

Nom et prénom de l'élève ingénieur : Bouilly Enzo,
Le Borgne Pierre
Année / Spécialité : 2017 IAI
Tuteur Polytech : Flavien Vernier et François
Leplus

RAPPORT DE PFE

Réhabilitation d'un robot Pekee I



POLYTECH[®]
ANNECY-CHAMBÉRY

Polytech Annecy
5 Chemin de Bellevue, 74940 Annecy-le-Vieux
04 50 09 66 00

Introduction :

Dans le cadre de notre projet de fin d'étude (PFE) en 5^{ème} année d'IAI, nous avons choisis le sujet « Réhabilitation d'un robot Pekee I ». L'objectif était de remettre en service le robot Pekee I.

Ce projet consistait alors à reprendre les parties mécaniques (structure, moteurs) du robot, et si possible les capteurs présent sur la carte électronique. Tout en intégrant un nouveau système permettant de piloter le robot, à savoir : un module pont en H pour commander les moteurs, une carte Arduino mega pour le contrôle bas niveau, et enfin une Raspberry PI 3 pour la programmation du robot.

Ce PFE nous a paru intéressant dans le fait qu'il intègre 2 domaines différents (électronique/informatique) mais que ceux-ci sont complémentaires. Ainsi nous pourrions approfondir nos connaissances et compétences dans ces domaines.

Le projet a été réalisé au sein de l'école, dans un premier temps avec ½ journée par semaine de la semaine 47 à 51 pendant 5 semaines. Puis une journée par semaine pendant 2 semaines de la semaine 2 à 4. Et enfin à plein temps de la semaine 5 à 12 durant 8 semaines. Pendant la réalisation du projet nous avons été accompagnés par nos responsables : M. Vernier pour la partie informatique/programmation, et M. Leplus pour la partie électronique/mécanique. Nous avons eu aussi accès à une salle où nous pouvions trouver le matériel nécessaire pour les tests que nous avons à réaliser.

Ainsi pendant ce PFE, nous avons mis en place un système de pilotage du robot. Ce rapport va résumer les semaines passées à réhabiliter ce robot. Dans un premier temps nous allons présenter les tests moteurs, puis comment nous les avons pilotés à l'aide de l'Arduino et du module. Puis la partie Raspberry qui comprend la connexion et communication avec l'Arduino. Ensuite une explication de nos tests sur le capteur Infra-rouge et enfin les parties rajoutées au robot comme la coque 3D et le support ainsi que le shield Arduino.

S O M M A I R E

Moteurs & Encodeurs 1

Test des Moteurs	1.1
Pilotage des Moteurs	1.2
Test Virages	1.3
Blocs de Mouvements	1.4
Pilotage des Moteurs en Boucle Fermée	1.5
Machine d'état Arduino	1.6

Raspberry 2

Connexion avec l'Arduino	2.1
Communication avec l'Arduino	2.2
API	2.3

Capteur IR 3

Test Capteur Pekee	3.1
Proposition de nouveau capteurs IR	3.2

Pièces 3D & Shield arduino 4

Conception	4.1
------------	-----

Test des moteurs :

Afin de pouvoir commander efficacement les moteurs Gauche et Droite, nous réalisons des essais et mesures. Un des objectifs serait d'avoir des vitesses de roues homocinétiques pour une avancée en ligne droite parfaitement rectiligne par exemple. Ainsi, nous réalisons un premier montage qui permettra d'établir en fonction de la consigne d'envoi, la vitesse de sortie de chacune des roues.

Voici le montage d'essai moteur :

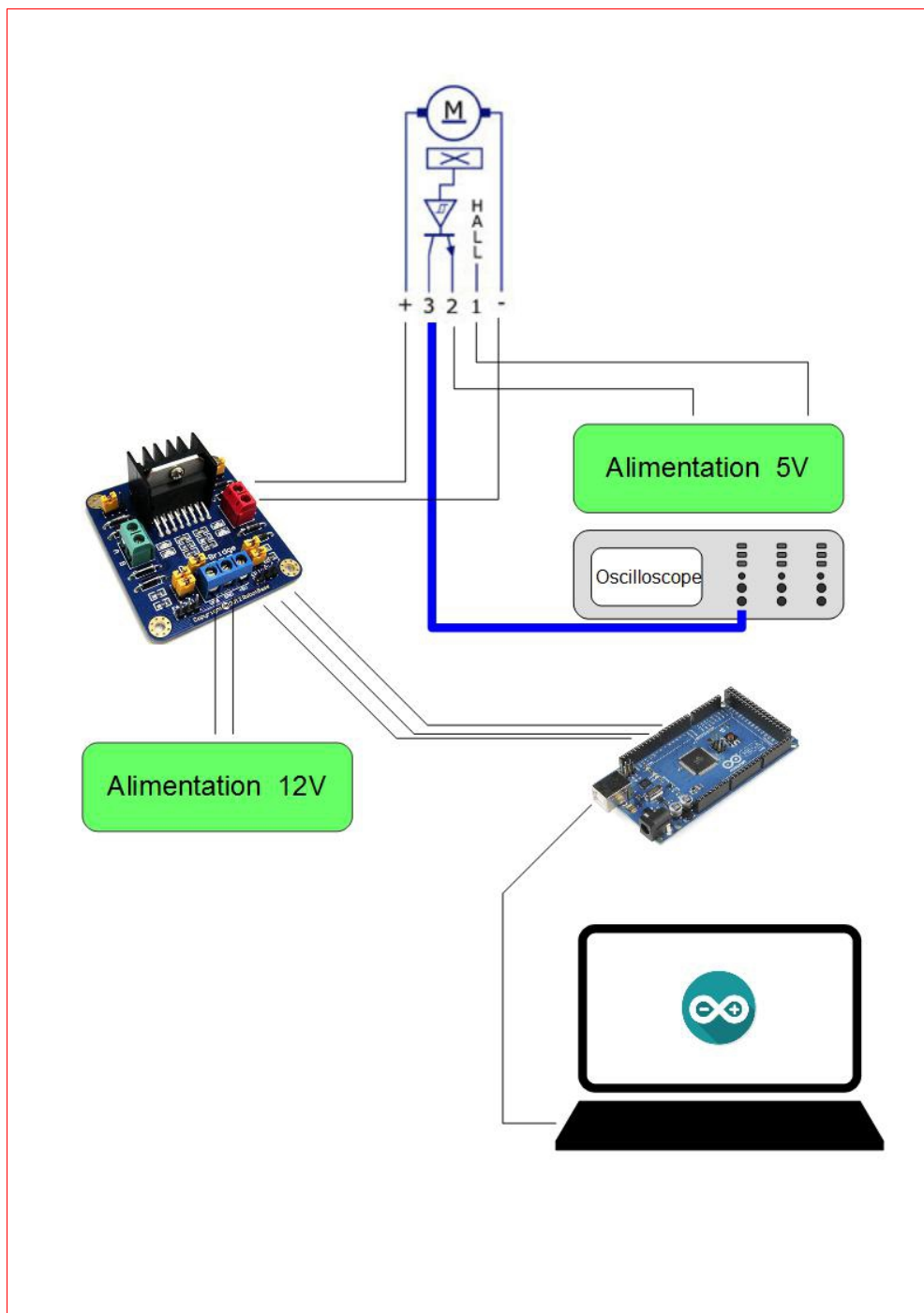


Figure 1.1

Nous connectons le moteur G ou D (Gauche ou Droit) au module 1298 lui-même relié et piloté par notre carte Arduino MEGA. Le module de contrôle moteur est alimenté en 12 V et prend en charge l'alimentation des roues. L'encodeur, quant à lui, est alimenté en 5 V de manière externe. Nous récupérons le signal de l'encodeur sur l'oscilloscope. Cette mesure nous permettra de trouver, pour une consigne d'Arduino donnée, la valeur de la vitesse de sortie de la roue G ou D.

NB : Nous réalisons cet essai pour les deux roues et ce, dans chacun des sens de rotation.

Nous savons que :

3 impulsions de l'encodeur correspondent à un tour du rotor (hors réducteur).

Ainsi, en mesurant la période de plusieurs périodes d'impulsions et en divisant par ce nombre de périodes, nous obtenons la période d'une impulsion avec une relative précision expérimentale (mesure réalisée grâce aux curseurs de l'oscilloscope). En multipliant cette période par 3, nous obtenons une valeur donc en sec/tour.

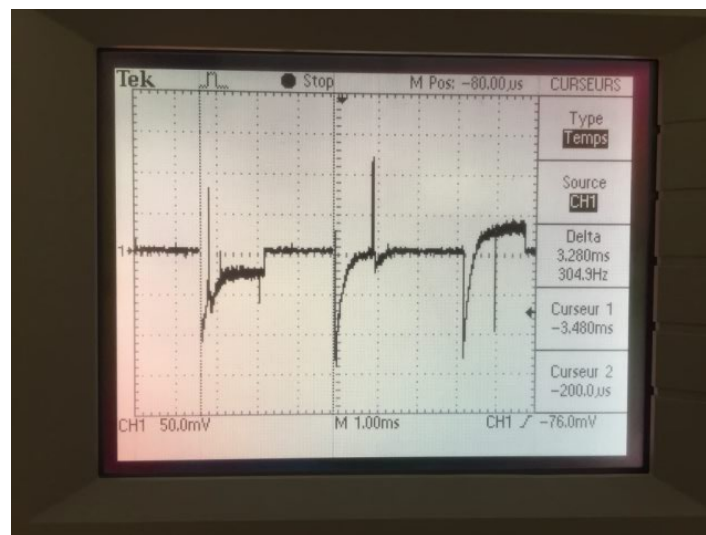


Figure 1.2

Nous récupérons les valeurs de période pour chaque consigne variant : **[30 – 250] par pas de 20.**

En prenant par la suite l'inverse, nous obtenons une valeur exprimée en tours/sec. Puis, en passant des tours/sec du rotor en tours/sec à la sortie du réducteur en divisant par 29,75 (valeur correspond au rapport de réduction du réducteur), et par $2 \times \pi \times R$, nous obtenons la vitesse de la roue en m/s.

Nous obtenons pour le **moteur Gauche** dans le **sens horaire** la courbe suivante :

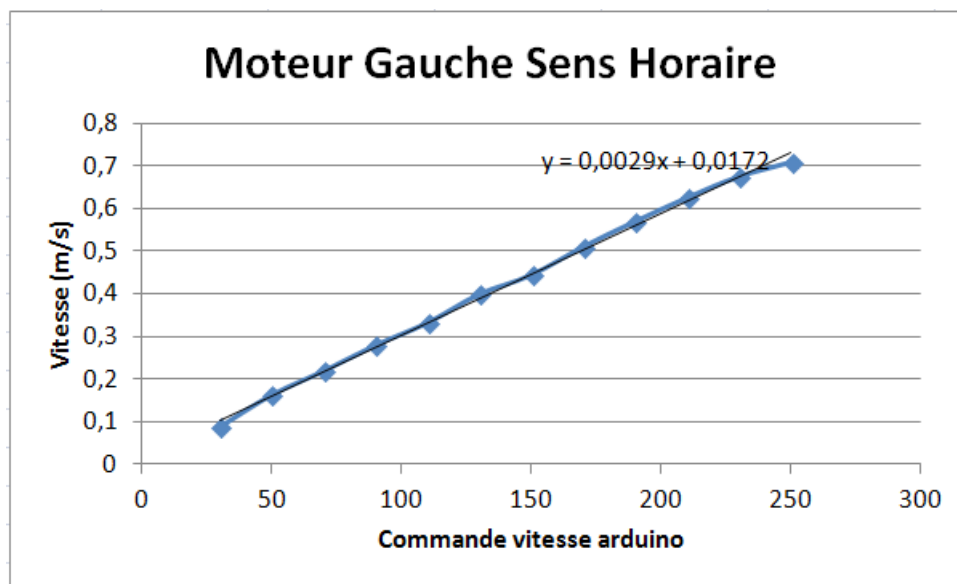


Figure 1.3

L'équation de la courbe de tendance à fonction affine est : $y = 0,002850x + 0,01718$

→ La courbe de tendance est plus que fidèle à notre courbe expérimentale.

