06\x07\x08\x01\x02\
x03\x04\x05\x06\x07\x08"
08: NODE_ID        = 100
09:
10: # RFM Interface
11: spi = SPI(0, miso=Pin(4), mosi=Pin(7), sck=Pin(6),
              polarity=0, phase=0, firstbit=SPI.MSB)
12: nss = Pin( 5, Pin.OUT, value=True )
13: rst = Pin( 22, Pin.OUT, value=False )
14:
15: rfm = RFM69( spi=spi, nss=nss, reset=rst )
16: rfm.frequency_mhz = FREQ
17: rfm.encryption_key = ( ENCRYPTION_KEY )
18: rfm.node = NODE_ID
19:
20: SPEED_CENTER_US = 1489
21: DIR_CENTER_US = 1551
22: RANGE_US = 400
23:
24: def map(value, istart, istop, ostart, ostop):
25:   return ostart + (ostop - ostart) * ((value - istart) /
            (istop - istart))
26:
27: # Joystick information
28: joy_xpos = 65535//2
29: joy_ypos = 65535//2
30: joy_btn = 0
31:
```

```
32: z = ZumoShield()
33: z.play_blip()
34: while True:
35:
36:   packet = rfm.receive()
37:   if packet is None:
38:     continue
39:   else:
40:     packet_text = str(packet, "ascii")
41:    _l = packet_text.split(",")
42:    if len(_l)!=3:
43:      z.led.toggle()
44:      continue
45:    try:
46:      joy_xpos = int(_l[0])
47:      joy_ypos = int(_l[1])
48:      joy_btn = int(_l[2])
49:    except:
50:      z.led.toggle()
51:
52:  # Debug: print( joy_xpos, joy_ypos, joy_btn )
53:
54:  # us_speed  = time_pulse_us( ch_speed, 1 )
55:  # us_dir = time_pulse_us( ch_dir, 1 )
56:  us_speed = map( joy_ypos, 0, 65535,
                SPEED_CENTER_US+RANGE_US,
                SPEED_CENTER_US-RANGE_US )
57:  us_dir = map( joy_xpos, 0, 65535,
                DIR_CENTER_US+RANGE_US,
                DIR_CENTER_US-RANGE_US)
58:
59:  if joy_btn==1:
60:    us_speed = -1
61:    us_dir = -1
62:
63:  # Original code for the RC time_pulse_us based control
64:  if (us_speed < 0) or (us_dir < 0):
65:    z.motors.stop()
66:    z.led.toggle()
67:    # no need to sleep, time_pulse_us timeout
68:    # will insert the required delay in the execution.
69:    continue
70:
71:  # uSec --> -400 to +400
72:  speed = map( us_speed, SPEED_CENTER_US-RANGE_US,
                 SPEED_CENTER_US+RANGE_US, +300, -300 )
73:  dir_diff = map( us_dir, DIR_CENTER_US+RANGE_US,
                    DIR_CENTER_US-RANGE_US, -100, +100 )
74:
75:  # print( speed, dir_diff )
76:  if abs(speed)<=25:
77:    speed = 0
78:  if abs(dir_diff)<=10:
79:    dir_diff = 0
80:  if speed<=-25:
81:    dir_diff *= -1
82:  elif speed==0:
83:    dir_diff *= 2
84:  speed_left = int(speed + dir_diff)
85:  speed_right = int(speed - dir_diff)
86:  if speed_left<-400:
87:    speed_left = -400
88:  if speed_left>400:
89:    speed_left = 400
90:  if speed_right<-400:
91:    speed_right=-400
92:  if speed_right>400:
```