Moteur Gauche dans le **sens anti horaire** :

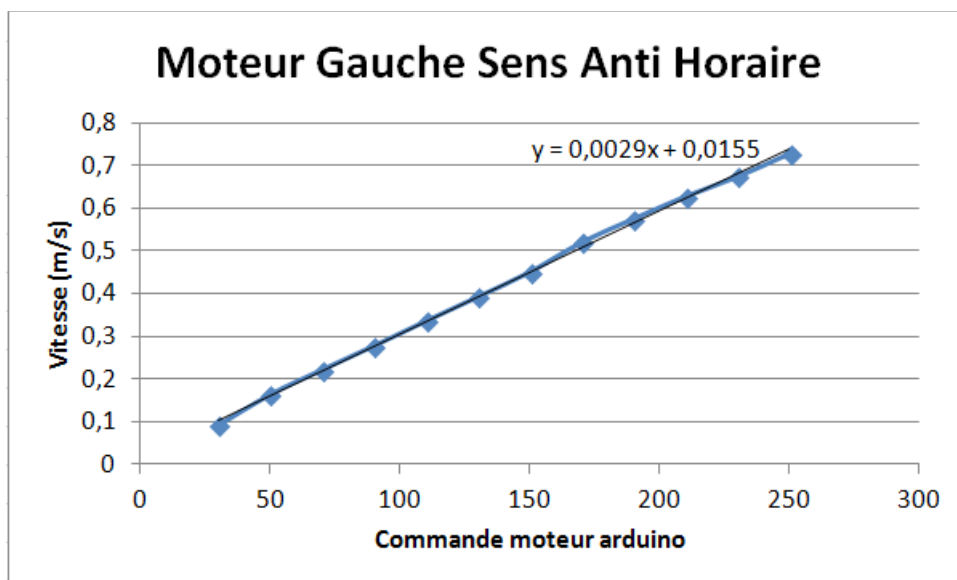


Figure 1.4

L'équation de la courbe de tendance à fonction affine est : $y = 0,002893x + 0,01551$

Moteur Droit dans le **sens horaire** :

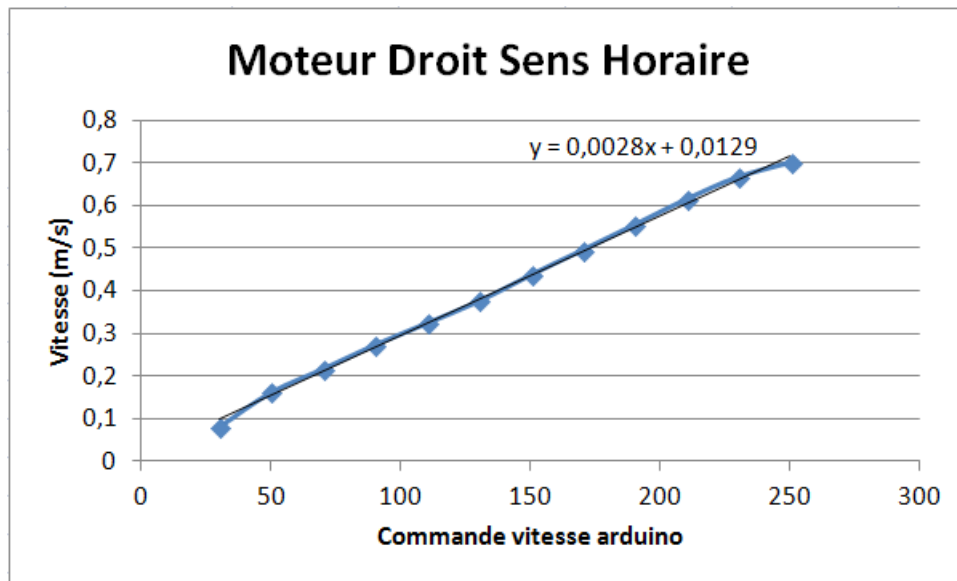


Figure 1.5

L'équation de la courbe de tendance à fonction affine est : $y = 0,002816x + 0,01290$

Moteur Droit dans le **sens anti horaire** :

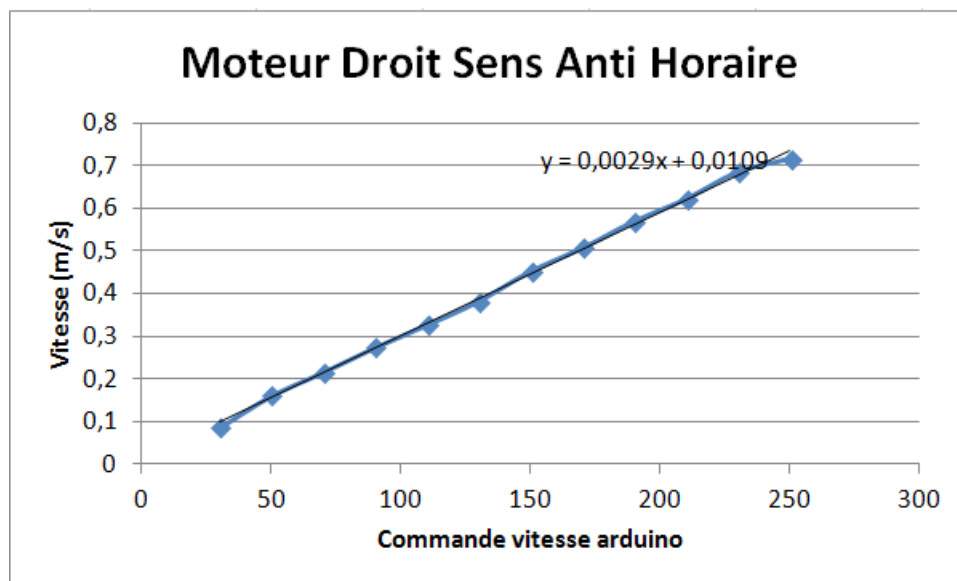


Figure 1.6

L'équation de la courbe de tendance à fonction affine est : $y = 0,002893x + 0,01085$

En ajoutant une courbe de tendance, et en comparant ses coefficients, nous pouvons remarquer quelques légères variations qui peuvent être prises en compte pour apporter un éventuel offset afin de s'assurer du caractère homocinétique des roues. Cet offset est en partie dû au fait que ces relevés soient expérimentaux et donc qu'il serait quasiment impossible d'acquérir à chaque fois les mêmes valeurs au fur et à mesure des mêmes essais.

Reprenons ces résultats :

$$\text{MotG_s_h} : y = 0,002850x + 0,01718$$

$$\text{MotG_s_antih} : y = 0,002893x + 0,01551$$

$$\text{MotD_s_h} : y = 0,002816x + 0,01290$$

$$\text{MotD_s_antih} : y = 0,002893x + 0,01085$$

Impact du sens :

Nous pouvons observer que les coefficients directeurs des droites des moteurs en sens anti horaire sont légèrement plus élevés que ceux en sens horaire :

$$\rightarrow \text{MotG} : 0,002893 - 0,002850 = \mathbf{0,000043} \text{ m/(s.consigne)}$$

$$\rightarrow \text{MotD} : 0,002893 - 0,002816 = \mathbf{0,000077} \text{ m/(s.consigne)}$$

En revanche, les ordonnées à l'origine des droites des moteurs en sens anti horaire sont légèrement plus faibles que ceux en sens horaire :

$$\rightarrow \text{MotG} : 0,01718 - 0,01551 = \mathbf{0,00167} \text{ m/(s.consigne)}$$

$$\rightarrow \text{MotD} : 0,01290 - 0,01085 = \mathbf{0,00205} \text{ m/(s.consigne)}$$

Impact du moteur :

Quant aux différences entre les moteurs à sens identique :

Coefficients directeurs :

$$\rightarrow \text{Sens Horaire} : 0,002850 - 0,002816 = \mathbf{0,000034} \text{ m/(s.consigne)}$$

$$\rightarrow \text{Sens anti Horaire} : 0,002893 - 0,002893 = \mathbf{0} \text{ m/(s.consigne)}$$

Ordonnées à l'origine :

$$\rightarrow \text{Sens Horaire} : 0,01718 - 0,01290 = \mathbf{0,00428} \text{ m/(s.consigne)}$$

$$\rightarrow \text{Sens anti Horaire} : 0,01551 - 0,01085 = \mathbf{0,00466} \text{ m/(s.consigne)}$$

Ces différences nous paraissent trop faibles pour être prises en compte dans la commande des moteurs. Aussi, nous supposons les moteurs homocinétiques dans les deux sens et que le sens n'influe pas sur la consigne en sortie, autrement dit, qu'aucun frottement n'est retenu dans l'envoi de consignes moteurs.

Pilotage des moteurs :

Voici une image du module, le module I298 dual h-bridge motor driver :

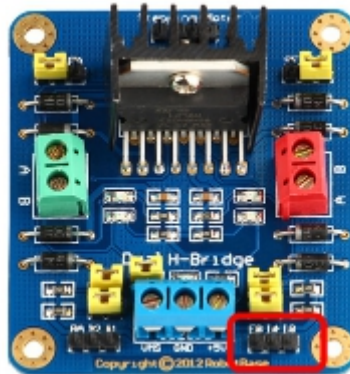


figure 1.7

C'est à partir des pins EA, I2, I1 pour le moteur de Gauche et EB, I4 et I3 pour le moteur Droit, que nous changeons le sens de rotation des moteurs donc des roues :

Par exemple, pour le moteur Droit (qui correspond à la zone rectangulaire rouge sur la *figure 1.7*)

EB	I4 (ou I2)	I3 (ou I1)	Sens de rotation Moteur
Consigne Vitesse	0 (= low)	0 (= low)	Pas d'entraînement Moteur (sans blocage rotor)
	0 (= low)	1 (= high)	Sens Anti horaire
	1 (= high)	0 (= low)	Sens Horaire
	1 (= high)	1 (= high)	Blocage Moteur

Programmation d'avance des moteurs sous Arduino :

```

int pinI1=2;           //define I1 port Moteur Droit
int pinI2=3;           //define I2 port Moteur Droit
int Dspeedpin=5;       //define EA(PWM speed regulation)port

int vitesse=100;       // vitesse moteurs

int pinI3=8;           //define I3 port Moteur Gauche
int pinI4=9;           //define I4 port Moteur Gauche
int Gspeedpin=11;      //define EB(PWM speed regulation)port

float encoderValue=0.0;

void count(void);
  
```

```
void setup()
{
    Serial.begin(9600);
    pinMode(21,INPUT); //pin de l'encodeur

    attachInterrupt(digitalPinToInterrupt(21),count,FALLING); //interruption pour l'encodeur

    encoderValue=0;
    pinMode(pinI1,OUTPUT); //define this port as output
    pinMode(pinI2,OUTPUT);
    pinMode(Dspeedpin,OUTPUT);
    pinMode(pinI3,OUTPUT); //define this port as output
    pinMode(pinI4,OUTPUT);
    pinMode(Gspeedpin,OUTPUT);
    digitalWrite(pinI1,HIGH); // DC motor rotates clockwise
    digitalWrite(pinI2,HIGH);
    digitalWrite(pinI3,HIGH); // DC motor rotates clockwise
    digitalWrite(pinI4,LOW);
    analogWrite(Gspeedpin,vitesse); // Ordre d'avance du moteur Gauche
}
```

```
void loop()
{
    Serial.print("Encoder Value=");
    Serial.println(encoderValue);
    if(encoderValue>=143) // valeur de l'encodeur pour realiser un virage de 90 degrees
    {
        digitalWrite(pinI4,HIGH);
    }
}
```

```
void count()
{
    encoderValue++;
}
```

Test Virages :

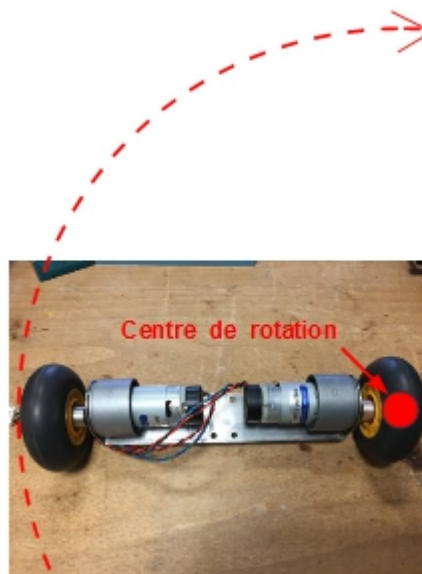
Récupération du nombre d'impulsions de l'encodeur dans Arduino :

Nous utilisons les interruptions sur la pin où l'encodeur est branché. Il en résulte une interruption à chaque impulsion de l'encodeur. C'est en comptant ce nombre d'interruptions que nous avons l'information du nombre de tours du rotor puis de la roue en passant par le rapport de réduction. Ainsi, nous trouvons que pour effectuer un tour de roue, environ **90 impulsions** sont nécessaires (89,25).

Données Utiles :

- Entre-axe des roues :
→ **22,3 cm** avec un centre à 11,15 cm.
- Diamètre des roues :
→ **7,23 cm**

Virage de 90° de type n°1 :



Périmètre du cercle engendré : $2 \times \pi \times 22,3 = \mathbf{140,115 \text{ cm}}$

→ $140,115 / 4 = \mathbf{35,028 \text{ cm}}$ ce qui correspond à un quart de cercle donc à un virage de 90°.

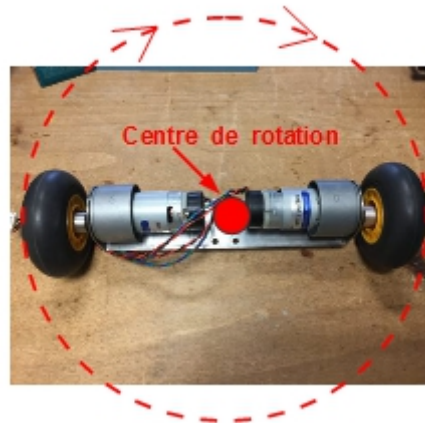
De plus, $\pi \times 7,23 = \mathbf{22,713 \text{ cm}}$ Donc $35,028 / 22,713 = \mathbf{1,674 \text{ tours de roue}}$

Sachant que 1 tour de moteur ↔ 3 impulsions et que 1 tour de roue ↔ 89,25 impulsions

Nous trouvons : $89,25 \times 1,674 = 149,404 \text{ impulsions} \approx 149 \text{ ou } 150 \text{ impulsions.}$

Expérimentalement et sous charge, nous remarquons un offset du nombre d'impulsions. En effet, il provient du fait que la condition d'arrêt du moteur est : ≥ 150 donc si l'Arduino ne détecte pas la 150 ième impulsion au moment de la lecture, il s'arrêtera à la suivante impulsion détectée. Nous stoppons alors le moteur de manière immédiate dès que la première impulsion lue par l'Arduino est supérieure ou égale à 150. Si nous avions mis une condition : $=150$, l'impulsion pourrait ne pas être détectée entraînant de ce fait le ou les moteurs sans arrêt.

Virage de 90° de type n°2 :



Périmètre du cercle engendré : $2 \times \pi \times 11,15 = \mathbf{70,057 \text{ cm}}$

→ $70,057 / 4 = \mathbf{17,514 \text{ cm}}$ ce qui correspond au quart de cercle donc à un virage de 90°.

De plus, $\pi \times 7,23 = \mathbf{22,713 \text{ cm}}$ Donc $17,514 / 22,713 = \mathbf{0,771 \text{ tours de roue}}$

Sachant que 1 tour de moteur ↔ 3 impulsions et que 1 tour de roue ↔ 89,25 impulsions

Nous trouvons : $89,25 \times 0,771 = 68,814 \text{ impulsions} \approx 69 \text{ impulsions pour chaque moteur.}$

Expérimentalement et sous charge, nous remarquons un offset du nombre d'impulsions. En effet, il provient du fait que la condition d'arrêt du moteur est : ≥ 69 donc si l'Arduino ne détecte pas la 69 ième impulsion au moment de la lecture, il puisse quand même s'arrêter. Nous stoppons alors le moteur de manière immédiate dès que la première impulsion lue par l'Arduino est supérieure ou égale à 69.

Blocs de mouvement :

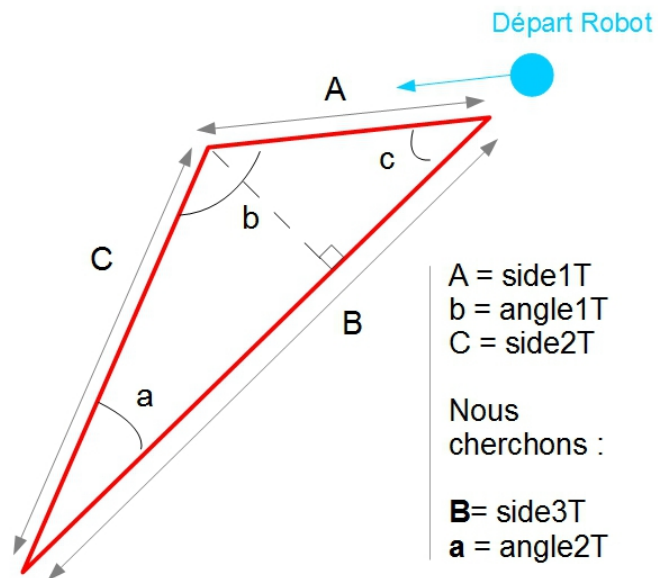
Rappel :

–1 tour de roue \leftrightarrow 89,25 impulsions de l'encodeur

– $\pi \times D = 22,713 \text{ cm}$ \rightarrow distance parcourue par la roue en un tour

Voici la liste des fonctions que nous souhaitons implémenter au début du projet :

Fonctions	Paramètres	Steps
Rectangle() :	sideXR - directionR - sideYR <i>sideXR & sideYR :</i> - Longueurs des côtés du rectange <i>directionR :</i> - Sens du rectangle. Ce paramètre définit le choix du premier virage.	→Avancer(sideXR) →Tourner(90, directionR) →Avancer(sideYR) →Tourner(90, directionR) →REPEAT →STOP
Triangle() :	side1T – directionT – angle1T – side2T <i>side1T & side2T :</i> - Longueurs des côtés du triangle <i>angle1T :</i> - Premier angle du triangle à effectuer ANGLE et DISTANCE sont calculés automatiquement	→Avancer(side1T) →Tourner(angle1T, directionT) →Avancer(side2T) →Tourner(ANGLE, directionT) →Avancer(DISTANCE) → STOP



Loi des Cosinus : $B^2 = A^2 + C^2 - 2 \cdot A \cdot C \cdot \cos(b)$

$$\rightarrow B = \sqrt{A^2 + C^2 - 2 \cdot A \cdot C \cdot \cos(b)}$$

Loi des Sinus : $A / \sin(a) = B / \sin(b) = C / \sin(c)$

$$\rightarrow a = \arcsin(A \cdot \sin(b) / B)$$

Circle() :	<p>radiusC - directionC</p> <p>DirectionC : - représente un booléen qui définit laquelle des deux roues dessinera le cercle externe ↔ cercle à gauche / droite</p> <p>RadiusC : - représente la valeur du rayon du cercle externe à réaliser : D1</p>	<p>→AvancerMot1(vitesseC1, distanceC1) & AvancerMot2(vitesseC2, distanceC2)</p>
------------	---	---

Distance à parcourir pour le cercle 1 : $\pi \cdot D1$

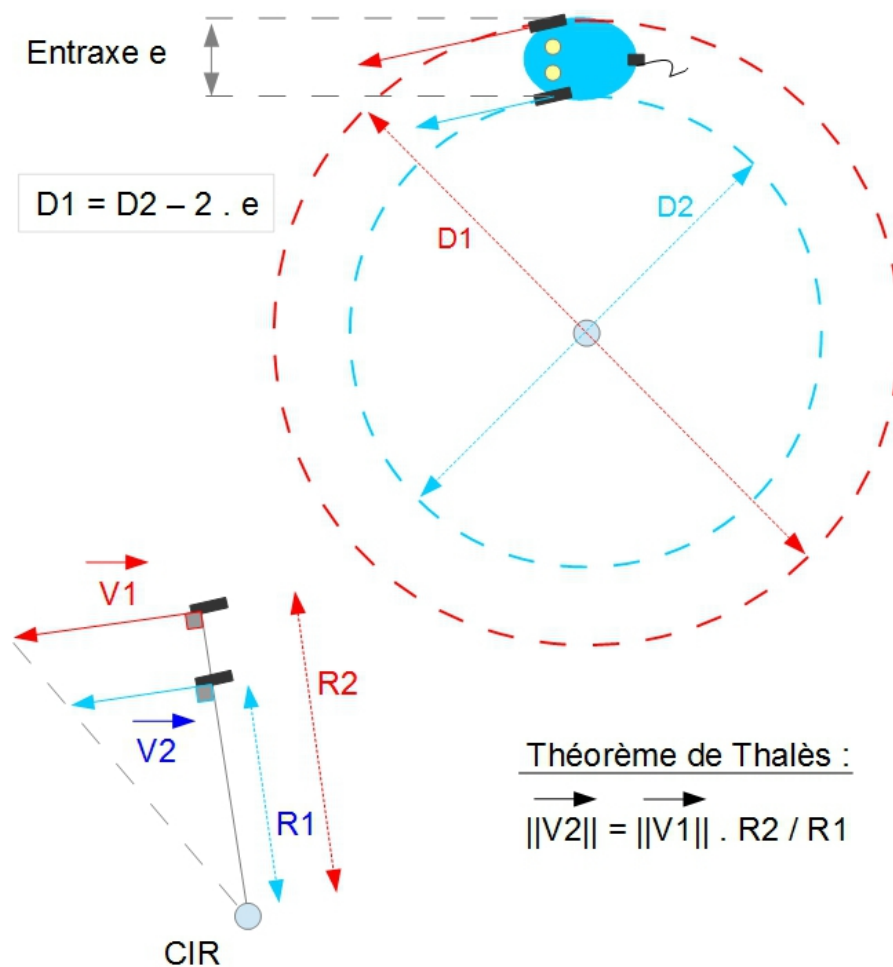
→ Si consigne de temps $T1 \rightarrow V1 = \pi \cdot D1 / T1$

→ Si consigne de vitesse $V1 \rightarrow V1$

Distance à parcourir pour le cercle 2 : $\pi \cdot D2$

→ Si consigne de temps $T1 \rightarrow V2 = \pi \cdot D2 / T1$

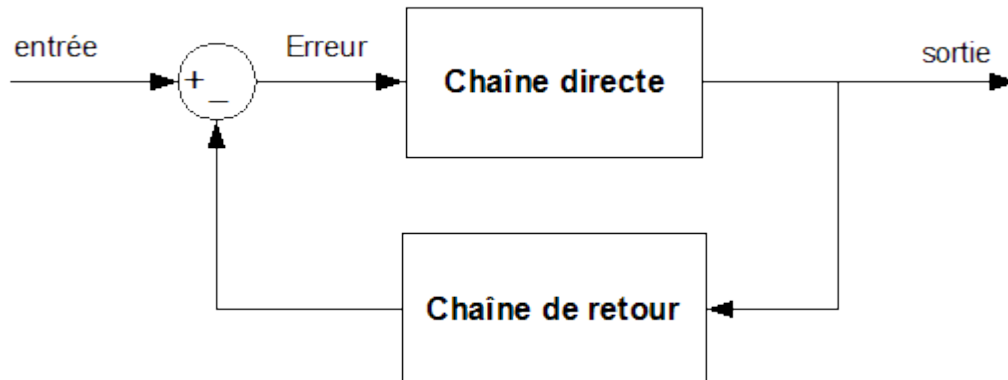
→ Si consigne de vitesse $V1 \rightarrow V2 = V1 \cdot R2 / R1$



	distanceSL	Avancer(distanceSL)
StraightLine() :	DirectionSL : - représente la longueur de la ligne à parcourir	

RoundTrip() :	DistanceSL - directionRT DirectionSL : - représente la longueur de la ligne à parcourir	→Avancer(distanceSL) →Tourner(180°, directionRT) →Avancer(distanceSL)
LeftTurn() :	optionTurnG - angleLT optionTurnG : - représente un booléen permettant de fixer le centre de rotation pour le virage : semi entraxe des roues / centre de la roue gauche	→Tourner(optionsTurn, angleLT)
RightTurn() :	optionTurnD - angleRT optionTurnD : - représente un booléen permettant de fixer le centre de rotation pour le virage : semi entraxe des roues / centre de la roue droite	→Tourner(optionsTurn, angleRT) → STOP
Stop() :	distanceS distanceS : - représente la distance d'arrêt du robot : > ou = à 0	→Stop()

Pilotage des moteurs en boucle fermée :



– Intervalle possible de consigne vitesse : $[0 - 2]$ (tours de roues/s) $\leftrightarrow [0 - 0,7]$ (m/s roue)

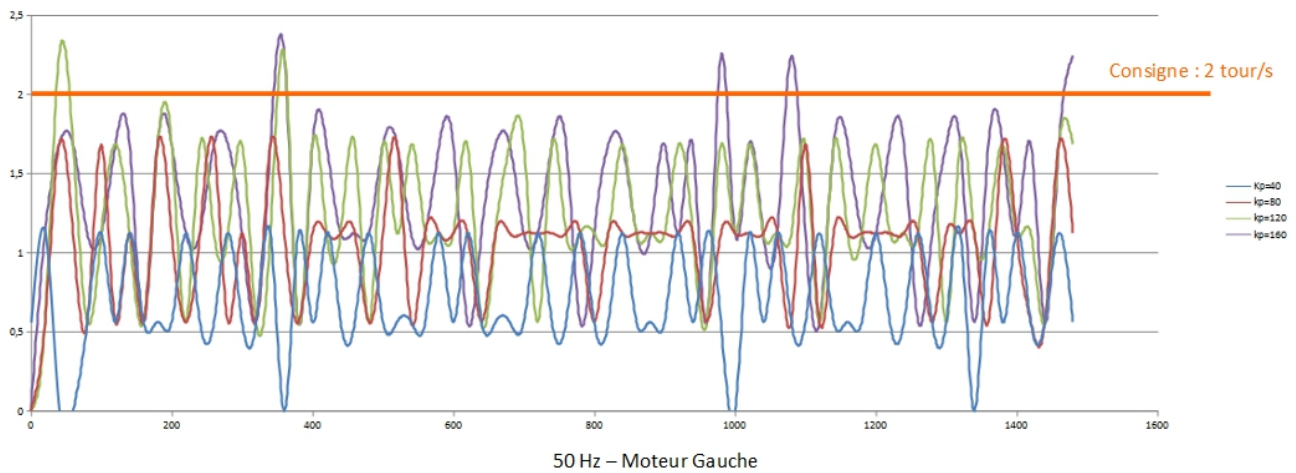
– Intervalle possible de fréquence: $[0 - 281]$ (Hz ou ImpulsionsEncodeur/s) $\leftrightarrow [0 - 93,6]$ (tourMoteur/s)

L'échantillonnage est une technique utilisée pour le traitement du signal et permet de dire à l'Arduino de faire des calculs tous les intervalles donnés afin de ne pas saturer la puce de calculs inutiles.

Ainsi, au lieu de faire des mesures continues et de ralentir le microprocesseur, celui-ci effectuera ses calculs, par exemple, toutes les 50ms. C'est la base de l'asservissement. En effet, toutes les 50ms, l'Arduino refera les calculs et nous dira si oui ou non, nous nous sommes éloignés de la consigne d'entrée. Cela nous permettra alors de corriger la sortie.