```
93:    speed_right=400
94:
95:  z.motors.setSpeeds( speed_left, speed_right )
96:  time.sleep_ms(100)
```

Voici quelques détails utiles concernant le script :

•Lignes 1 à 4 : import des bibliothèques nécessaires.

•Lignes 6 à 18 : création des constantes et instances relatif au module RFM69 (voir précédentes description). Le Zumo Robot porte le numéro de nœud 100. Il agira principalement comme récepteur.

•Lignes 20 à 21 : constantes reprises depuis l'implémentation de la télécommande RadioControle. Il s'agit de la longueur d'impulsion (en microsecondes) des joysticks au repos (en position centrale) pour la vitesse avec `SPEED_CENTER_US` et la direction `DIR_CENTER_US`. De ces positions centrales, le signal d'impulsion varie de -400 µSec à +400µSec (`RANGE_US`).

Les constantes RC sont préservées dans ce script car le corps du script de contrôle du reste identique puisque parfaitement fonctionnel. Les données reçues par la commande RFM69 seront convertie en leurs équivalents microsecondes pour contrôler le Zumo Robot.

•Lignes 24 et 25 : définition de la fonction d'interpolation linéaire `map()` déjà abordée à plusieurs reprise dans cet ouvrage.

•Lignes 28 et 29 : définition des variables `joy_xpos` et `joy_ypos` destinées à stocker les valeurs x et y communiquées par le module RFM69. Initialisation à la valeur de repos du joystick, donc à la moitié de la valeur maximale. L'utilisation de l'opérateur « `//` » permet de faire une division entière (sans décimale).

•Ligne 30 : définition de la variable `joy_btn` dont la valeur est également communiquée par le module RFM69. La variable est initialisée à 0 (bouton non pressé).

•Ligne 32 à 34 : création de l'instance du ZumoShield + signal sonore indiquant que le robot est prêt à être piloté. La ligne 34 début de la boucle infinie (lignes 35 à 96) prenant en charge le contrôle du Zumo.

•Ligne 36 : réception d'un paquet de données depuis l'émetteur (appel non bloquant).

•Lignes 37 et 38 : si aucun paquet réceptionné alors l'instruction `continue` redémarre la boucle `while` à la ligne 35.

•Lignes 40 à 50 : dans le cas contraire, un paquet est réceptionné et les lignes suivantes sont consacrées au décodage des 3 paramètres `xpos`, `ypos` et `btn` normalement inclus dans ce paquet.

•Ligne 40 : décodage du paquet binaire en chaîne de caractère. A ce stade, la variable `packet_text` est supposé contenir une chaîne de caractères similaire à `"23857,52017,0"`.

•Ligne 41 : la méthode `split(",")` permet de découper la chaîne de caractères sur la virgule. La variable `_l` contient le résultat de la découpe sous forme de liste. Pour l'exemple ci-dessus, cela produira la liste suivante `["23857", "52017", "0"]` contenant toujours des chaînes de caractères.

•Ligne 42 à 44 : Si le paquet contient bien les données attendue alors la liste doit avoir contenir exactement 3 éléments (`len(_l)` doit être égal à 3). Si ce n'est pas le cas alors l'état de la LED du Zumo Robot est inversé avec la méthode `toggle()` puis la boucle `while` redémarrée avec l'instruction `continue` .

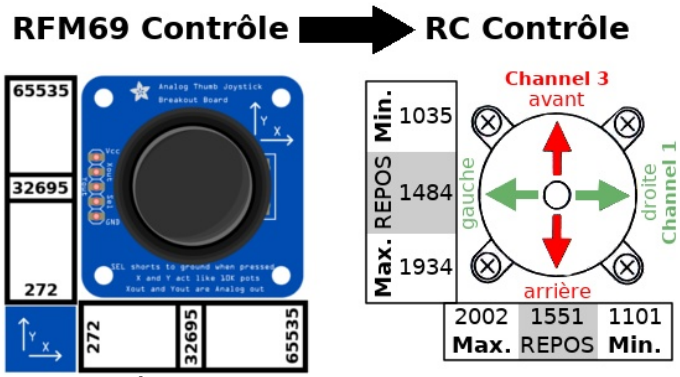
•Lignes 45 à 50 : L'instruction `int(_l[0])` prend la première chaînes de caractères de la liste (`"23857"`) et la transforme en entier qui est stocké ensuite dans la variable `joy_xpos`. Les lignes 47 et 48 utilisent le même procédé pour les variables `joy_ypos` et `joy_btn` . La structure `try...except` permet de capturer les erreurs intervenant durant la conversion en entier (ex : l'une des chaînes de caractères contient du texte). En cas d'erreur entre le `try` et `except` alors le contenu de la section `except` est exécuté. La section except fait clignoter la LED du Zumo Robot mais ne redémarre pas la boucle `while`. En effet, les dernières données connues de `joy_xpos`, `joy_ypos` et `joy_btn` sont probablement encore d'application.

•Ligne 52 : ligne en commentaire. Retirer le caractère « # » en tête de ligne pour réactiver celle-ci et afficher les données `joy_xpos`, `joy_ypos` et `joy_btn` réceptionnées dans la session REPL.

•Lignes 54 et 55 : lignes en commentaire. Ces dernières permettaient d'obtenir les longueur d'impulsion `us_speed` et `us_dir` reçus par un récepteur RC (Radio Commande de modélisme).

<u>Un petit jeu de conversion</u>

A partir de maintenant, le but du jeu est de convertir `joy_ypos` vers `us_speed` et convertir `joy_xpos` vers `us_dir`.



08RI50 – Équivalence données RFM et implusion RC

Dans la version contrôle RC, la constante `RANGE_US=400` était utilisée couvrir une gamme de 1484-400 à 1484+400 microsecondes pour le contrôle de vitesse. La valeur centrale du joystick (au repos) de 1484 microsecondes correspond à la constante `SPEED_CENTER_US` déclarée en début de script.

Ainsi, la gamme de valeur RC est de `SPEED_CENTER_US-RANGE_US` à `SPEED_CENTER_US+RANGE_US` microsecondes.

Il faudra mettre cette gamme microsecondes `SPEED_CENTER_US-RANGE_US` à `SPEED_CENTER_US+RANGE_US` avec la gamme de la commande RFM69 allant de 272 à 35532.

Reprise des explications

•Ligne 56 : utilisation de la fonction `map()` pour convertir la valeur de `joy_ypos` de la gamme 0 à 65535 vers la gamme `SPEED_CENTER_US-RANGE_US` à

`SPEED_CENTER_US+RANGE_US`. La variable `us_speed` contient cette nouvelle valeur.

•Ligne 57 : utilisation de la fonction `map()` pour convertir la valeur de `joy_xpos` de la gamme 0 à 65535 vers la gamme `DIR_CENTER_US-RANGE_US` à `DIR_CENTER_US+RANGE_US`. La variable `us_dir` contient cette nouvelle valeur.

•Lignes 59 à 61 : si le bouton est pressé sur la commande RFM69 alors le robot doit s'arrête. Dans le projet de la commande RC, la fonction `time_pulse_us()` retourne la valeur -1 lorsque la communication avec l'émetteur RC est perdue. Dans pareil cas, le robot s'arrêtait. Ainsi donc, lorsque le bouton de la commande RFM69 est pressé, il suffit de remplacer la valeur de `us_speed` et de `us_dir` par -1.

•Lignes 64 à 96 : code identique à celui utilisé dans le projet de commande RC. Les détails du fonctionnement de ces lignes y est décrit.

2.5.REPL à distance

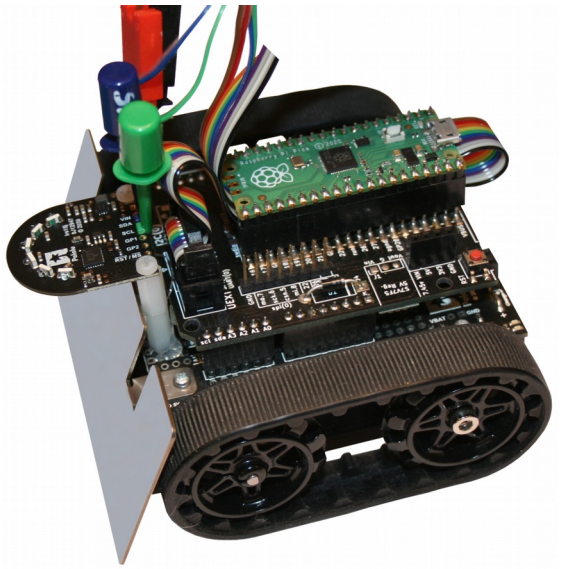
Xxx

2.6.Interface Radio/série

Xxx

3.Capteur de distance (OPT3101)

Cette section du chapitre va se concentrer sur l'utilisation d'un capteur de distance permettant au Robot Zumo d'anticiper certaines situations particulières comme l'évitement d'objet, l'arrêt préventif, etc



08RI53 – Capteur de distance OPT3101 à l'avant du Robot Zumo

En utilisant quelques supports en plastique/nylon et un peu de colle chaude, il est possible d'équiper le Robot Zumo avec un où plusieurs capteurs de distance.

3.1.Les technologies de mesure

Xxx ultrasonique, Vlx, Capteur infrarouges

3.2.Le capteur OPT3101

Xxx spécial dans son domaine

3.3.Tester l'OPT3101

3.3.1.Détails sur le module

Broches

3.3.2.Brancher sur le Pico

Xxx
tableau des broches

3.3.3.Brancher sur le Robot Zumo

Xxx via connecteur UEXT
xxx via connecteur QWIIC

3.3.4.Tester le capteur

Xxxxxxxxx
liens vers le script dans extras/opt3101
description ligne par ligne

3.4.Suivre un objet

ssss

3.5.Freinage automatique

sssssss

3.6. Évitement automatique

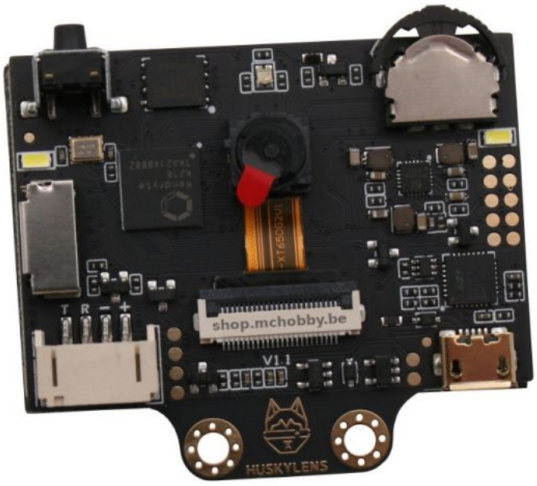
Garder un objet à distance minimale durant le déplacement
*** essayer de passer dans un tunnel ***

4.HuskyLens : Camera à Intelligence artificielle

Cette section du chapitre va se concentrer sur l'utilisation d'une caméra épaulée par intelligence artificielle.

Les capacités de la caméra seront utilisées pour réaliser un suivit de ligne intelligent.

4.1.HuskyLens

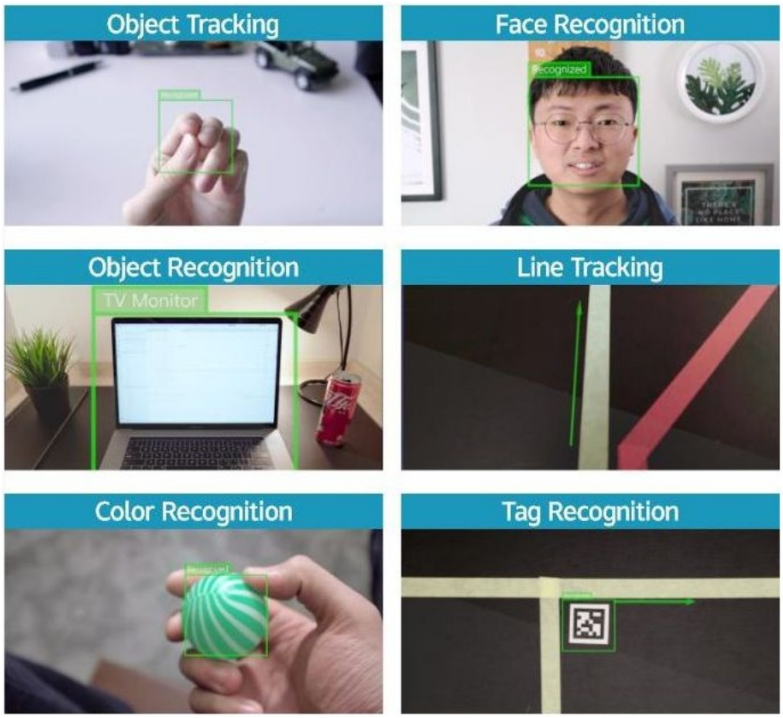


08RI68 – HuskyLens

HuskyLens est un module de vision intelligent équipé d'un écran (au verso) pour visualiser les éléments identifiés par la caméra et les algorithmes d'intelligence artificielle. Le processeur Kendryte K210 est spécialement conçu pour le traitement de tâches en intelligence artificielle.

Les algorithmes supportés par HuskyLens sont :

- reconnaissance faciale (*Face Recognition*),
- suivit d'objet (*Object Tracking*),
- identification d'objet (*Object Recognition*),
- suivit de ligne (*Line Tracking*),
- identification de couleur (*Color Recognition*),
- identification de tag/QR (*Tag Recognition*)

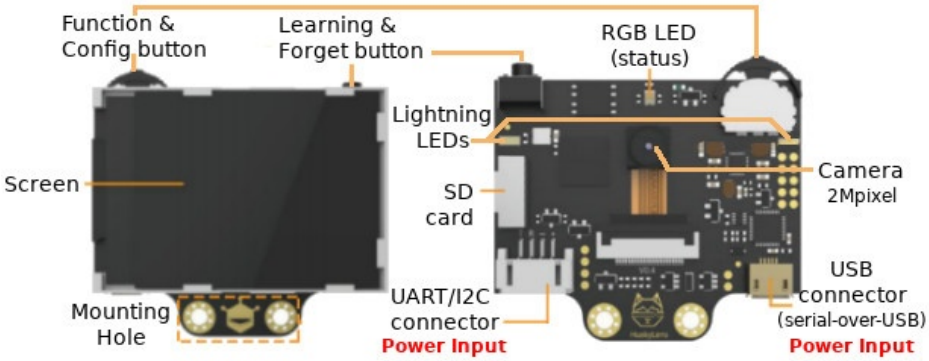


08RI69b – mise en application des algorithmes

La force de HuskyLens réside dans son interface simplissime permettant (1) de sélectionner l'algorithme souhaité et (2) d'effectuer l'apprentissage requis.

Grâce à son écran, le HuskyLens est un périphérique totalement autonome. L'image capturée par la caméra 2 mégapixel est reproduite sur l'écran pour faciliter le cadrage et la sur-impression d'information informe l'utilisateur des éléments identifiés.

Alimentez le HuskyLens via le port USB et le périphérique est prêt à l'emploi.



08RI69a – Fonctions du HuskyLens

L'autre grand intérêt du HuskyLens est son port de sortie pouvant être configuré en I2C ou UART (port série). Grâce à ce port de sortie, un microcontrôleur ou un ordinateur peut être notifié des éléments identifiés par le HuskyLens.

Voir le lien suivant pour plus d'informations sur HuskyLens :

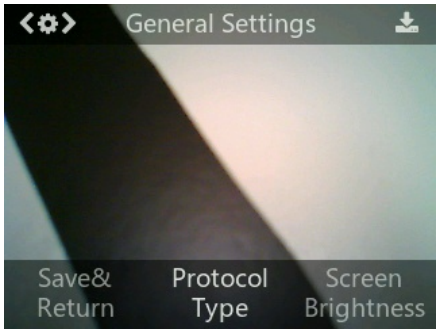
- https://shop.mchobby.be/product.php?id_product=2421

4.1.1.Configuration du port

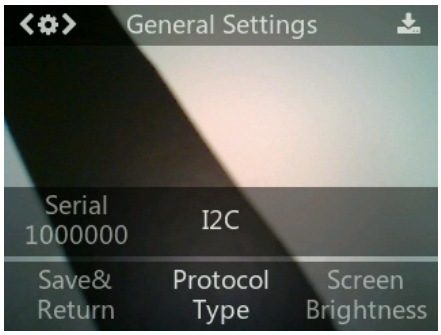
Une fois le HuskyLens mis sous-tension, la première opération à réaliser est la configuration du port de sortie en I2C pour pouvoir le brancher sur le bus I2C du Pico-Zumo.

Presser la molette de configuration pour afficher le menu sur l'écran du HuskyLens puis naviguer jusqu'à l'entrée « *General Settings* » puis « Protocol Type » et enfin .

08RI70a – Sélection des paramètres généraux

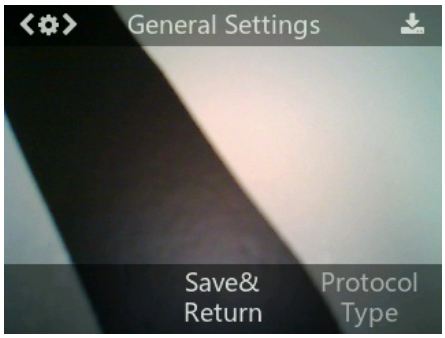


08RI70b – Sélection du protocole pour le port de sortie



08RI70c – Sélection du protocole I2C

Enfin, sauver et activer la configuration en sélectionnant « *Save & Return* ».



08RI70d – Sauvegarder les modifications

A partir de maintenant, il sera possible d'interroger le HuskyLens via le bus I2C du Raspberry-Pi Pico.

4.1.2.Configurer le suivit de ligne

Dans le cas d'un projet de suivit de ligne il convient de sélectionner l'algorithme « *Line Tracking* » et effectuer l'apprentissage de la ligne à suivre.

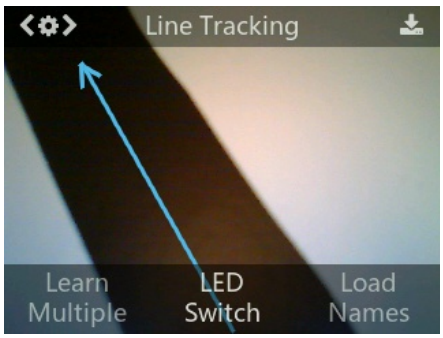
Presser la molette de configuration pour afficher le menu sur l'écran du HuskyLens puis naviguer jusqu'à l'entrée « *Line Tracking* ».

08RI71a – Sélection de l'algorithme de suivit de ligne

Une fois l'algorithme sélectionné, le menu affiche plusieurs options en relation avec le suivit de ligne.

Le HuskyLens dispose de deux LEDs en lumière blanche de part et d'autre de la caméra. Il est conseillé de les allumer pour offrir une image avec un meilleure contrast à l'algorithme de reconnaissance de ligne.

Sélectionner l'entrée « *LED Switch* » pour activer les LEDs.



08RI71b – Option des LEDs d'éclairage

08RI71c – Activer/désactiver les LEDs d'éclairage

L'option « Learn Multiple » est une option activable permettant au HuskyLens de conduire plusieurs cycles d'apprentissages consécutifs de la ligne à suivre. Si cette option est activée, il faudra revenir dans le menu pour désactiver l'option une fois que l'apprentissage offrira un résultat satisfaisant.

Pour un parcours en ligne noir sur arrière plan blanc, un seul apprentissage est suffisant. L'option « Learn Multiple » n'est donc pas nécessaire dans pareil cas.

4.1.3.Apprentissage de la ligne

HuskyLens est capable d'utiliser la couleur de la ligne comme élément discriminatoire (à condition qu'elle soit bien contrastée avec l'arrière plan). Les différentes couleurs de lignes doivent également être bien contrastées entre elles.

La ligne doit, bien entendu, être d'une seule couleur (monochrome) uniforme.

Il est recommandé de débuter l'expérimentation avec un circuit exclusivement monochrome.

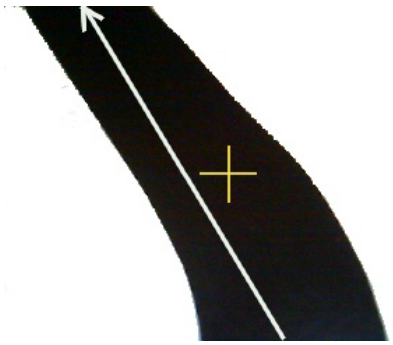
Durant l'apprentissage, la ligne située sous la caméra doit centrée verticalement sur l'écran allant du bas de celui-ci vers le haut de celui-ci. Cette disposition facilite l'apprentissage.

L'éclairage ambiant peut avoir une énorme influence sur la perception des couleurs et sur le contraste ligne/arrière-plan. Il est donc important d'avoir un éclairage aussi uniforme que possible.

Presser le bouton d'apprentissage (*Learning)* pour débuter l'apprentissage de la ligne. Si HuskyLens dispose déjà d'un apprentissage, il est possible d'annuler ce dernier en pressant une seconde fois sur le bouton d'apprentissage lorsque le message.

08RI72 – Oublier le précédent apprentissage

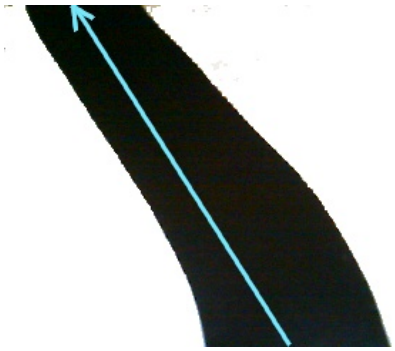
Durant l'apprentissage, un curseur au centre de l'écran permet de positionner la caméra au dessus de la ligne à suivre. Si l'algorithme détecte une ligne, il affiche une flèche blanche identifiant la position et la direction de celle-ci. La **ligne blanche indique qu'il s'agit d'une suggestion** et non le résultat de l'apprentissage !



08RI73 – Sélection de la ligne durant l'apprentissage

Presser le bouton d'apprentissage (*Learning*) pour identifier cet élément comme la ligne à suivre.

L'apprentissage fait, la croix disparaît et la direction à suivre est représenté par une flèche bleue. **Une ligne bleue indique que la flèche est le résultat issu de l'apprentissage**.



08RI74 – résultat de l'apprentissage (flèche bleue)

Une fois l'apprentissage terminé, balader la caméra en différents points du circuit permet de confirmer que l'apprentissage permet une détection fiable de la ligne en differents points.

Plusieurs options s'offre à l'utilisateur la détection de la ligne n'est pas fiable :

1.Vérifier l'homogénéisation de la couleur et épaisseur de la ligne.

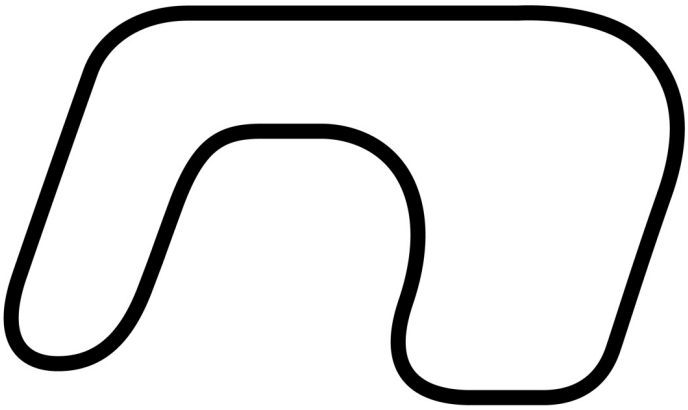
2.Vérifier les conditions de luminosité ambiante

3.Allumer les LEDs d'éclairage (de la caméra)

4.Réaliser un apprentissage multiple en activant l'option « *Learn Multiple* » dans le menu de suivit de ligne («*line tracking* »).

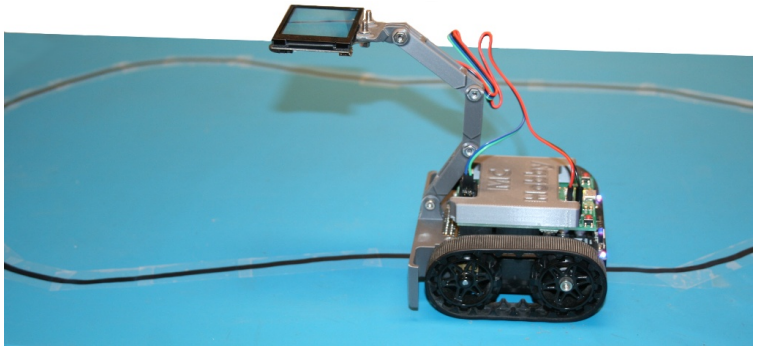
4.1.4.Circuit de test

Comme pour le suivit de ligne à l'aide du capteur de ligne du Zumo, il est possible de réaliser un circuit à l'aide d'une bande adhésive noir (et mat) de 8 à 20mm.



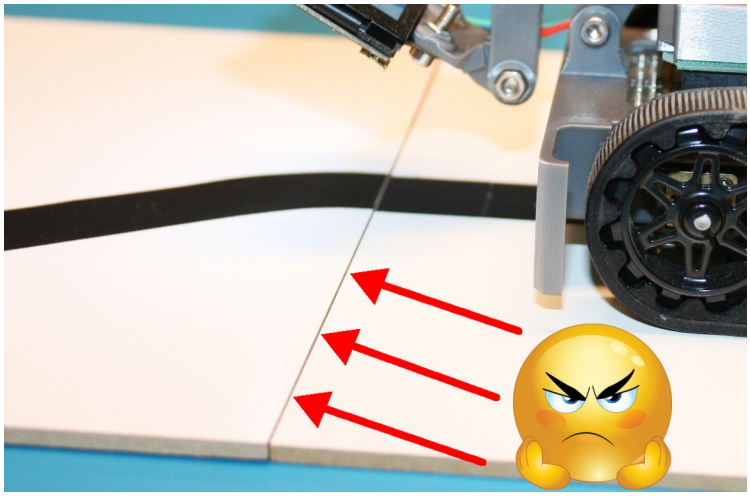
08RI75 – Exemple de circuit de test

Il est également possible de coller du papier adhésif (ex : papier collant transparent 3M) et finaliser le tracé du circuit avec marqueur indélébile épais (6mm dans la capture ci-dessous).



08RI94 – Circuit réalisé à l'aide d'un marqueur indélébile de 6mm.

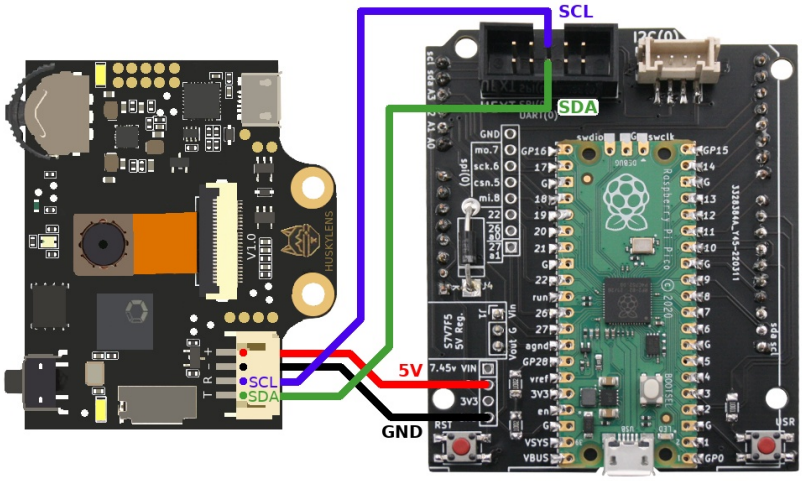
Il convient par contre d'éviter l'usage de tuiles pour réaliser un circuit car les interstices entre les tuiles capturent la lumière et sont identifiés comme des lignes.



04RI94b – Les interstices sont identifiés comme des lignes

4.2.Brancher

Le HuskyLens est branché sur la carte Pico-Zumo comme suit :



08RI90 – Brancher HuskyLens sur le Pico-Zumo

Il est impératif d'alimenter le HuskyLens en 5V afin d'obtenir une communication stable sur le bus I2C, même si le bus I2C garde des signaux en logique 3,3V.

En procédant à ce raccordement, le signal SDA du HuskyLens est branché sur GP8 (=SDA) tandis que signal SCL est branché sur GP7 (=SCL).

Le HuskyLens est donc branché sur le port I2C(0) du Pico.

4.3.Installer la bibliothèque

La bibliothèque `husky.py` est disponible dans le dépôt Github esp8266-upy :

https://github.com/mchobby/esp8266-upy/tree/master/huskylens

Le bibliothèque est constituée d'un seul fichier `husky.py` qu'il faut copier sur le Pico à l'aide de votre outil préféré.

Il est aussi possible d'utiliser l'utilitaire **mpremote** pour télécharger et installer automatiquement la bibliothèque sur le pico.