ASSERVISSEMENT EN **VITESSE**

Nous avons œuvré dans un premier temps pour réaliser un asservissement basé sur la consigne de vitesse. Cependant, les résultats obtenus furent malgré la mise en place de correcteur Proportionnel et/ou Intégrateur, inexploitable. Ce manque d'information s'explique selon nous, par le manque d'impulsions d'encodeur de nos moteurs \rightarrow 3 impulsions pour un tour offrent peu d'informations contrairement à d'autres moteurs où par exemple l'encodeur propose plus de 40 impulsions pour un tour du rotor.

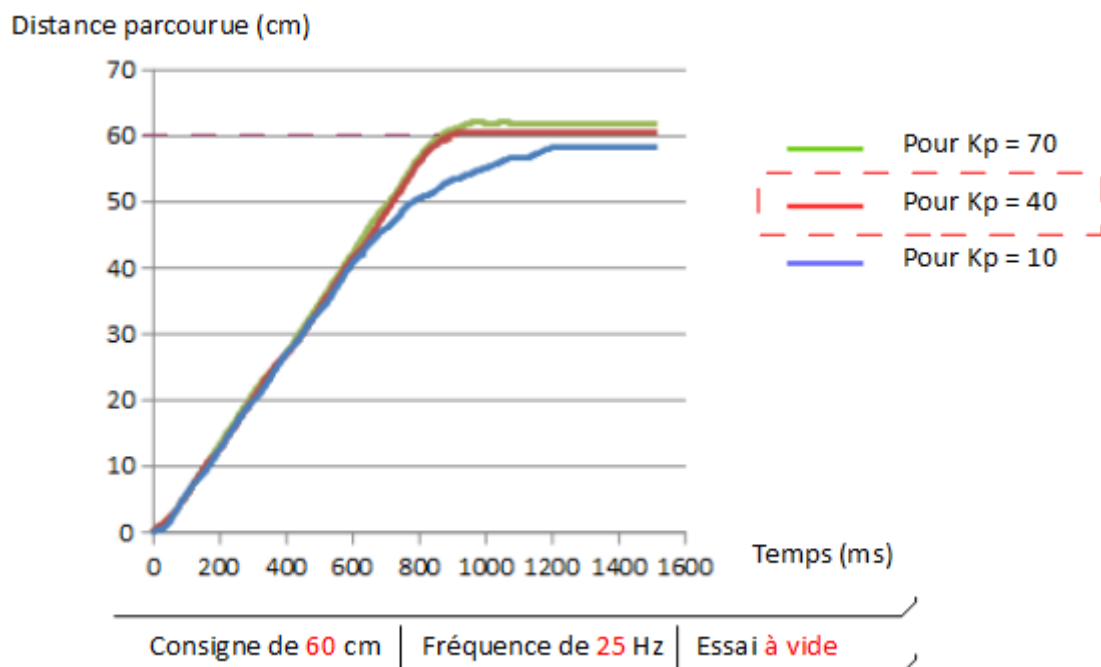


ASSERVISSEMENT EN **POSITION**

Aussi, nous avons envisagé un asservissement de position.

Correcteur Proportionnel P

Dans un premier temps, nous voudrions analyser l'impact dû à l'utilisation d'un correcteur P (Proportionnel) sur une consigne de distance. Voici les différentes courbes obtenues pour le moteur Gauche pour 3 valeurs différentes du K_p .

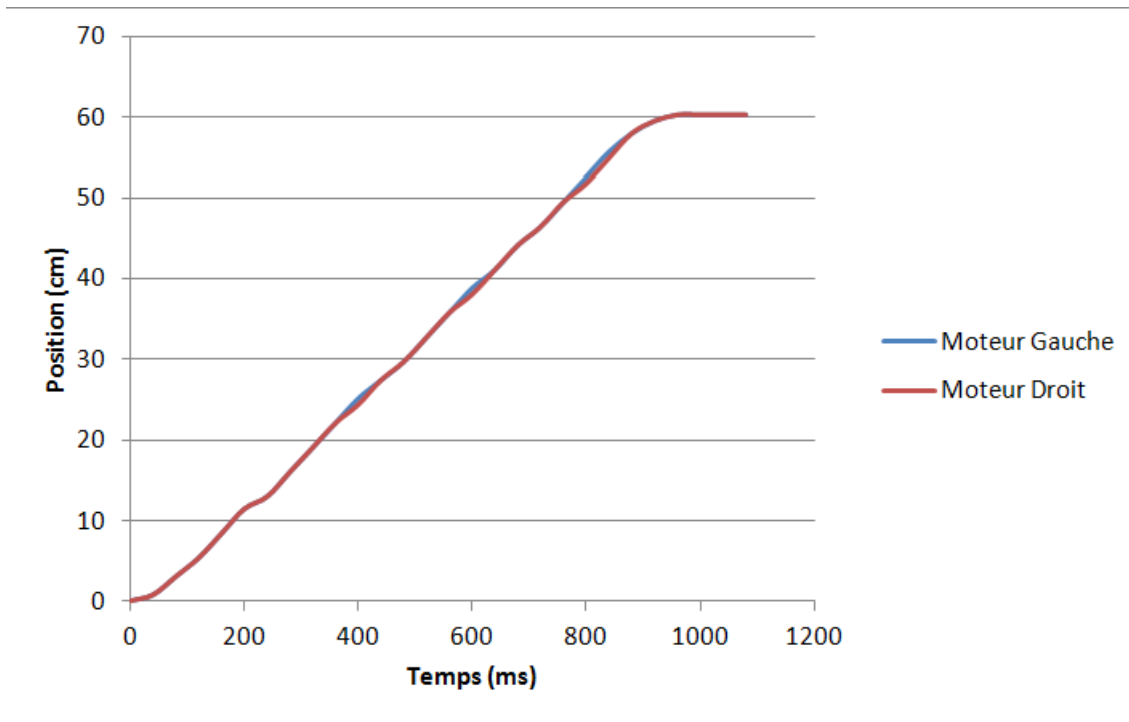


Pour une valeur intermédiaire de $K_p = 40$, nous remarquons que notre système tend vers la consigne souhaitée.

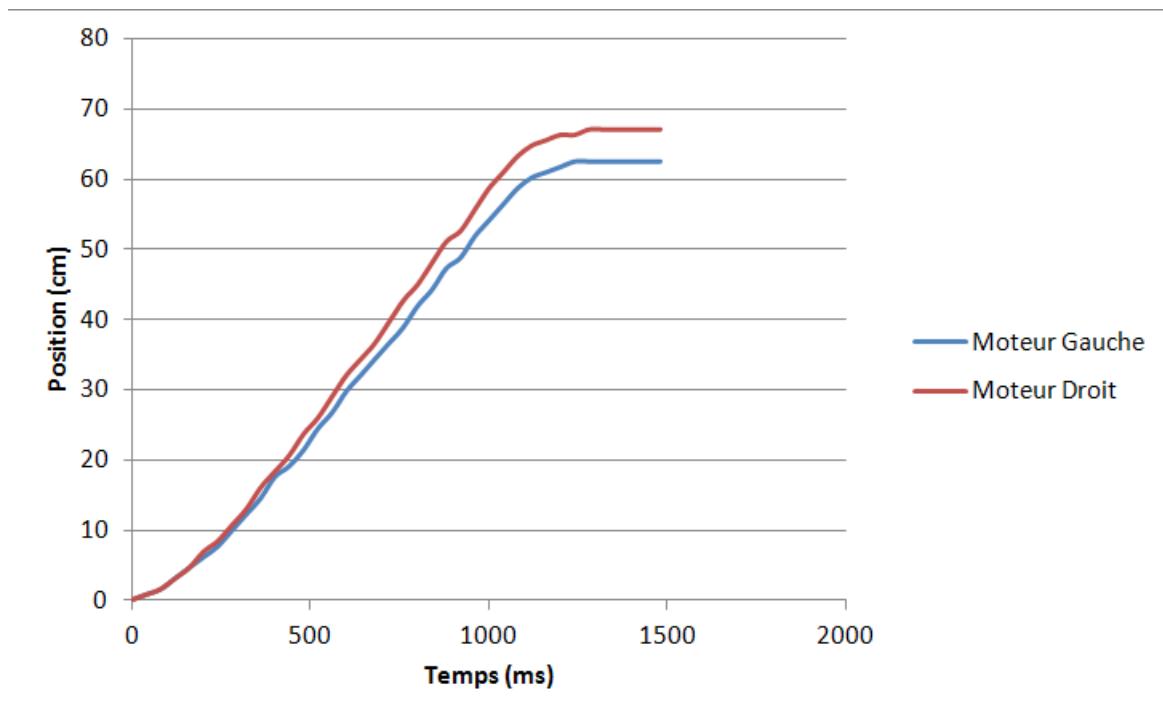
Remarque : Les tests ont été réalisés pour des consignes de 60, 90, 120, 150. Lors de ces essais,

la réponse du système est à chaque fois précise pour une valeur de $K_p = 40$.

En comparant nos deux moteurs, il en résulte ce graphique :



Ainsi, à vide, nos deux moteurs semblent homocinétiques. Malheureusement, ce n'est plus le cas en charge. En effet, nous observons une divergence entre les courbes des deux moteurs et ce malgré notre asservissement en position :

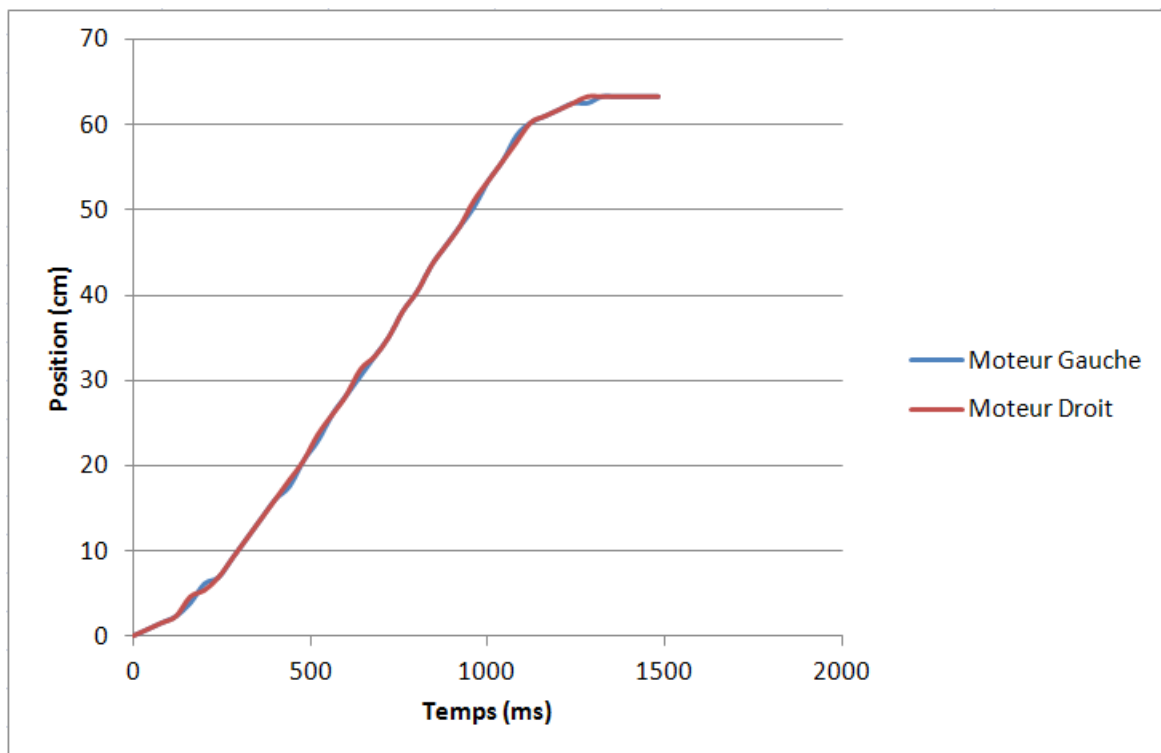


Remarque : la valeur charge lors des essais fut d'environ 2 Kg. La consigne était de 60 cm avec

une commande maximale Arduino de 255, une fréquence d'échantillonnage de 25Hz et une valeur de $k_p = 40$.

Afin de remédier à ces écarts, nous rajoutons cette brique de code : Si l'un des moteurs a une plus grande erreur que l'autre, il s'arrêtera le temps pour le second moteur de rattraper son retard et vice-versa.

```
if(erreur_Mot_G > erreur_Mot_D)
{
    cmd_Mot_D= 0;
} else if (erreur_Mot_G < erreur_Mot_D)
{
    cmd_Mot_G= 0;
}
```

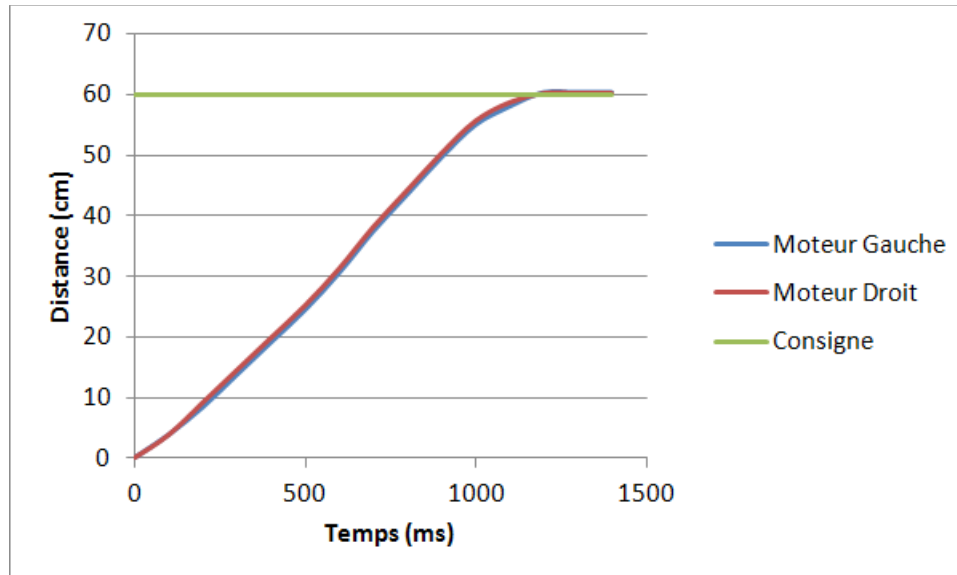


L'observation expérimentale en est que meilleure. En effet notre écart réduit mais cette amélioration reste insuffisante.

Après mûre réflexion et plusieurs essais, nous avons remarqué que **la répartition du poids** jouait un rôle sur la trajectoire. De plus, **la roue folle** a un impact direct sur la direction. Nous avons par conséquent retiré cette roue et nous nous sommes aperçus que la trajectoire n'était plus aléatoire et bien plus conforme à la consigne. Dans ces nouvelles conditions d'expérimentation, nous analysons les nouvelles courbes de l'asservissement en position puis de l'asservissement en vitesse

→ Asservissement en position SANS ROUE FOLLE

En utilisant un asservissement de POSITION, nous trouvons un gain statique de $60.3154 / 60 = 1,00525$ pour le moteur Gauche et le moteur Droit avec une valeur affinée de **Kpp de 18** et une **fréquence d'échantillonnage de 10 Hz**. Nous garderons ce choix de paramètre afin d'avoir un système des plus précis en position en charge. Cependant, il est important de garder à l'esprit qu'une élévation du poids du robot pourrait influencer notre valeur de Kpp.



Cf Annexe

Asservissement en vitesse : Abscisses : Temps – Ordonnées : Nbr tours/s Roue

On observe que pour une valeur de Kvp intermédiaire de 90 à 25 Hz, nous obtenons le meilleur graphique de réponse.

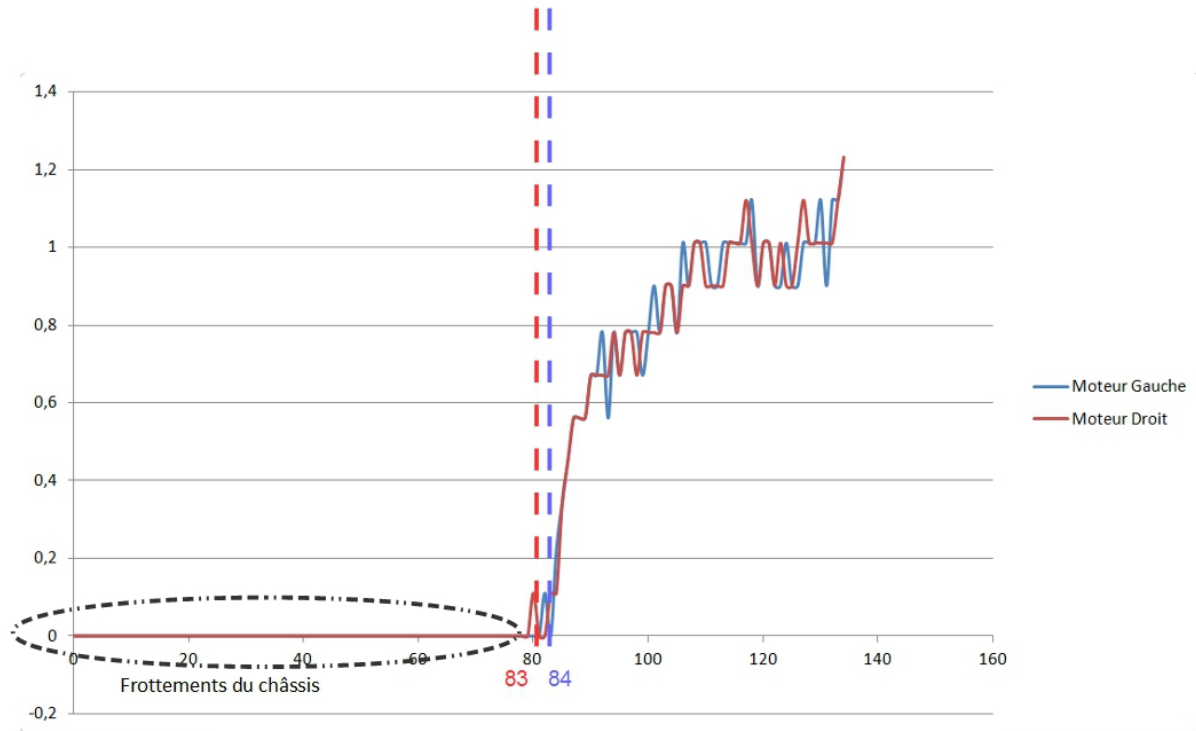
Cf Annexe

D'autres essais furent effectués en utilisant un correcteur Intégrateur mais les résultants restèrent insuffisants.

Par la suite, nous désirons quantifier la valeur minimale de consigne vitesse à envoyer pour entraîner la rotation des roues. Il s'agit de la consigne minimale de compensation du poids et des frottements secs.

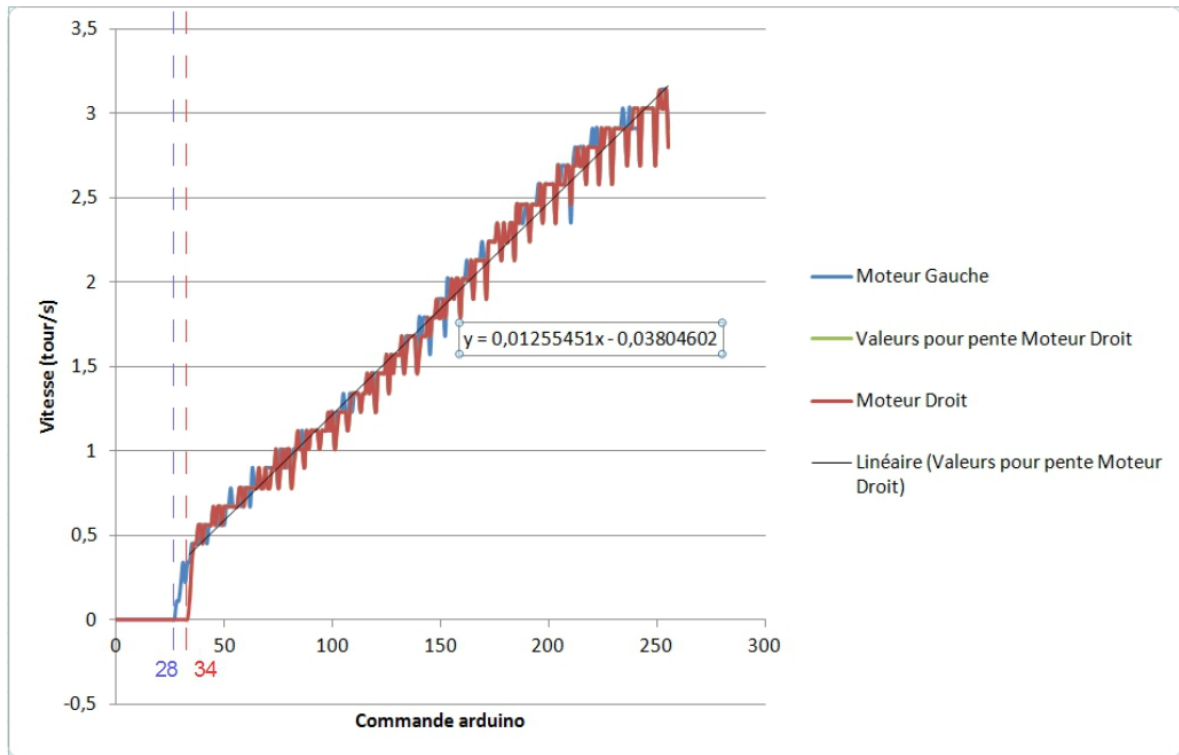
Voici les résultats obtenus pour :

- Coefficient Frottements secs en BO
- Sans roue folle**
- En charge (env. 2 Kg)



Voici les résultats obtenus pour :

- Coefficient Frottements secs en BO
- Sans roue folle
- A vide



Bilan :

–Utilisation du courant

Pas forcément utile pour l'asservissement en vitesse. De plus, la récupération de celui-ci reste très délicate sans documentation précise de la carte.

–Asservissement en vitesse

Cet asservissement s'est vu mis de côté étant donné les résultats et courbes obtenus. En effet, le fait de n'avoir que 3 impulsions pour un tour de rotor moteur représente un réel frein au calcul de vitesse et donc à la précision ainsi qu'à la stabilité de l'asservissement. Par conséquent, la fonction **Circle()** sera supprimée des blocs de mouvements.

–Asservissement en position

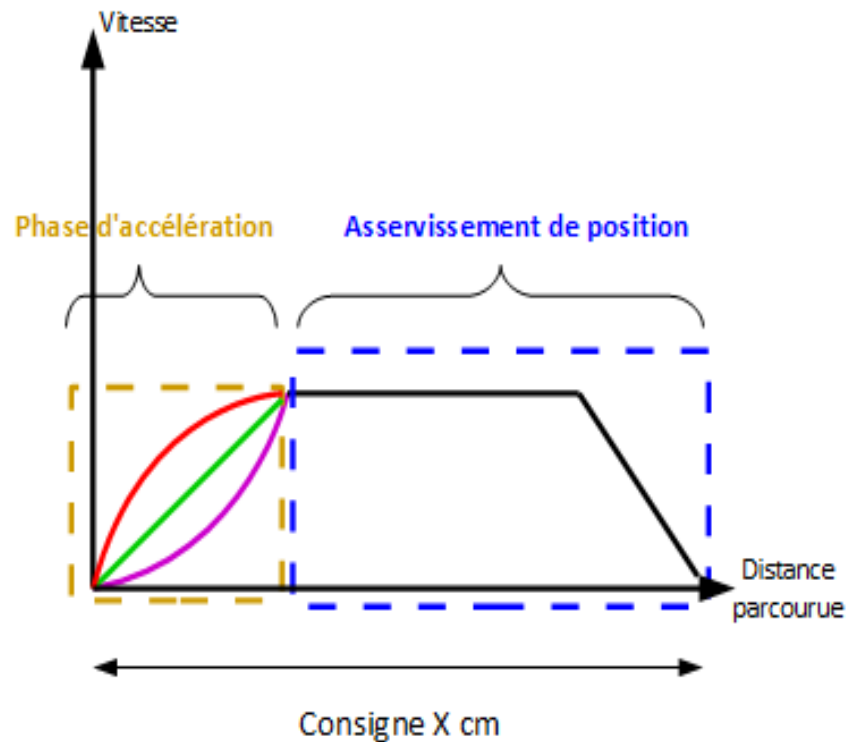
Notre choix s'est donc tourné vers un asservissement en position. A travers une pente d'accélération dans la phase initiale et une pente de décélération dans la phase finale du mouvement, couplées à un asservissement en position tout au long du trajet, nous assurerons un déplacement des plus précis possible.

→ Réalisation de la pente d'accélération :

Cette pente est utile lors de petits déplacements. Elle joue un rôle essentiel dans ce type de trajets car elle évitera un dépassement de la faible consigne appliquée. Elle est fonction du Poids du robot et du Déplacement désiré. La courbe ci-dessous lie la

consigne vitesse à appliquer en fonction du nombre d'impulsions de l'encodeur.