```
dodo:~$ mpremote mip install github:mchobby/esp8266-upy/huskylens
Install github:mchobby/esp8266-upy/huskylens
Installing github:mchobby/esp8266-upy/huskylens/package.json to
           /lib
Installing: /lib/husky.py
Done
```

4.4.Tester

Le plus important à ce stade est de tester la communication entre le Pico et le HuskyLens et s'assurer qu'ils peuvent dialoguer ensembles.

Le plus simple est d'utiliser le script `simple.py` disponible dans les exemples du dépôt.

https://github.com/mchobby/esp8266-upy/blob/master/huskylens/examples/simple.py

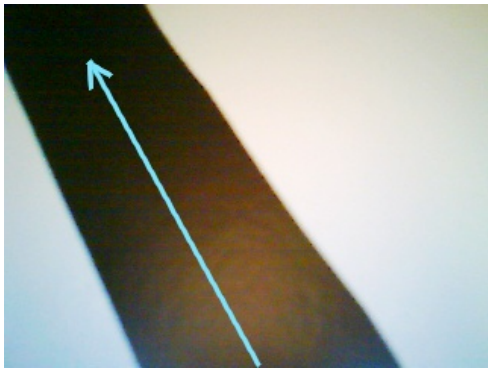
Il est nécessaire de s'assurer que le port I2C utilisé par le script est bien celui sur-lequel est branché le HuskyLens. Le port I2C doit être déclaré comme suit dans le script de test.