Poids	Consigne	Pente d'accélération
Faible	Petite	Lente
Elevé	Petite	Lente
Faible	Grande	Rapide
Elevé	Grande	Rapide

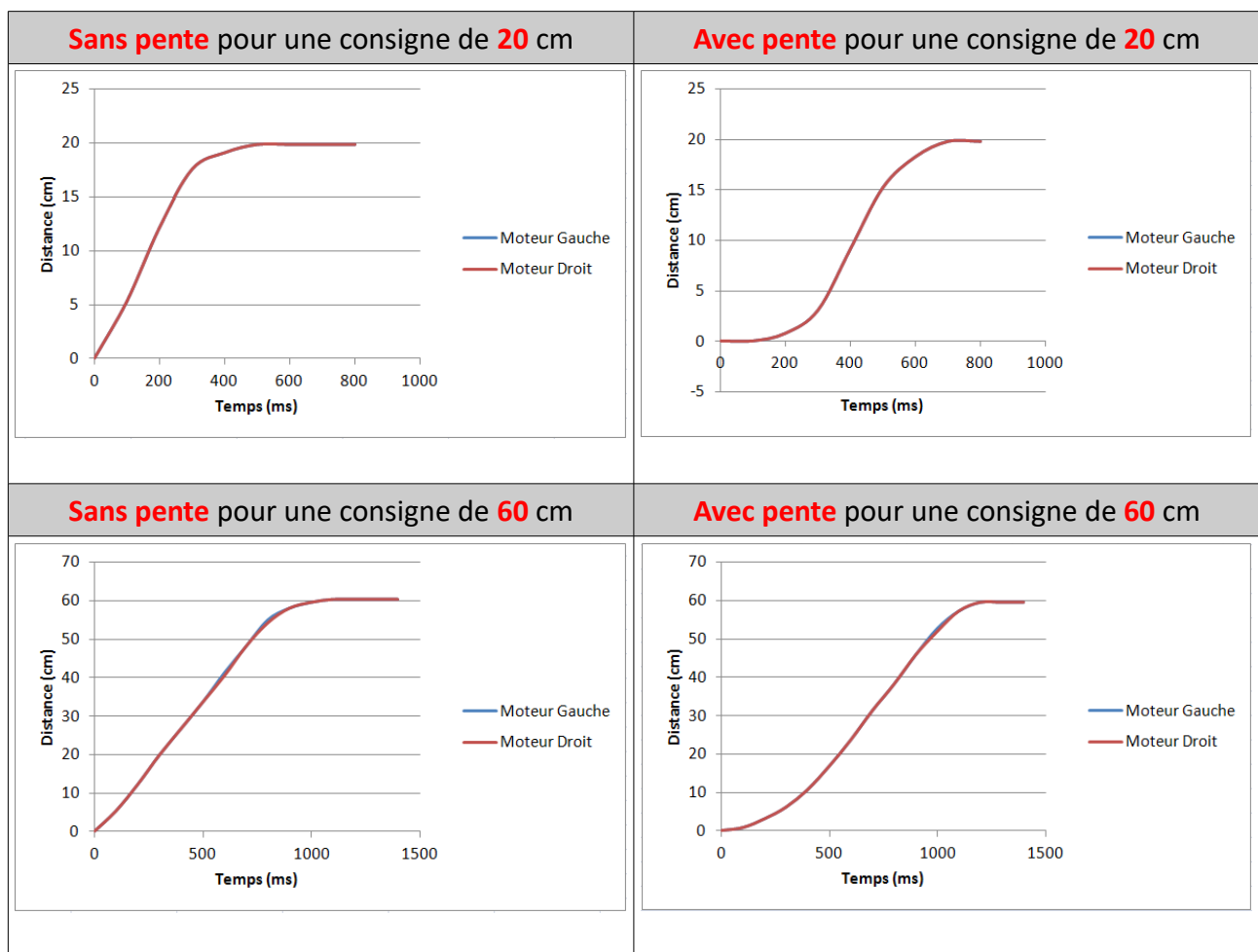


Il est possible de réaliser plusieurs pentes d'accélération comme montré sur le graphique ci-dessus. Trois pentes différentes qui peuvent agir différemment sur l'évolution du déplacement. Pour une fréquence d'échantillonnage de 10 Hz, voici les résultats obtenus afin de pouvoir observer la différence avec et sans pente d'accélération à petite et grande consigne.

Petit Déplacement sans Pente d'accélération	Petit Déplacement avec Pente d'accélération
0.0000:0.0000	0.0000:0.0000
5.3444:5.3444	0.7635:0.7635
12.2158:12.2158	3.0539:3.0539
17.5602:17.5602	9.1618:9.1618
19.0872:19.0872	15.2697:15.2697
19.8506:19.8506	18.3237:18.3237
19.8506:19.8506	19.8506:19.8506

Grand Déplacement sans Pente d'accélération	Grand Déplacement avec Pente d'accélération
0.0000:0.0000	0.0000:0.0000
5.3444:5.3444	0.7635:0.7635
12.2158:12.2158	3.0539:3.0539
19.8506:19.8506	6.1079:6.1079

26.7220:26.7220	10.6888:10.6888
33.5934:33.5934	16.7967:16.7967
41.2283:40.4648	23.6681:23.6681
48.0996:48.0996	31.3029:31.3029
54.9710:54.2075	38.1743:38.1743
58.0250:58.0250	45.8092:45.8092
59.5519:59.5519	52.6806:51.9171
60.3154:60.3154	57.2615:57.2615
60.3154:60.3154	59.5519:59.5519
60.3154:60.3154	59.5519:59.5519
60.3154:60.3154	59.5519:59.5519
60.3154:60.3154	59.5519:59.5519
60.3154:60.3154	59.5519:59.5519
60.3154:60.3154	59.5519:59.5519
60.3154:60.3154	59.5519:59.5519



Machine d'état sous Arduino

Afin de réaliser les différents mouvements proposés, nous avons fait le choix d'implémenter une machine d'état qui séquencerait les déplacements les uns à la suite des autres à l'aide notamment d'une variable partagée qui s'incrémenterait par le déplacement précédent pour lancer le suivant.

Nous avons dans un premier temps réalisé un déplacement en ligne droite puis effectué un virage de manière indépendante. Ensuite, nous avons lié ces deux mouvements afin d'obtenir une première séquence. Il demeure important de garder à l'esprit que les précisions des encodeurs ne nous permettent pas d'obtenir une position angulaire davantage précise.

→ **Tolérance quantifiée de 4°**

Au sein de ce code, nous retrouvons plusieurs parties et sous-parties qui dessinent l'architecture squelette de notre programme :

Les Constantes

- Constantes nécessaires aux différents calculs réalisés
- L'Initialisation de la machine d'état
- Les Informations de commande
- Les Moteurs
- Les prototypes des Fonctions

Le Setup()

- Déclarations des interruptions
- Initialisation des moteurs à l'instant initial

Le Loop()

- Lancement du timer
- Lancement de la machine d'état

La Machine d'état

- Case STRAIGHT_LINE
- Case TURN
- Case MAKE_L
- Case MAKE_ROUND_TRIP
- Case MAKE_TRIANGLE
- Case MAKE_RECTANGLE

Envoi valeur a Raspberry

–requestEvent

Récupération ordre Raspberry

–receiveData

Les compteurs

–Moteur Gauche

–Moteur Droit

Les déclarations des fonctions Principales :

–Turn()

–StraightLine()

L'asservissement

–Ligne droite

–Virage

–Pente d'accélération

Remarque : Chaque bloc de mouvements fait appel à deux fonctions principales qui sont : **StraightLine()** et **Turn()**.

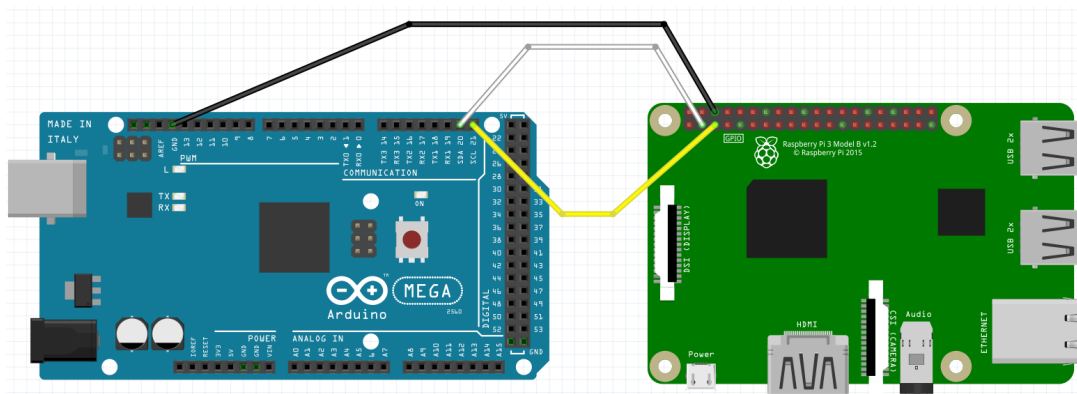
Le virage de type n°1 sera par ailleurs supprimé. En effet, dans ce cas précis, un seul moteur doit pouvoir entraîner en rotation tout le reste du robot pendant que l'autre moteur est à l'arrêt. Il est plus aisé de réaliser le virage de type n°2 où les deux moteurs tournent dans le même sens. Nous remarquons que les limitations matérielles telles que les encodeurs ou encore les moteurs représentent un frein pour les capacités du robot.

Connexion à l'Arduino :

Afin de pouvoir programmer le robot nous avons choisi d'utiliser une Raspberry Pi 3 (RPi). En lien avec l'Arduino cette dernière permet d'envoyer les commandes de l'utilisateur mais aussi de récupérer les informations du robot telles que les valeurs des capteurs de proximité. Elle permettra aussi l'utilisation de périphériques comme une webcam pour réaliser du streaming vidéo ou encore une manette pour piloter le robot.

Pour réaliser la communication entre l'Arduino et la RPi, nous avons fait le choix du protocole I2C offrant un temps de communication rapide tout en laissant les ports USB et Série libres pour d'autres applications telles que celles citées ci-dessus.

Le protocole utilise les ports SCL et SDA des deux cartes. La RPi sera définie comme maître et l'arduino l'esclave. Nous avons alors activé l'utilisation de I2C sur la Raspberry donnant accès à un périphérique « /dev/i2c-1 », nous verrons son utilisation par la suite.



Communication avec l'Arduino :

Pour tester cette communication nous avons pris un exemple sur internet. Pour la partie Arduino nous utilisons la bibliothèque Wire.h. L'Arduino étant défini comme esclave une adresse doit lui être associée avec Wire.begin(). Wire.onReceive() et Wire.onRequest() appelle les fonctions passées en paramètre lorsque, respectivement, l'esclave reçoit une donnée et lorsque qu'une requête est faite à l'esclave.

Ensuite Wire.available() permet de savoir si une donnée est présente sur le bus I2C, permettant ici de lire les données envoyées tant qu'il y a des données sur le bus. Pour lire ces données il faut utiliser Wire.read(). Pour envoyer des données au maître il faut utiliser Wire.write().

```

1 #include <Wire.h>
2
3 #define SLAVE_ADDRESS 0x12
4 int dataReceived = 0;
5
6 void setup() {
7   Serial.begin(9600);
8   Wire.begin(SLAVE_ADDRESS);
9   Wire.onReceive(receiveData);
10  Wire.onRequest(sendData);
11 }
12
13 void loop() {
14   delay(100);
15 }
16
17 void receiveData(int byteCount){
18   while(Wire.available()) {
19     dataReceived = Wire.read();
20     Serial.print("Donnee recue : ");
21     Serial.println(dataReceived);
22   }
23 }
24
25 void sendData() {
26   int envoi = random(50);
27   Wire.write(envoi);
28 }

```

Pour la Raspberry, l'exemple trouvé sur internet était fait avec Python et utilise la bibliothèque smbus qui permet de gérer les connexions I2C. Il faut d'abord créer un bus avec smbus.SMBus(). Puis pour écrire des données et lire les données il faut utiliser bus.write_byte() et bus.read_byte().

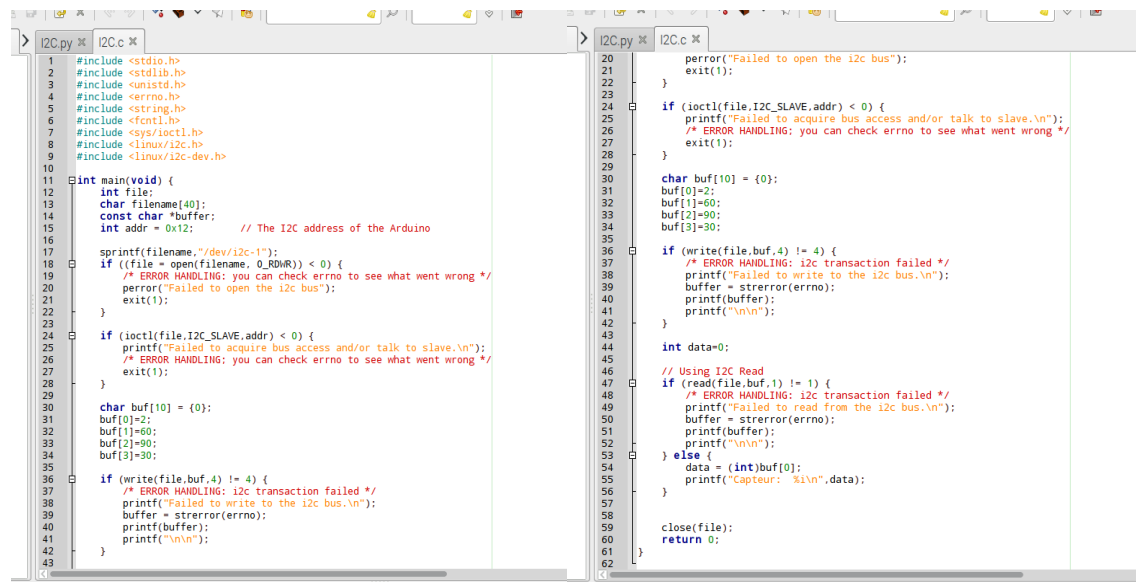
```

1 import smbus
2 import time
3
4 # Remplacer 0 par 1 si nouveau Raspberry
5 bus = smbus.SMBus(1)
6 address = 0x12
7
8 print "Envoi de la valeur 3"
9 bus.write_byte(address, 3)
10 # Pause de 1 seconde pour laisser le temps au traitement de se faire
11 time.sleep(1)
12 reponse = bus.read_byte(address)
13 print "La reponse de l'arduino : ", reponse
14

```

Pour notre projet nous avons choisi d'utiliser le langage C pour la programmation. Il a donc fallu traduire ce code Python en C. Nous utilisons alors la librairie i2c.h, elle permet d'utiliser le périphérique « /dev/i2c-1 » présent sur la RPi.

Pour initialiser le bus il faut définir l'accès à ce périphérique avec open() puis utiliser ioctl() pour donner l'adresse de l'esclave avec lequel la communication va être faite. Ensuite on utilise write() et read() pour lire et écrire des données.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <errno.h>
5 #include <string.h>
6 #include <fcntl.h>
7 #include <sys/ioctl.h>
8 #include <linux/i2c.h>
9 #include <linux/i2c-dev.h>
10
11 int main(void) {
12     int file;
13     char filename[40];
14     const char *buffer;
15     int addr = 0x12; // The I2C address of the Arduino
16
17     sprintf(filename, "/dev/i2c-1");
18     if ((file = open(filename, O_RDWR)) < 0) {
19         /* ERROR HANDLING: you can check errno to see what went wrong */
20         perror("Failed to open the i2c bus");
21         exit(1);
22     }
23
24     if (ioctl(file, I2C_SLAVE, addr) < 0) {
25         printf("Failed to acquire bus access and/or talk to slave.\n");
26         /* ERROR HANDLING: you can check errno to see what went wrong */
27         exit(1);
28     }
29
30     char buf[10] = {0};
31     buf[0]=2;
32     buf[1]=60;
33     buf[2]=90;
34     buf[3]=30;
35
36     if (write(file, buf, 4) != 4) {
37         /* ERROR HANDLING: i2c transaction failed */
38         printf("Failed to write to the i2c bus.\n");
39         buffer = strerror(errno);
40         printf(buffer);
41         printf("\n\n");
42     }
43
44     int data=0;
45
46     // Using I2C Read
47     if (read(file, buf, 1) != 1) {
48         /* ERROR HANDLING: i2c transaction failed */
49         printf("Failed to read from the i2c bus.\n");
50         buffer = strerror(errno);
51         printf(buffer);
52         printf("\n\n");
53     } else {
54         data = (int)buf[0];
55         printf("Capteur: %i\n", data);
56     }
57
58     close(file);
59     return 0;
60 }
61
62 }
```

i24:16: Volici Geany 1.24.1.

i24:16: Volici Geany 1.24.1

Les modifications réalisées au sein du code Arduino afin d'assurer la communication avec la Raspberry furent légères. Voici les lignes de code rajoutées :

Dans la partie **MOTEURS** :

```
int encodeurPin_G = 3;
int encodeurPin_D = 2;
```

Dans la fonction **setup()** :

```
pinMode(encodeurPin_G, INPUT_PULLUP);
pinMode(encodeurPin_D, INPUT_PULLUP);

////////////////////////////////// Raspberry ////////////////////////////////////
Wire.begin(SLAVE_ADDRESS); // adresse de l'esclave arduino dans le bus I2C
Wire.onReceive(receiveData); // appel de la fonction receiveData
//////////////////////////////////
```

Dans la partie **CONSTANTES** :

```
#include <Wire.h>
#define SLAVE_ADDRESS 0x12
#define SDA 20; // Connexion Raspberry 1.1
#define SCL 21; // Connexion Raspberry 1.2
```

Dans la partie **INITIALISATION DE LA MACHINE D ETAT** :

```
int etape = 0; // se mettra à un lorsque nous recevrons quelque chose
```

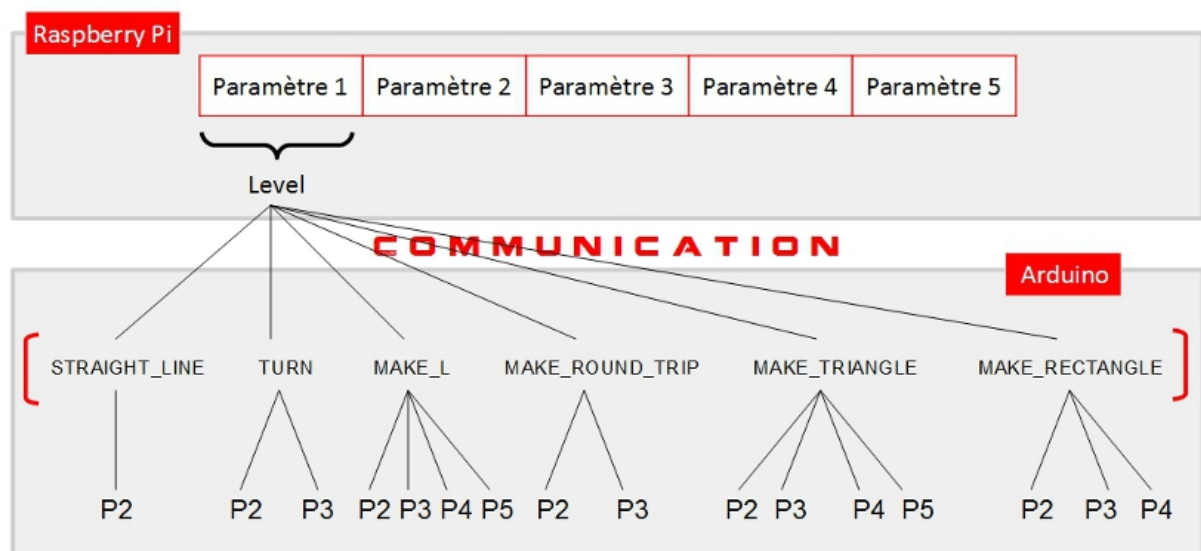
La fonction **receiveData(int byteCount)** :