```
i2c = I2C( 0, sda=Pin(8), scl=Pin(9), freq=100000 )
```

Une fois le script copié sur le Pico, il est possible de l'exécuter à la volée en saisissant la commande `import simple` depuis une invite REPL.

Le script récupère les objets identifiés par HuskyLens par l'intermédiaire de la liaison I2C et affiche les informations dans la session REPL.

La capture HuskyLens ci-dessous :



08RI100 – Identification d'une direction (flèche) par HuskyLens

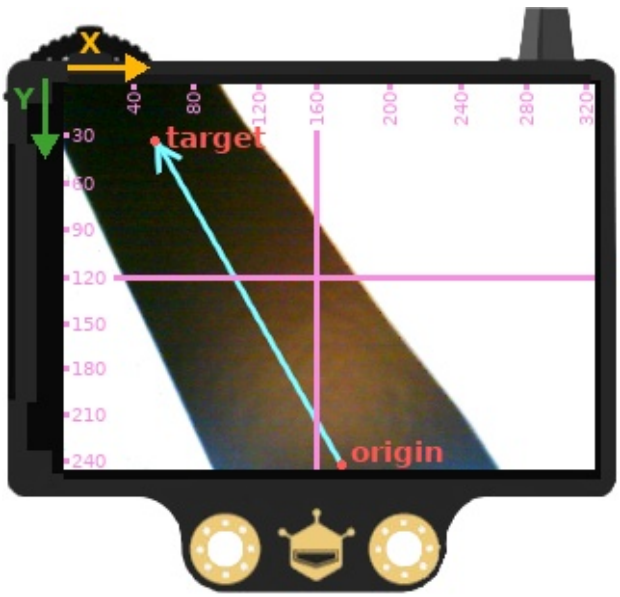
peut être mis en relation avec le système de coordonnées du HuslyLens.

08RI91 – Système de coordonnées du Huskylens

L'exécution du script `simple.py` produit les informations suivantes où il est possible de mettre en relation les informations retournées avec le système de coordonnées du HyskyLens.

```
>>> import simple
----------------------------------
item 0 is an Arrow ID1. Origin is 168,238, Target is 56,38
----------------------------------
item 0 is an Arrow ID1. Origin is 168,238, Target is 56,38
----------------------------------
item 0 is an Arrow ID1. Origin is 168,238, Target is 56,38
...
```

L'image ci-dessous met en correspondance l'image et les informations de la flèche (*Arrow*) retournée par HuskyLens.



08RI101 – Coordonnées de la flèche

4.5.Assembler

Jean-Christophe du site ArduiBlog à créer un pare-choc pour Zumo Robot. Cet objet que l'on peut reproduire avec une imprimante 3D.

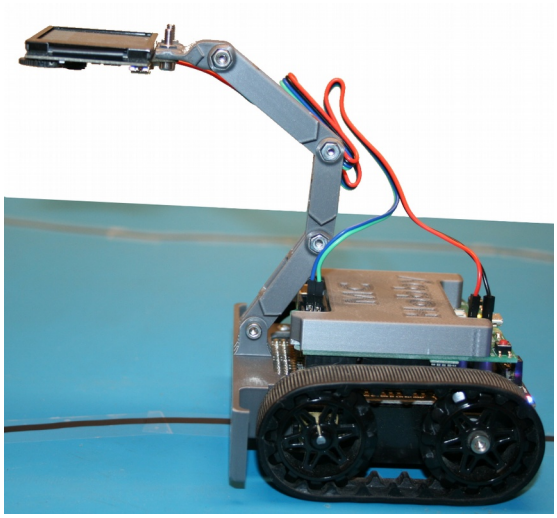
Pour réaliser ce projet, Jean-Christophe modifié le pare-choc pour pouvoir y fixer le HuskyLens au bout d'un bras.

08RI92 – Pièces pour monter la caméra HuskyLens

Ces éléments sont disponibles en libre accès sur le site Printables.

https://www.printables.com/fr/model/742851-accessories-for-zumo-robot/files

Une fois imprimés, les divers éléments sont assemblés comme suit pour placer la camera au dessus de l'avant du robot face au sol (bien à l'horizontal pour éviter le phénomène de distorsion d'image).



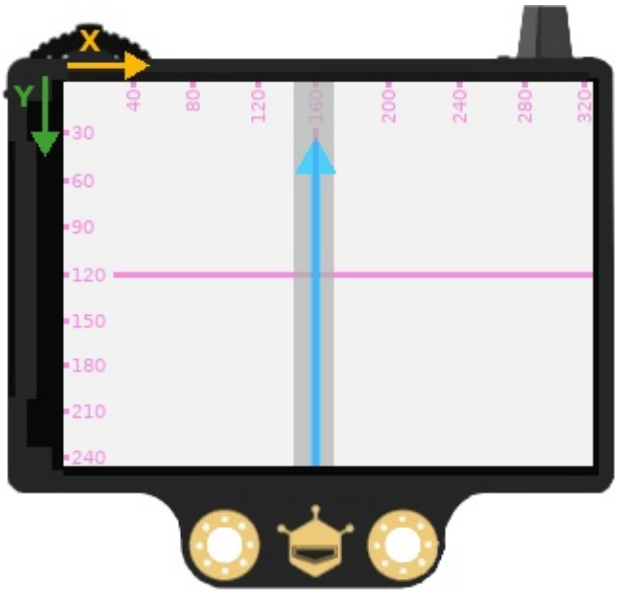
08RI93 – montage du HuskyLens sur le Zumo Robot

4.6.Principe de fonctionnement

Voici une description de quelques cas de figure et l'impact sur le contrôle moteur du robot.

4.6.1.Cas idéal

Lorsque la ligne (flèche) est pile sous la verticale centrale de l'écran alors le robot peut continuer tout droit.



08RI102 – Ligne centrée sous la caméra

4.6.2.Tourner à droite ou à gauche

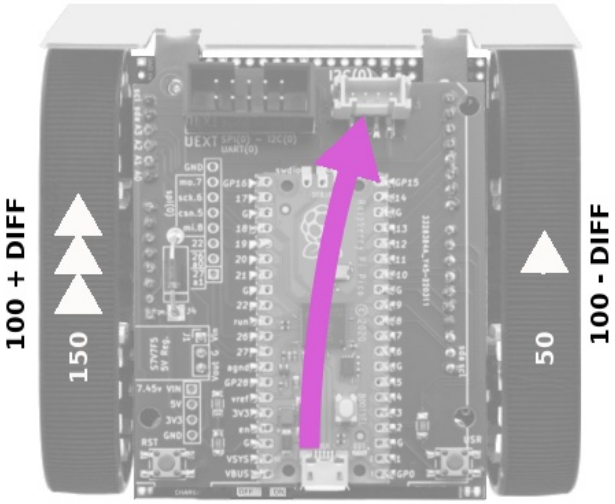
Si la ligne n'est pas centrée sous la verticale alors il convient de corriger la direction du robot en modifiant la vitesse des moteurs.

08RI103 – Ligne dérivant à droite

La flèche indique une dérive sur la droite. Plus la différence sur l'axe X entre les points *target* et *origin* est grand et plus la dérive est importante (plus le tournant oblique).

Cette différence `target.x-origin.x` est communément appelée l'erreur, erreur qu'il faut minimiser pour rester au dessus de la ligne.

L'erreur est utilisée pour appliquer une différence de vitesse entre les moteurs du zumo de sorte à corriger la trajectoire. Dans le cas présent, le robot doit tourner sur la droite suivre la trajectoire (donc accélérer le moteur gauche et ralentir le moteur droit).



08RI103b – modification de la vitesse des moteurs

Lorsque l'`erreur=target.x-origin.x` est positif alors le tournant vire sur la droite. Lorsque la différence est négative alors le tournant vire sur la gauche.

4.6.3.Attention aux pièges

S'il est traité comme ci-dessus, le cas présenté ci-dessous produira un défaut de conduite probablement irrécupérable.

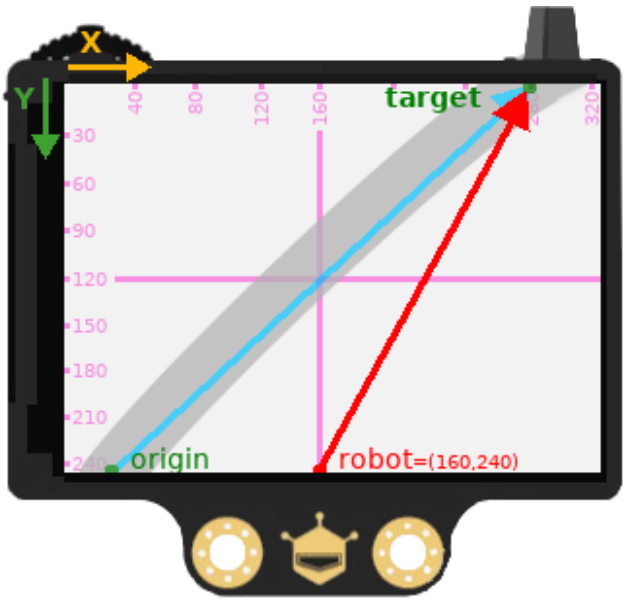
08RI104 – cas limite produisant une erreur.

La différence de coordonnée X entre les points *origin* et *target* est beaucoup plus marqué. Cela traduit une erreur importante qui implique une différence de vitesse plus importante entre les moteurs. Le virage du robot sur la droite sera donc très important.

Le risque, c'est de tourner tellement fort sur la droite que le robot ne puisse rejoindre la ligne !

Le problème est ici que la ligne démarre à gauche de la caméra alors qu'elle devrait idéalement se trouver pile sous la caméra (donc sur la verticale centrale et en bas de l'écran).

En considérant le centre inférieur de l'écran (160,240) comme l'avant centrale du robot, il sera plus efficace de diriger le robot en calculant l'erreur depuis ce point vers la destination (*target*).



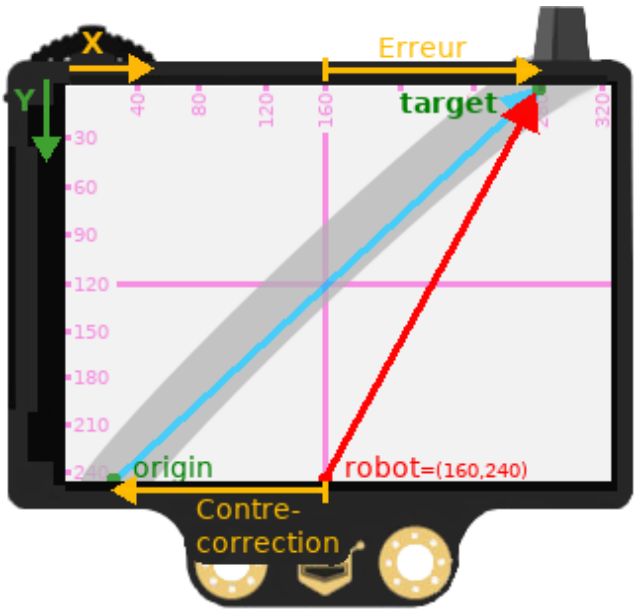
08RI105 – utiliser le point robot comme point de départ

En partant du centre inférieur de l'écran (point « robot ») vers le point « *target* », la différence `erreur=target.x-robot.x=target.x-160` produit une différence de vitesse plus faible.

4.6.4.Appliquer une contre-correction

Le cas ci-dessus est intéressant à plus d'un point. La conduite idéale serait un déplacement en ligne droite jusqu'à ce que la ligne coupe le point « robot ».

Dans un fonctionnement normal, la ligne se trouve sous le point « robot », il est donc possible d'utiliser la différence `ccorr=robot.x-origin.x=160-origin.x` pour appliquer une contre-correction.



08RI105b – Evaluation de l'erreur et de sa contre-correction

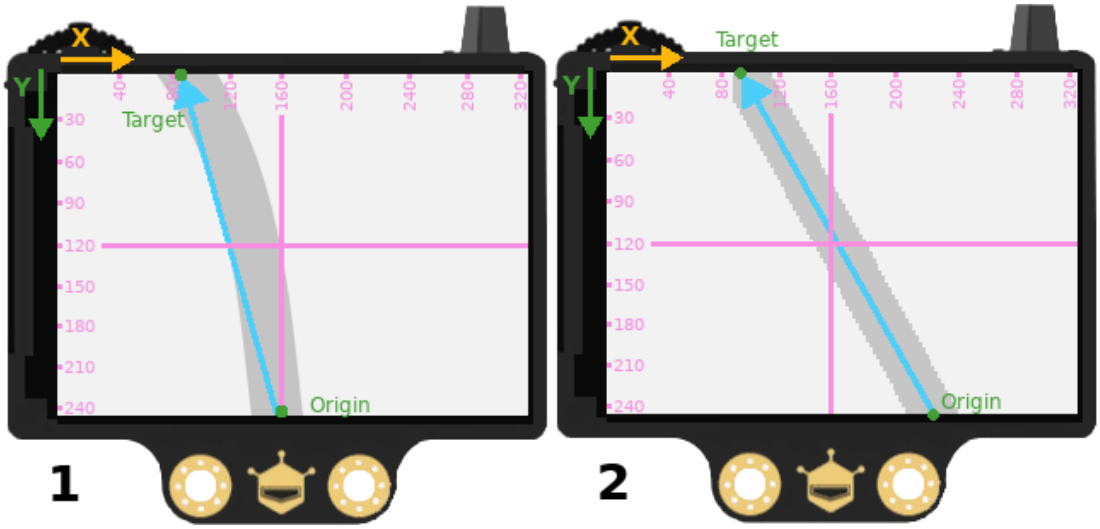
On peut intuitivement réaliser que soustraire la contre-correction à l'erreur de trajectoire permet d'offrir une trajectoire rectiligne dans le cas présent.

Cette contre-correction tend vers 0 lorsque le robot est au dessus de la ligne.

La formulation finale de l'erreur devient :

```
erreur=(target.x-160)-(160-origin.x)
```

Quelques cas de figure complémentaires vont permettre de confirmer l'utilité de la contre-correction.



08RI106a – cas de figure complémentaires

Figure 1

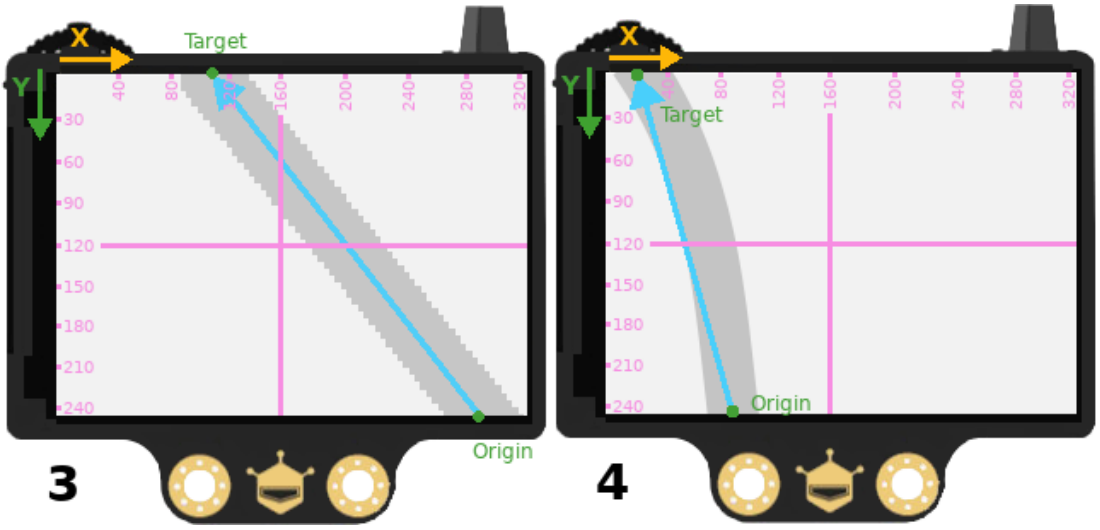
```
origin = (160, 240)
target = (80, 0)
erreur=(target.x-160)-(160-origin.x)
erreur = (80-160)-(160-160) = -80
```

Impact moteur : virage sur la gauche

Figure 2

```
origin = (220, 240)
target = (90, 0)
erreur=(target.x-160)-(160-origin.x)
erreur = (90-160)-(160-220) = -10
```

Impact moteur : léger virage sur la gauche. Cette conduite en ligne presque droite permet de rejoindre plus rapidement la ligne.



08RI106b – cas de figure complémentaires

Figure 3

```
origin = (290, 240)
target = (110, 0)
erreur=(target.x-160)-(160-origin.x)
erreur = (110-160)-(160-290) = +80
```

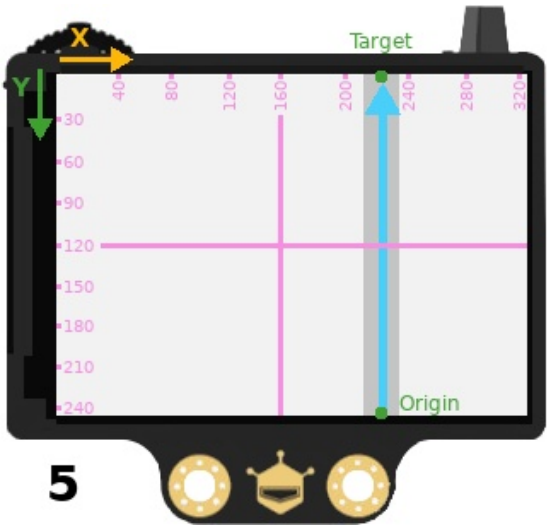
Impact moteur : virage sur la droite. Permet de rejoindre la ligne plus rapidement !

Figure 4

```
origin = (60, 240)
target = (20, 0)
erreur=(target.x-160)-(160-origin.x)
erreur = (20-160)-(160-60) = -240
```

Impact moteur : virage serré sur la gauche pour retrouver la ligne.

x



08RI106c – cas de figure complémentaires

Figure 5