```
void receiveData(int byteCount)
{
    while(Wire.available())
    {
        dataReceived = Wire.read();
        Serial.print("Donnee recue : ");
        Serial.println(dataReceived);
        tableau[tabIndice] = dataReceived;
        Serial.println(tabIndice);
        Serial.println(tableau[tabIndice]);
        tabIndice++;
    }
    etape = 1;
}
```

Après avoir réalisé ces différents tests, nous avons défini les fonctions permettant de réaliser les mouvements du robot. L'Arduino utilisant une machine d'état pour réaliser ces mouvements, nous avons établi une norme pour la communication entre l'Arduino et la RPi. Chaque cas de la machine d'état correspond à une fonction particulière ainsi la première donnée qui est envoyée sera le nombre correspondant au cas. Ensuite en fonction de celle-ci, l'Arduino traitera les données suivantes de façon à correspondre au cas associé.

Par exemple : l'envoi de « 1 , 50 » correspondra au cas StraightLine donc l'Arduino traitera 50 comme la distance à parcourir en cm. Alors que l'envoi de « 3, 20, 50, 0, 10 » correspondra au cas MakeL et les données seront traitées dans l'ordre comme étant : la première distance à parcourir (20cm), l'angle de rotation (50°), la direction du virage (0 pour gauche et 1 pour droite), et enfin la 2nd distance à parcourir (10cm). Voici un schéma qui met en avant notre protocole de communication :

Une fois qu'un cas est terminé l'Arduino envoie un signal à la RPi. Ceci permet à la RPi d'attendre ce signal pour passer à un autre cas, et ainsi d'éviter des conflits entre les différentes fonctions.



API :

Une fois que la communication fonctionnait et que les fonctions envoyées étaient bien réalisées, nous avons pensé à réaliser une API. Ceci permettrait de monter encore d'un niveau en programmation.

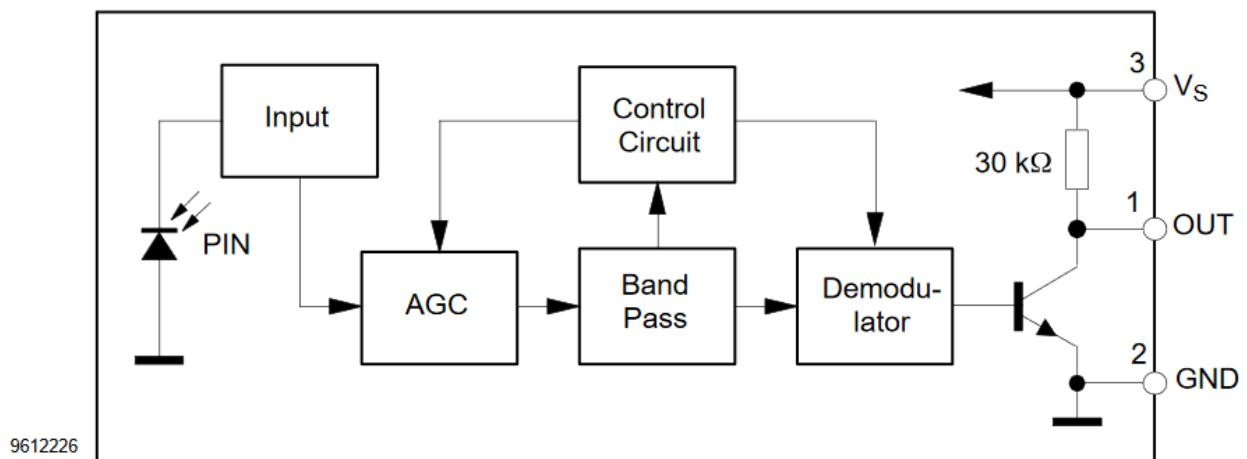
Ainsi un utilisateur lambda pourra prendre en main facilement le robot. Cependant la réalisation de la communication et la correction des différents problèmes liée à celle-ci a pris plus de temps que prévu et nous n'avons pas pu implémenter cette API.

Test du capteur infrarouge référencé : :

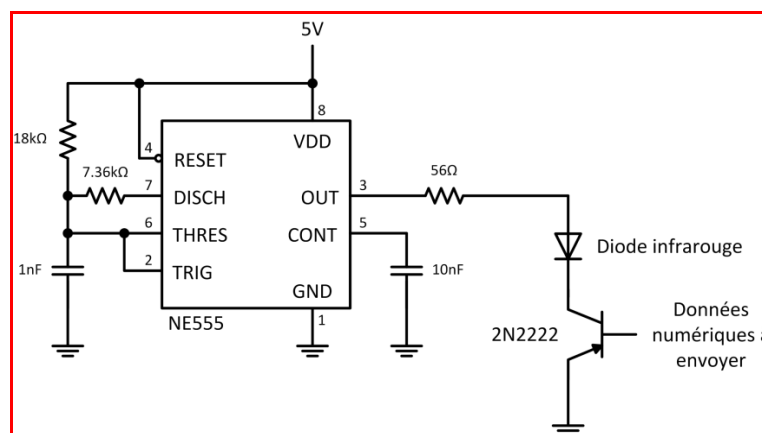
Nous avons dans un premier temps, tenté de réutiliser les capteurs du robot Peeke. En les dessoudant, il apparaît 5 pattes : 3 réservées au détecteur IR et 2 propres à la LED IR.

Pour réaliser des tests sur ces capteurs nous avons recherché de la documentation les concernant. Malheureusement ces capteurs étant anciens, aucune documentation n'a été trouvée à leur sujet. Cependant, de nouveaux modèles sont apparus. Nous avons alors supposé que la documentation de ces nouveaux capteurs était semblable à celle des anciens. Après lecture de celle-ci, nous nous sommes aperçus qu'il fallait générer un signal avec une certaine fréquence pour la LED pour que le récepteur puisse le détecter. En effet, le récepteur possède un circuit lui permettant d'accepter qu'une fréquence précise et de renvoyer un signal fonction de la distance de détection.

Block Diagram



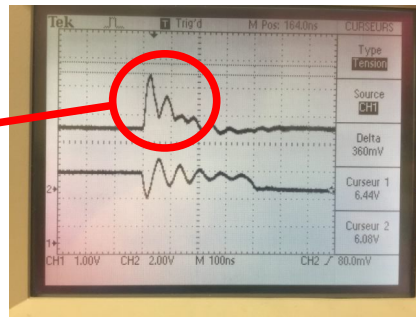
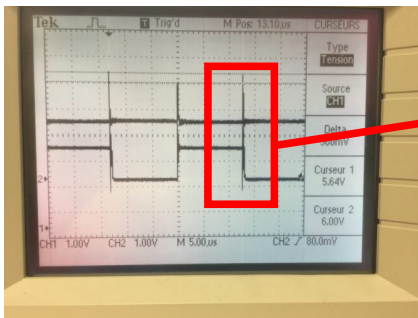
Nous avons alors réalisé le circuit ci-dessous pour générer un signal pour la LED correspondant à la fréquence du récepteur.



Malheureusement, les tests réalisés ne furent pas concluants. Les capteurs n'ont donc pas été utilisés.

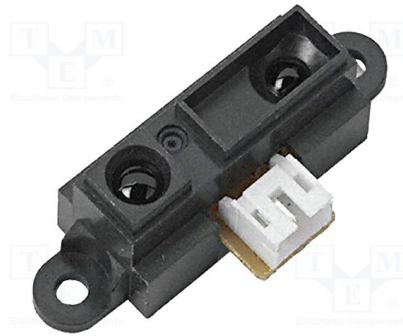


Les courbes ci-dessous montrent la différence de signal entre un obstacle présent ou non devant le capteur. Aucune information de distance entre le capteur et l'obstacle n'est exploitable et seul le passage d'un objet est détecté. Autrement dit, une fois l'obstacle devant le capteur, nous retrouvons le signal lorsqu'aucun objet ne l'obstruait.



Achat d'un capteur infrarouge :

Nous avons retenu le capteur suivant : SHARP GP2Y0ASK0F de par ses caractéristiques répondant à nos besoins. La portée du capteur ainsi que sa facilité d'implémentation sur le robot correspondent à nos critères (détection d'obstacles & positionnement sur coque).



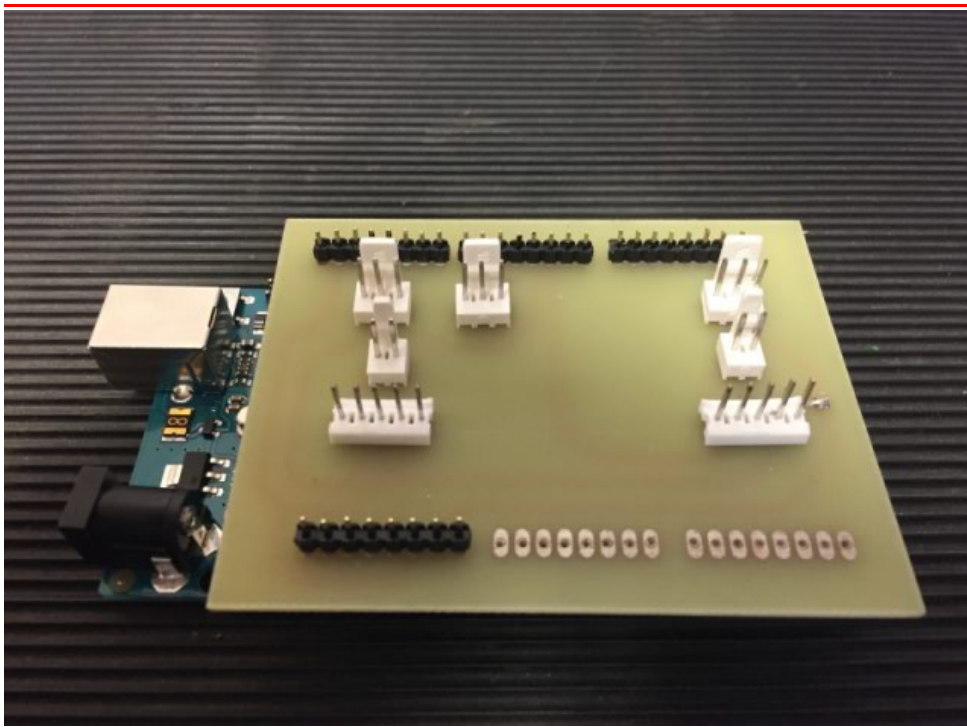
Voici ses caractéristiques :

- Type de capteur : **Photoélectrique**
- Portée : **40 – 300 mm**
- Configuration de sortie : **Analogues**
- Tension d'alimentation : **-0,3V – 7V (4,5 à 5V tension de travail)**
- Courant de service maximal : **22mA**
- Température de travail : **-10 – 60 °C**
- Poids : **6,23g**

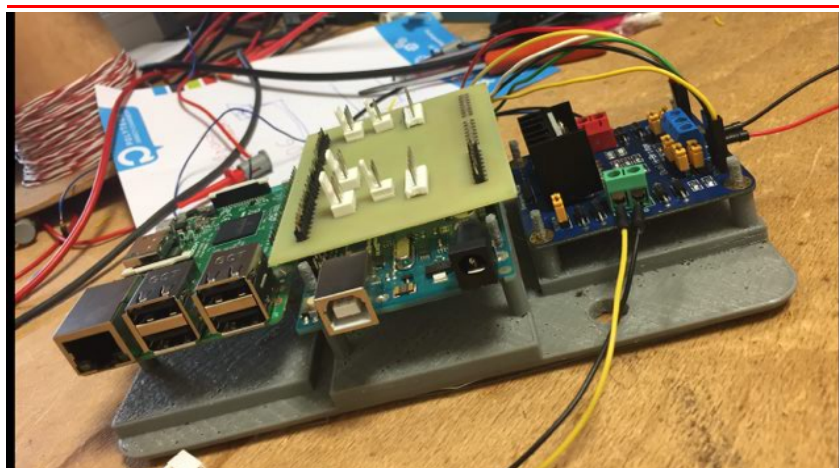
Pièces 3D et Shield Arduino

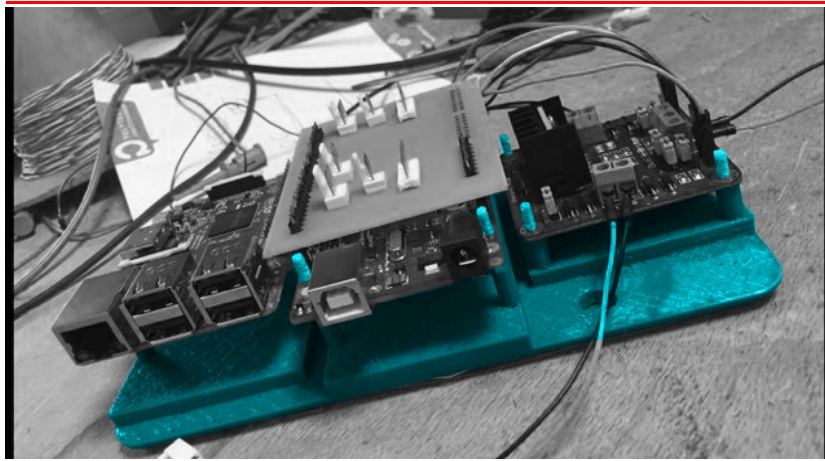
Afin d'avoir une meilleure structuration des cartes utilisées et une facilité de branchements entre ces différents éléments, nous avons oeuvré pour produire un shield ainsi qu'une base support des cartes réalisée sous logiciel de CAO puis matérialisée grâce à une imprimante 3D. Par ailleurs, une coque robot a été conçue en CAO par nos soins de façon à permettre, à l'avenir, une identité du projet et une facilité d'implémentation des éléments (cartes, capteurs IR, moteurs, évolution future).

Shield Arduino :

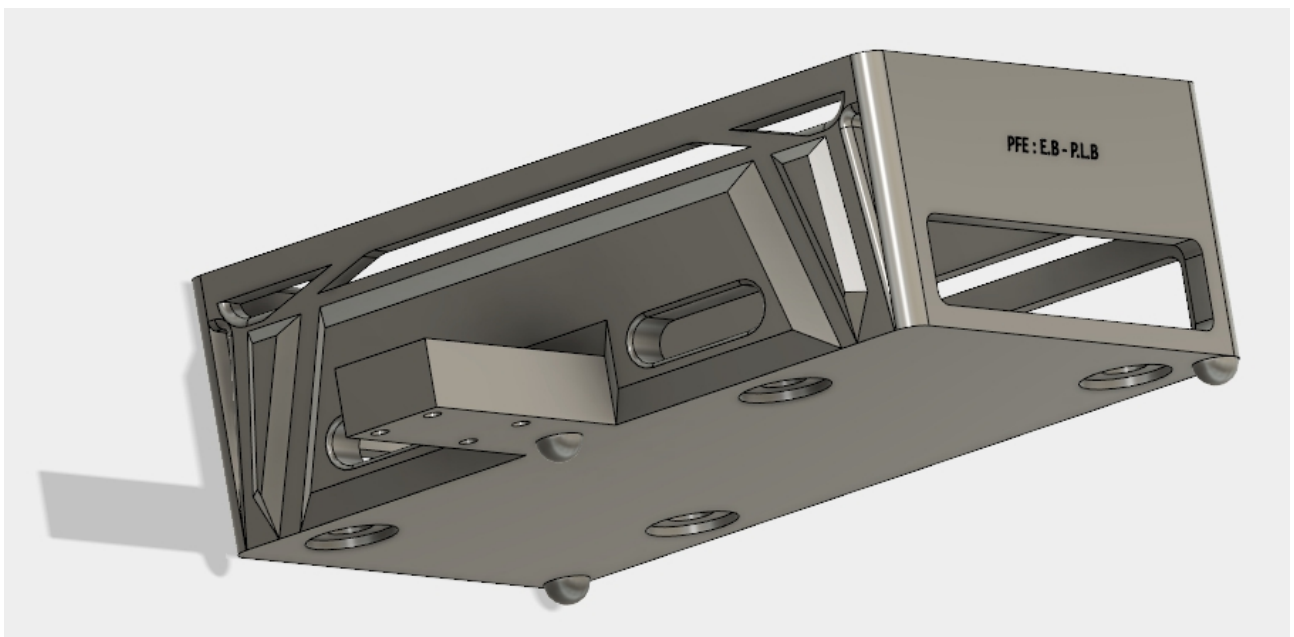


Support Cartes :





Coque robot :



Remarque :

- Les fichiers .stl du support des cartes et de la coque sont fournis avec le rapport.
- Le projet EAGLE du shield est aussi disponible.

Conclusion :

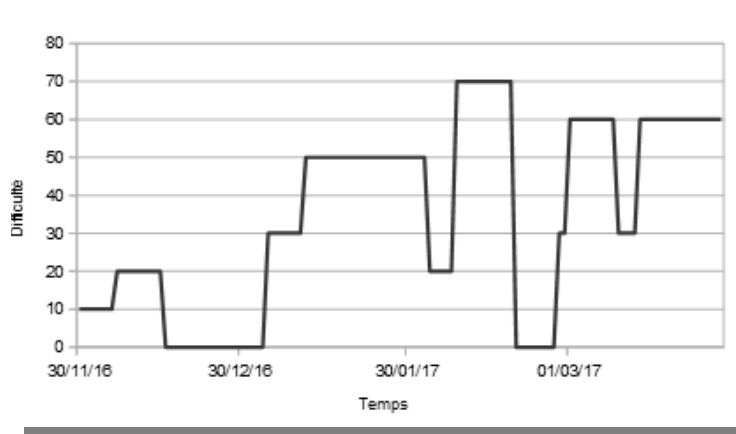
Ce projet fut destabilisant dans un premier temps dans la mesure où nous devions prendre en main d'anciens composants dont le fonctionnement pouvait être remis en question. Une partie de test fut donc nécessaire pour analyser quelles parties pourraient être réhabilitées ou non. Au cours de ce PFE, deux parties ont dû être distinguées, à savoir, la partie Arduino ainsi que la partie Raspberry. Nous avons séparé l'étude par couches hiérarchiques :

- mécanique
- mécatronique
- électro-informatique (bas niveau)
- programmation du robot
- API

Les compétences requises ont été pluridisciplinaires et nous ont permis de nous adapter. Certes, nous n'avons pas pu aboutir à une API livrable et implémenter d'autres capteurs. Cependant, notre investissement en totale autonomie au sein de ce projet ainsi que le travail rendu représentent, à nos yeux, une base d'évolution future solide.

Malgré une précision faible des encodeurs, une roue folle désavantageuse et des capteurs non réutilisables, nous avons apporté des solutions concrètes et fiables. Nous sommes capable de piloter les moteurs à l'aide de fonctions intuitives simples telles qu'un virage ou une ligne droite, mais aussi plus complexes comme un rectangle, un « L » ou encore un triangle.

L'approche temporelle au sein d'un projet reste le facteur perturbant dans le sens où il est difficile d'estimer le temps nécessaire à telle ou telle sous-partie. Voici un graphique mettant en avant le niveau de difficulté rencontré en fonction du temps. Chaque palier correspond à un numéro explicité ci-dessous :



- 1) Analyse, Cahier des charges, Prise en main du matériel : 30/11/16 – 6/12/16
- 2) Recherche de batteries & Consommation des éléments 7/12/16 – 15/12/16
- 3) Etude des moteurs, essais par alimentation + encodeurs + tests d'utilisabilité 4/01/17 – 10/01/17
- 4) Capteurs, essais des capteurs, test d'utilisabilité et mouvements 11/01/17 – 2/02/17
- 5) Courbes moteurs à vide, en charge avec module et arduino 3/02/17 – 7/02/17

- 6) Asservissement Vitesse / Position 8/02/17 – 18/02/17
- 7) Pente d'accélération 27/02/17 – 28/02/17
- 8) Mouvements & machine d'état 1/03/17 – 9/03/17
- 9) Connexion Raspberry 10/03/17 – 13/03/17
- 10) Pilotage des fonctions de mouvements par Raspberry 14/03/17

Au cours de ce projet, nous avons fait le choix de travailler ensemble sur chaque partie. Notre objectif fut de pouvoir enrichir, grâce à cette collaboration, nos connaissances tout en optimisant notre temps de travail. De cette manière, chacun apprenait sur chacune des phases de travail à savoir : électronique, mécanique et informatique.

Ces trois domaines qui, imbriqués les uns aux autres, forment l'anatomie de chaque robot.