```
origin = (220, 240)
target = (220, 0)
erreur=(target.x-160)-(160-origin.x)
erreur = (220-160)-(160-220) = 120
```

Impact moteur : virage appuyé sur la droite pour rejoindre la ligne

4.6.5.Ligne hors de portée

La ligne est supposée se trouver sous le zumo robot (ou sur sa droite ou sur sa gauche).

Dans les fait, lors de l'identification de la ligne par HuskyLens, il arrive que celle-ci « oscille » autour de son point d'origine (et aussi de destination) sans que cela n'ai un réel impact sur le suivit de parcours car la direction est sensiblement toujours la même.

A noter que l'asservissement est uniquement réalisé sur les coordonnées X. La position Y du point *origin* est supposée être à `y=240` et à `y=0` pour *target*.

Les coordonnées Y sont elles aussi victimes d'une forme de bruit. Par conséquent Y peut être légèrement inférieur à 240 pour le point *origin* et légèrement supérieur à 0 pour le point *target*.

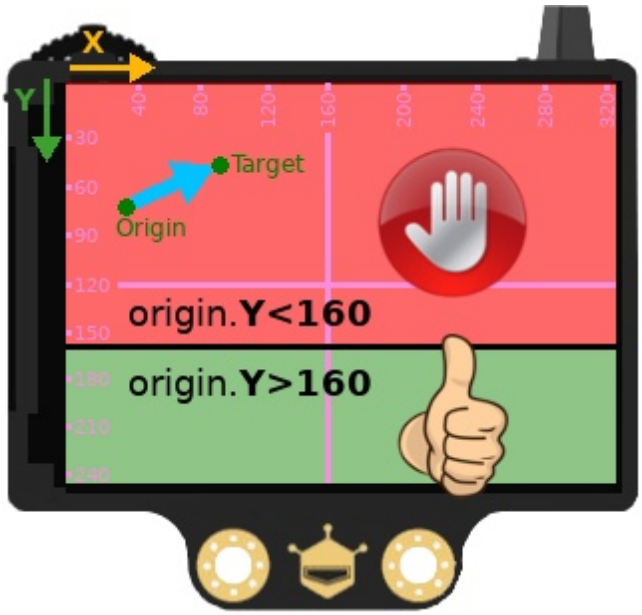
L'algorithme peut également identifier une ombre ou un défaut de sol comme ligne, celle-ci se trouvant dans une portion inattendue de la capture (c'est toujours considéré comme du bruit).

L'exemple ci-dessous présente le cas d'un artefact identifié par erreur comme une ligne.

08RI107 – Artefact identifié comme une ligne

Les artefacts n'étant que temporaire, il suffit simplement d'ignorer toutes les lignes dont l'origine n'est pas dans le dernier tiers inférieur de l'écran.

Si l'artefact persiste (plus de 20 détections successives) alors il convient d'arrêter le robot pour qu'il puisse être manuellement réaligné sur la ligne.



08RI108 – Zone de validité pour origin.Y

Si l'artefact persiste plus de 20 détections consécutives c'est peut être parce que cela n'est pas un artefact mais une vraie ligne. Pour une ligne se trouvant dans les 2ier tiers supérieur de l'écran, le robot pourrait se diriger vers le point d'origine.

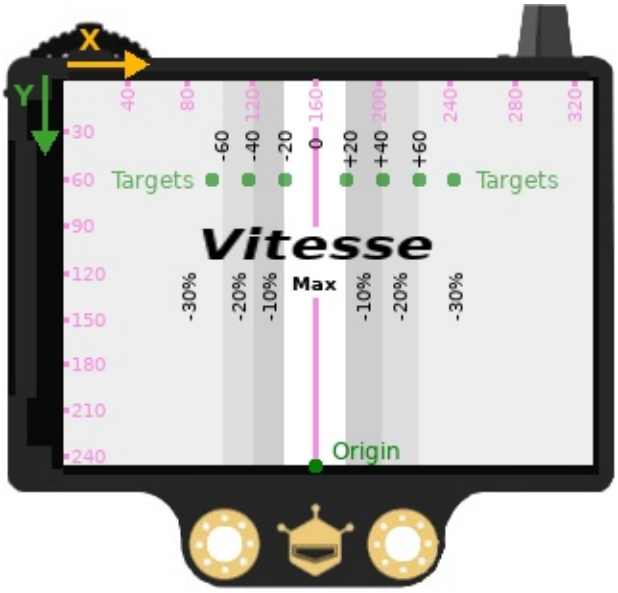
4.6.6.Tournant à vitesse réduite

C'est connu, il ne faut jamais prendre un virage a pleine vitesse ! Dans le cas d'une automobile c'est un problème de force centrifuge et d'adhérence des pneu au sol.

Dans le cas du Zumo, la vitesse risque de faire sortir la ligne hors du champ de vision de la caméra avant que la correction ne puisse être appliquée sur les moteurs.

Il y a donc intérêt à ralentir durant la négociation d'un virage. Il est assez facile d'identifier un virage puisque la flèche présente un `ΔX>0` avec `ΔX = abs(target.x - origin.x)` . Plus le virage est important et plus l'écart de X est grand. Plus le virage est important et plus il convient de ralentir !

L'image ci-dessous présente la diminution de vitesse (en pourcent de la vitesse max) en fonction de l'écart par rapport à une trajectoire rectiligne.



08RI109 – modification dynamique de la vitesse

4.7.Script husky_line

Les conditions décrites ci-dessous sont mises en œuvre dans le script `husky_line.py` de 50 lignes qui permet de contrôler le Zumo Robot avec le HuskyLens.

La bonne exécution de ce script nécessite de configurer le HuskyLens en mode « Line Tracking » et d'avoir effectuer l'apprentissage.

Le script `husky_line.py` est également disponible dans le dépôt du projet :

•https://github.com/mchobby/micropython-zumo-robot/blob/main/extras/HuskyLens/husky_line.py

```
01: from zumoshield import ZumoShield
02: from husky import HuskyLens
03: import time
04:
05: z = ZumoShield()
06: hl = HuskyLens( z.i2c )
07:
08: DEFAULT_SPEED = 100
09: DYN_SPEED_RATIO = DEFAULT_SPEED//10
10: ERROR_RATIO = 0.5
11:
12: def clamp( val, _min, _max ):
13:   return max(min(_max, val), _min)
14:
15: print( "Press Button to start" )
16: z.buzzer.play(">g8>>c8")
17: z.button.waitForButton()
18: time.sleep(1)
19:
20: dyn_speed = DEFAULT_SPEED
21: retries = 0
22: try:
23:   while(True):
24:     if z.button.isPressed():
25:       raise Exception( "User abort")
26:     lst = hl.get_arrows( learned=True )
27:     if not(lst):
```

```
28:       retries+=1
29:       if retries > 20:
30:         z.motors.stop()
31:       continue
32:
33:     arrow = lst[0]
34:
35:     if arrow.origin.y < 160:
36:       retries+=1
37:       if retries > 20:
38:         z.motors.stop()
39:       continue
40:
41:     retries = 0
42:
43:     error = (arrow.target.x-160) - (160-arrow.origin.x)
44:     speed_diff = int(error * ERROR_RATIO)
45:
46:     dyn_speed = DEFAULT_SPEED - clamp(
            abs(arrow.target.x-arrow.origin.x)//20, 0, 3 )
            * DYN_SPEED_RATIO
47:
48:     m1Speed = dyn_speed+speed_diff
49:     m2Speed = dyn_speed-speed_diff
50:     m1Speed = clamp( m1Speed, 0, 400 )
51:     m2Speed = clamp( m2Speed, 0, 400 )
52:
53:     z.motors.setSpeeds(m1Speed,m2Speed)
54: finally:
55:   z.motors.stop()
56:   z.play_2tones()
```

Voici les détails du fonctionnement du programme dont le corps principal s'étale des lignes 23 à 53.

•Ligne 1 et 2 : import des bibliothèques nécessaires pour contrôler le Zumo (classe `ZumoShield`) et le HuskyLens (classe `HyskyLens`).

•Ligne 5 : création de l'instance du `ZumoShield` (variable `z`) qui alloue toutes les ressources nécessaire au pilotage du Zumo, y compris le bus I2C permettant l'accès à la centrale inertielle du Zumo.

•Ligne 6 : création de l'instance du `HuskyLens` (variable `hl`) en passant en paramètre l'instance du port I2C à utiliser pour communiquer avec HuskyLens. Le HuskyLens est branché sur le même bus I2C que la centrale inertielle du Zumo, la référence du bus I2C est retrouvée par l'intermédiaire de la classe `ZumoShield` (qui a déjà alloué cette ressource).

•Ligne 8 : constante `DEFAULT_SPEED` indique la vitesse par défaut du robot. La rotation vers la droite/gauche est assurée par application d'une différence de vitesse addittionnée & soustraite à la vitesse par défaut.

•Ligne 9 : constante `DYN_SPEED_RATIO` représente les 10 % de la vitesse par défaut utilisé pour réduire la vitesse du robot de 10 %, 20 % ou 30 % durant la négociation des virages. L'opérateur de division entière « // » permet d'éviter une valeur fractionnaire.

•Ligne 10 : constante `ERROR_RATIO` est la composante proportionnelle de l'erreur d'orientation (ΔX) reporté en différence de vitesses entre les moteurs droit et gauche. Cette constante permet de pondérer la rotation du robot dans un tournant.

•Lignes 12 et 13 : la fonction clamp() permet de borner une valeur (val) entre un minimal et un maxima. Toute valeur supérieure au maxima sera ramenée au maxima, toute valeur inférieure inférieure au minima sera ramenée au minima.

•Lignes 26 et 27 : invite sonore (et message REPL) indiquant qu'il faut presser le bouton utilisateur pour démarrer le programme de suivit de ligne.

•Ligne 28 : attente de la pression du bouton utilisateur du Zumo (aussi reporté sur l'adaptateur).

•Ligne 29 : attendre une seconde avant de démarrer le suivit de ligne.

•Ligne 31 : variable `dyn_speed` est destinée à recevoir la valeur de la vitesse dynamique (celle supposée diminuer durant la négociation d'un virage). Par défaut, cette vitesse est fixée à la vitesse par défaut.

•Ligne 32 : variable `retries` indiquant le nombre de requêtes consécutives du HuskyLens sans information de direction ou avec des informations de direction non admissibles (ne répondant aux contraintes de conduite).

•Ligne 33, 72 et 73 : section `try ... finally` qui assure l'exécution de la section `finally` en toutes circonstances (y compris en cas d'erreur). La section `finally` assurer ici l'arrêt des moteurs `z.motors.stop()` et produit un signal sonore deux tons.

•Ligne 34 : boucle infinie `while(True)` qui exécute continuellement le corps du programme décris ci-dessous.

Corps du programme

Le corps du programme se comporte comme suit :

1.L'acquisition de la direction à suivre en interrogeant HuskyLens (variable `arrow`).

2.Le calcul de l'erreur de trajectoire (variable `error`).

3.L'évaluation de la différence de vitesse (variable `speed_diff`) appliqué sur les chenilles en vue d'effectuer une rotation.

4.L'évaluation de la vitesse dynamique (variable `dyn_speed`) = vitesse par défaut diminuée d'un certain pourcentage en fonction de l'importance du virage.

5.Calcul de la vitesse du moteur droit et gauche (variables `m1Speed` et `m2Speed`) et application sur le zumo.

Voici un description détaillée du corps du script

•Lignes 24 et 25 : lancement d'une exception si le bouton utilisateur est enfoncé. Cette exception interrompt brutalement l'exécution de la boucle infinie (`while True`) et transfert l'exécution à la section `finally` (ligne 54) qui fait en sorte d'arrêter le robot.

•Ligne 26 : interrogation du HuskyLens pour obtenir la liste de flèches (méthode `get_arrows()`) identifiées. Le paramètre `learned=True` restreint les résultats aux flèches identifiées suite à l'apprentissage. Le résultat est stocké dans la variable `lst` destiné à recevoir une liste d'objets (ou `None` s'il n'y a pas d'éléments identifiés par HuskyLens).

•Lignes 27 à 31 : s'il n'y a pas d'objet retourné par `get_arrows()` alors il faudra redémarrer immédiatement une nouvelle acquisition de donnée sur le HuskyLens, ce que fait l'instruction `continue` (depuis la ligne 31). A noter que ce cas d'erreur provoque l'incrémentation du compteur `retries`. Si `retries` atteints 20 erreurs consécutives alors le Zumo arrête ses moteurs (lignes 29 et 30).

•Ligne 33 : La liste `lst` contient des objets, forcement la description d'une ou plusieurs flèches conséquent à l'appel de la méthode `get_arrows()` . Cette ligne

n'est exécutée que si une liste est retournée (et donc qu'elle contient un élément). L'expression en ligne 33 permet de récupérer la référence vers le premier objet de la liste et de stocker celle-ci dans la variable `arrow`.

•Lignes 35 à 39 : si le point d'origine de la flèche n'est pas dans le dernier tiers de l'écran (donc Y > 160) alors il s'agit d'un cas non admissible pour la conduite ! Le compteur `retries` est incrémenté et comme précédemment, si le compteur atteint 20 erreurs consécutives le robot est stoppé (sinon il poursuit son mouvement). Que le robot soit stoppé ou non, l'instruction continue redémarre la boucle `while True` pour effectuer une nouvelle acquisition.

•Ligne 41 : cette ligne est exécutée lorsque le HuskyLens retourne une flèche et que celle-ci répond aux contraintes de conduites. La variables `retries` (compte cumulé d'erreurs consécutives) peut être réinitialisée à zéro.

•Ligne 43 : évaluation de l'écart `arrow.target.x-160` et calcul de la contre-correction `160-arrow.origin.x` pour déterminer l'erreur (variable `error`).

•Ligne 44 : évaluer la proportion de l'erreur (`ERROR_RATIO`) utilisée pour calculer la différence de vitesse (`speed_diff`) qui sera appliquée sur les moteurs du Zumo.

•Ligne 46 : évaluation de la vitesse dynamique `dyn_speed` à partir de la vitesse de référence `DEFAULT_SPEED`. L'expression `abs(arrow.target.x-arrow.origin.x)` permet d'obtenir l'écart horizontal entre les deux extrémités de la flèche en valeur absolue (donc non signé). Cet écart est ensuite divisé par 20 (// pour division entière) pour déterminer l'écart en tranche de 20 pixels de large. Enfin la fonction `clamp()` permet de limiter le nombre de tranches entre 0 et 3, nombre qui sert de multiplicateur pour `DYN_SPEED_RATIO` afin de réduire la vitesse du véhicule. La fonction `clamp()` permet de limiter l'impact sur la vitesse à 30 %. Par ailleurs, si l'écart ΔX est inférieur à 20 alors la division entière par 20 retournera 0 (`dyn_speed` sera donc égal à `DEFAULT_SPEED`).

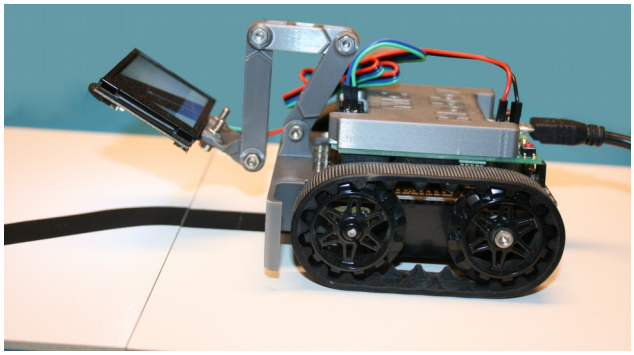
•Lignes 48 et 49 : calcul de la vitesse des moteurs (variables `m1Speed` et `m2Speed`). En ajoutant ou soustrayant `speed_diff` de la vitesse de référence du robot (variable `dyn_speed`).

•Lignes 50 et 51 : borner la vitesse des moteurs entre 0 et 400 (`speed_diff` pouvant dépasser la valeur de 300 dans les cas les plus extrêmes).

•Ligne 53 : modification des vitesses des deux moteurs.

4.8.Cas de la distorsion

Lors d'un premier essai d'usage du HuskyLens, celui-ci était monté sur le pare-choc avec une vue en angle sur le parcours (30° par rapport à l'horizontal).

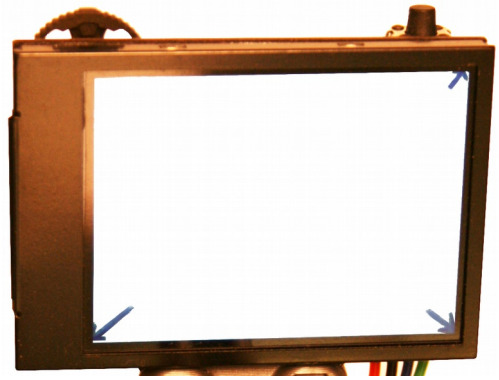


08RI93b – Vue en angle 30° sur le parcours

La mise en œuvre de algorithme présentait des instabilités relativement sévères.

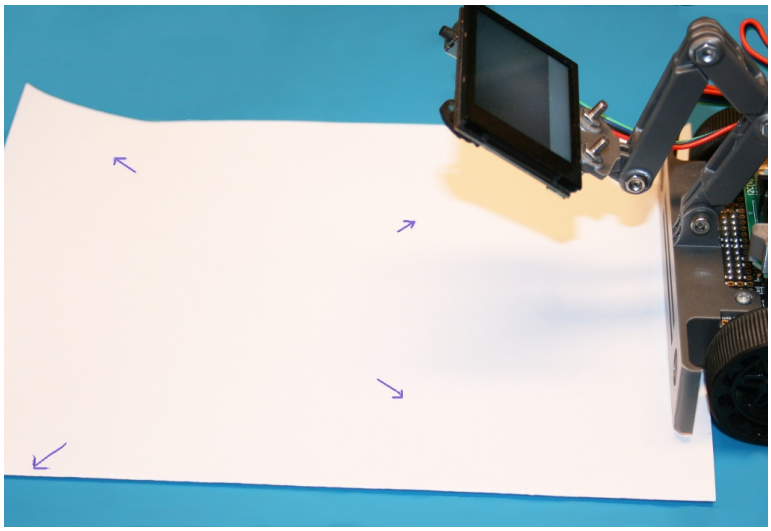
Il se fait que si l'écran présente une vue apparemment plane, la vue correspond en réalité à un trapèze.

Pour s'en convaincre, il faut placer une feuille de papier sous la camera puis dessiner l'emplacement des quatre coins de l'écran sur la feuille en regardant par l'écran (placer la pointe du crayon au centre de l'image puis la diriger vers un coin, une fois celui-ci atteint baisser la mine sur le papier pour tracer l'emplacement).



08RI110 – Marquage des coins via l'écran

Voici le résultat obtenu sur la feuille de papier où la distorsion en position haute est clairement identifiable.



08RI110b - distorsion en position haute

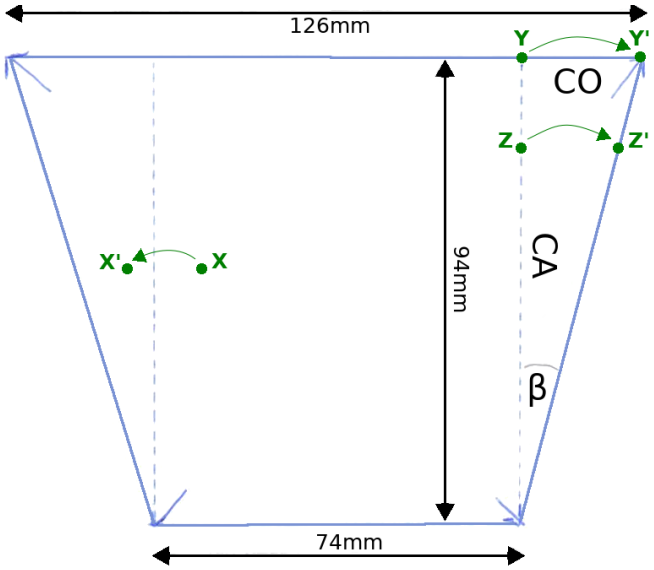
Par conséquent, une ligne droite sur l'écran correspond en réalité à une ligne en biais (sauf si celle-ci est placée pile sur la médiane verticale de l'écran).

4.8.1.L'angle de distorsion

Les coordonnées reportées par HuskyLens (coordonnées sur l'écran) doivent donc être corrigées pour être plus représentatif de la position réelle de la ligne (tel qu'elle est au niveau de la feuille de papier).

En effet, l'écart ΔX sur la feuille de papier (ΔX_réel) est plus grand que celui indiqué sur le HuskyLens (ΔX_huskylens).

Le relevé des coins (sur la feuille de papier) est complété pour former le trapèze et mesurer les différents dimensions.



08RI111 – vue réelle capturée (feuille de papier)

L'angle β est l'angle de distorsion celui qui projette :

•Le point Y (écran HuskyLens au coin supérieur-droit) sur Y' (position réelle du point Y sur le papier).

•Le point Z (écran HusyLens sur bord droit) sur Z' (position réelle du Z sur le papier).

•Le point X (écran HuskyLens) sur X' (position réelle sur le papier).

En identifiant la valeur de l'angle de distortion β, il sera possible possible de corriger les coordonnées X des points retournés par HuskyLens.

La définition trigonométrie que la tangente permettra de calculer β

```
Tan(β) = côté-opposé/côté-adjacent = CO/CA
Tan(β) = ((126-74)/2) / 94 = 0,276595
β = arc-tan( CO/CA ) = arc-tan( 0,276595 ) = 15,46°
```

Maintenant que l'angle de distorsion β est connu, il sera possible de l'utiliser dans un script python pour calculer la position réelle de la ligne depuis les coordonnées retournées par HuskyLens. Ainsi, le script disposera des vrais écarts ΔX pour évaluer la direction.

4.8.2.Script husky_line2

Les conditions de distorsion sont prises en compte dans le script `husky_line2.py` très similaire à la précédente version.

Le script `husky_line2.py` est également disponible dans le dépôt du projet :

•https://github.com/mchobby/micropython-zumo-robot/blob/main/extras/HuskyLens/husky_line2.py

```
01: from zumoshield import ZumoShield
02: from husky import HuskyLens, Point
03: from math import tan, radians
04: import time
05:
06: z = ZumoShield()
06: hl = HuskyLens( z.i2c )
07:
08: DEFAULT_SPEED = 100
09: DYN_SPEED_RATIO = DEFAULT_SPEED//10
```

```
10: ERROR_RATIO = 0.4
11:
12: DISTORTION_ANGLE = 15.46
13: DISTORTION_RAD = radians( DISTORTION_ANGLE )
14:
15: class CorrectedArrow:
16:   def __init__( self ):
17:     self.origin = Point()
18:     self.target = Point()
19:
20: def flatten( point ):
21:   mult = -1 if point.x < 160 else 1
22:   corr_x = point.x + mult * (240-point.y)
                * tan( DISTORTION_RAD )
23:   corr_y = point.y
24:   return (corr_x,corr_y)
25:
26: corr_arrow = CorrectedArrow()
27:
28: def clamp( val, _min, _max ):
29:   return max(min(_max, val), _min)
30:
31: print( "Press Button to start" )
32: z.buzzer.play(">g8>>c8")
33: z.button.waitForButton()
34: time.sleep(1)
35:
36: dyn_speed = DEFAULT_SPEED
37: retries = 0
38: try:
39:   while(True):
40:     if z.button.isPressed():
41:       raise Exception( "User abort")
42:     lst = hl.get_arrows( learned=True )
43:     if not(lst):
44:       retries+=1
45:       if retries > 20:
46:         z.motors.stop()
47:       continue
48:
49:     arrow = lst[0]
50:
51:     if arrow.origin.y < 160:
52:       retries+=1
53:       if retries > 20:
54:         z.motors.stop()
55:       continue
56:
57:     retries = 0
58:
59:     corr_arrow.target.set( flatten(arrow.target) )
60:     if arrow.origin.y < 230:
61:       corr_arrow.origin.set( flatten(arrow.origin) )
62:     else:
63:       corr_arrow.origin.set( arrow.origin )
64:
65:     error = (corr_arrow.target.x-160) -
                  (160-corr_arrow.origin.x)
66:     speed_diff = int(error * ERROR_RATIO)
67:
68:     dyn_speed = DEFAULT_SPEED - clamp(
           abs(corr_arrow.target.x-corr_arrow.origin.x)//20,
                0, 3 )*DYN_SPEED_RATIO
69:
70:     m1Speed = dyn_speed+speed_diff
71:     m2Speed = dyn_speed-speed_diff
72:     m1Speed = clamp( m1Speed, 0, 400 )
```

```
73:     m2Speed = clamp( m2Speed, 0, 400 )
74:
75:     z.motors.setSpeeds(m1Speed,m2Speed)
76: finally:
77:   z.motors.stop()
78:   z.play_2tones()
```

Le corps de cette version du script est très similaire au précédent ; ainsi seules les différences flagrantes ferons l'objet d'une description plus précise.

•Ligne 2 : import complémentaire de la classe Point qui sera nécessaire pour stocker les coordonnées corrigées.

•Ligne 3 : import de la fonction trigonométrique tangente `tan()` et la fonction `radians()` pour transformé des degrés angulaires en radians. Les fonctions trigonométriques de MicroPython utilisent des radians comme argument.

•Ligne 10 : `ERROR_RATIO` est diminué à 0,4 . L'expérience à démontré une plus grande stabilité en diminuant l'impact de l'erreur sur la différence de vitesse (`speed_diff`) .

•Ligne 12 : définition de l'angle de distorsion DISTORTION_ANGLE tel qu'identifié à l'aide de la feuille de papier (et d'un peu de trigonométrie).

•Ligne 13 : transformation de l'angle en degrés angulaires vers un angle en radians.

•Ligne 15 à 18 : définition de la classe `CorrectedArrow` permettant de stocker les coordonnées corrigées pour l'`origin` et le `target`. La classe `Point` utilisée dans la création permet, elle, de stocker une valeur pour `x` et pour `y`.

•Lignes 20 à 24 : définition de la fonction `flatten()` qui permet d'aplatir les coordonnées de l'écran vers « la feuille de papier ». La fonction prend un `Point` en paramètre et retourne le couple `( X_corrigé, Y_corrigé )` .

•Ligne 26 : déclaration `corr_arrow`, instance de la classe `CorrectedArrow` permettant de stocker facilement les coordonnées corrigées pour un point `origin` et un point `target` .

•Ligne 59 : calcul de la correction de la position `target` du HuskyLens avec la fonction `flatten()`. Comme `target` se trouve en haut de l'écran, la correction sera forcement importante.

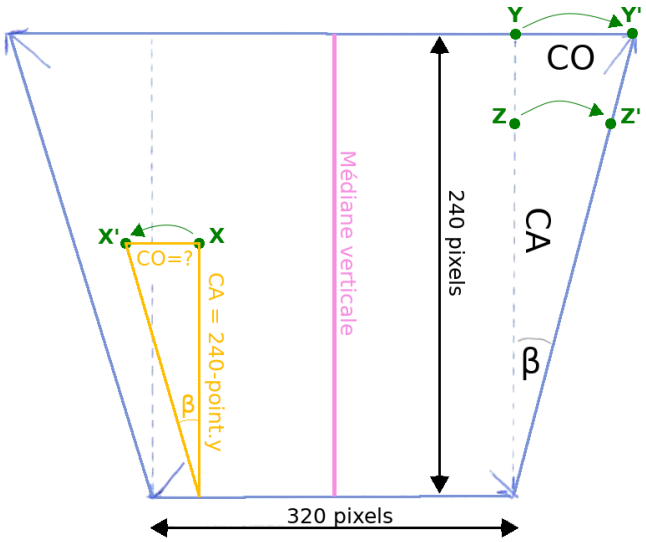
•Ligne 60 et 61 : si le point `origin` n'est pas dans la partie basse de l'écran, il faut donc y appliquer le calcul de distortion avec `flatten()`.

•Ligne 62 et 63 : le point `origin` se trouve dans les 10 derniers pixels en bas de l'écran. Pas besoin d'effectuer de correction de coordonnées dans ce cas car son influence sera négligeable sur le calcul d'erreur.

•Lignes 65 et suivantes: a partir de cette ligne l'instance `corr_arrow` contient la définition des points `origin` (via `corr_arrow.origin`) et `target` (via `corr_arrow.target`) dont les coordonnées X sont corrigées. Le restant du code est identique à l'exception que `arrow` est remplacé par `corr_arrow` .

La fonction flatten()

La fonction `flatten( point )` reçoit les coordonnées d'un `Point` du HuskyLens (ex : `source` ou `target`) et calcule les coordonnées à plat (sur la feuille de papier) pour tenir compte de l'effet de distorsion.



08RI112 – image de l'écran HuskyLens et corrections

Grâce à l'angle β et la hauteur du point (CA=240-point.y), il est possible de calculer la déviation horizontale (CO) correspondante.

C'est ce qui permet de calculer le point Y' (sur la papier) à partir de Y (sur l'écran). Il en est de même pour Z' à partir de Z et X' à partir de X.

```
Tan( β ) = CO/CA
CO = CA * Tan( β )
CO = (240-point.y)*Tan( β )
```

Deux cas de figures se présentent selon que le point est à droite ou a gauche de la médiane verticale de l'écran :

1.A droite : la correction calculée (CO) **est additionné** à la coordonnée x du point. Cela repousse le point plus loin sur la droite.

2. A gauche : la correction calculée (CO) **est soustraite** à la coordonnée x du point. Cela repousse le point plus loin à gauche, quitte à ce que la coordonnée devienne négative.

Le fait que la coordonnée x puisse devenir négative ne perturbe en rien le calcul d'écart ! Par exemple `origin.x=120` *et* `target.x=10` *alors l'écart* `origin.x-target.x=120 - 10 = 110`. *Si après correction* `target.x` *passe de 10 à -5 alors* `origin.x-target.x=120 - (-5) = 125`. *L'écart est effectivement plus grand.*

```
20: def flatten( point ):
21:   mult = -1 if point.x < 160 else 1
22:   corr_x = point.x + mult * (240-point.y)
                * tan( DISTORTION_RAD )
23:   corr_y = point.y
24:   return (corr_x,corr_y)
```

•Ligne 20 : définition de la fonction `flatten()`. Le paramètre point est destiné à recevoir un objet de type `Point` offrant l'accès aux attributs `x` et `y`.

•Ligne 21 : l'expression ternaire permet d'assigner -1 ou +1 suivant la position horizontale du point (coordonnée `x`) sur l'écran. Passé la médiane verticale le multiplicateur est +1 (pour faire une addition), en deçà le multiplicateur sera -1 (pour faire une soustraction).

•Ligne 22 : calcul de la coordonnée X corrigée `corr_x` . La formule répond en tout point aux explications présentées dans la description de la fonction `flatten()`.

•Ligne 23 : calcul de la coordonnée Y corrigée `corr_y`. Comme les coordonnées Y n'ont pas d'impact sur les calculs, celle-ci est conservée à l'identique.

•Ligne 24 : retourne un tuple contenant les coordonnées X et Y corrigées.

5.Zumo Logo

xxxx.

5.1.xxxx

Xxx

5.2.xxxx

Xxx