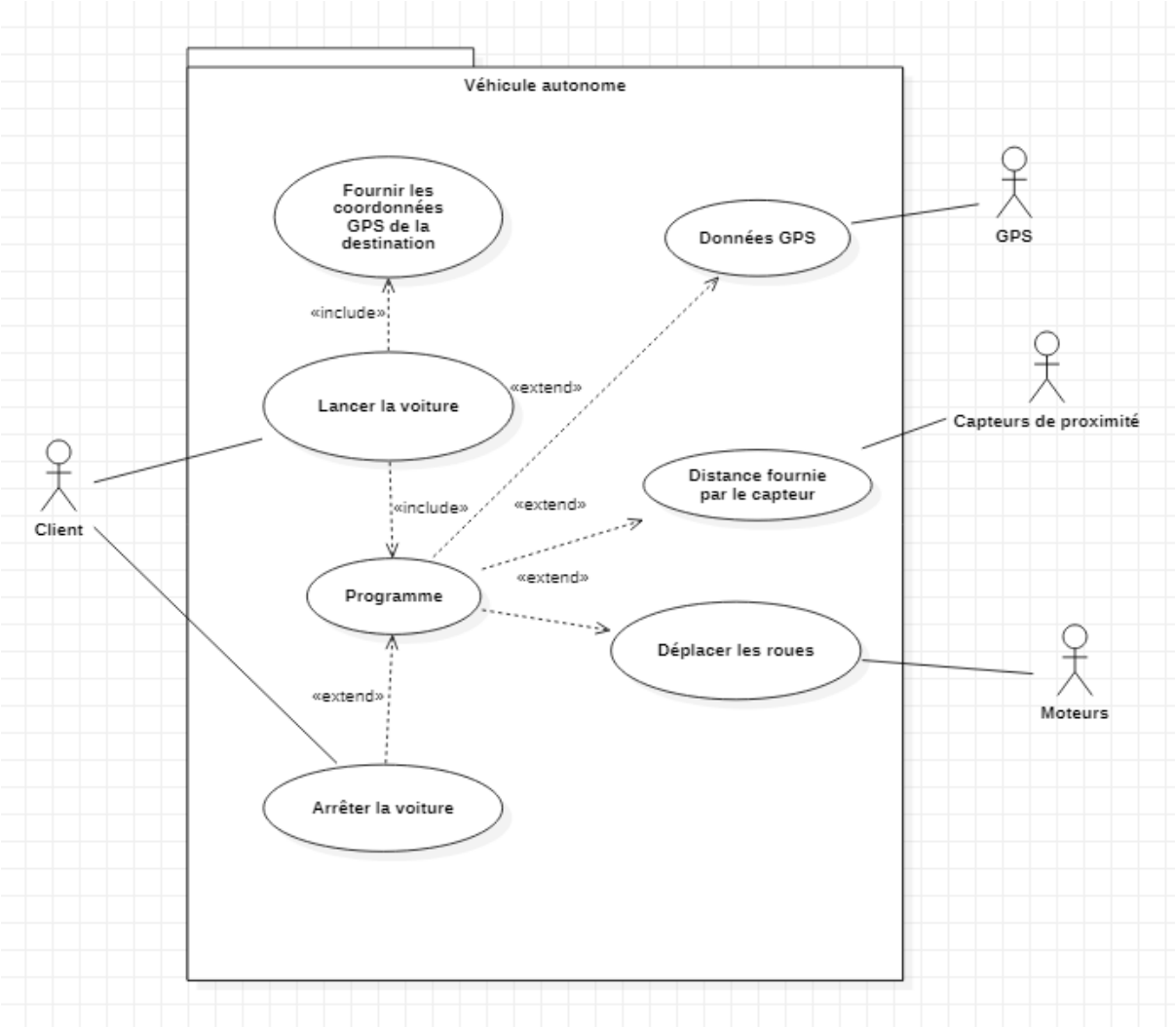
Compte rendu

1) Objectif

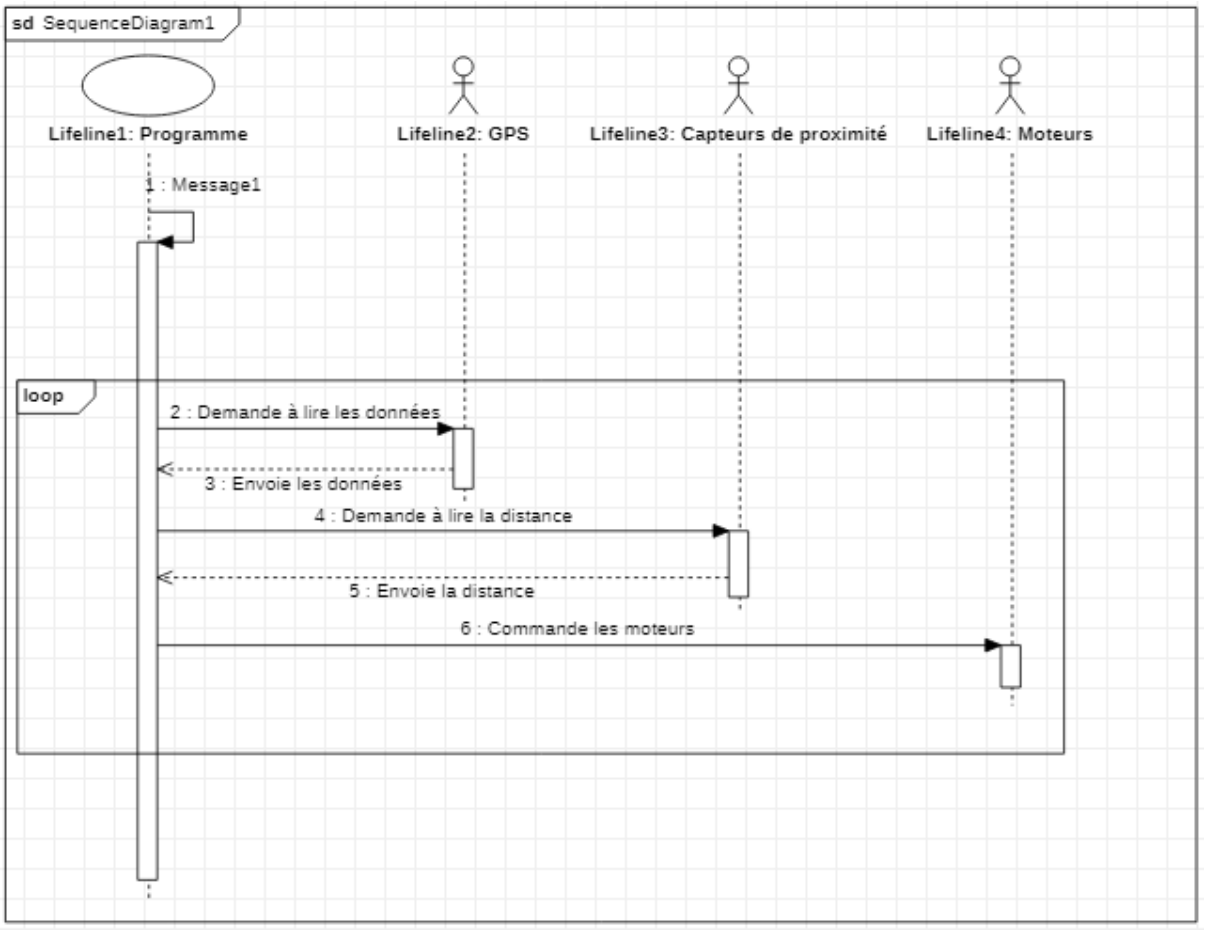
1.1) Description

Réaliser un véhicule autonome qui se déplace à des coordonnées GPS données par l'utilisateur. Nous avons également un capteur de proximité permettant d'éviter les obstacles.

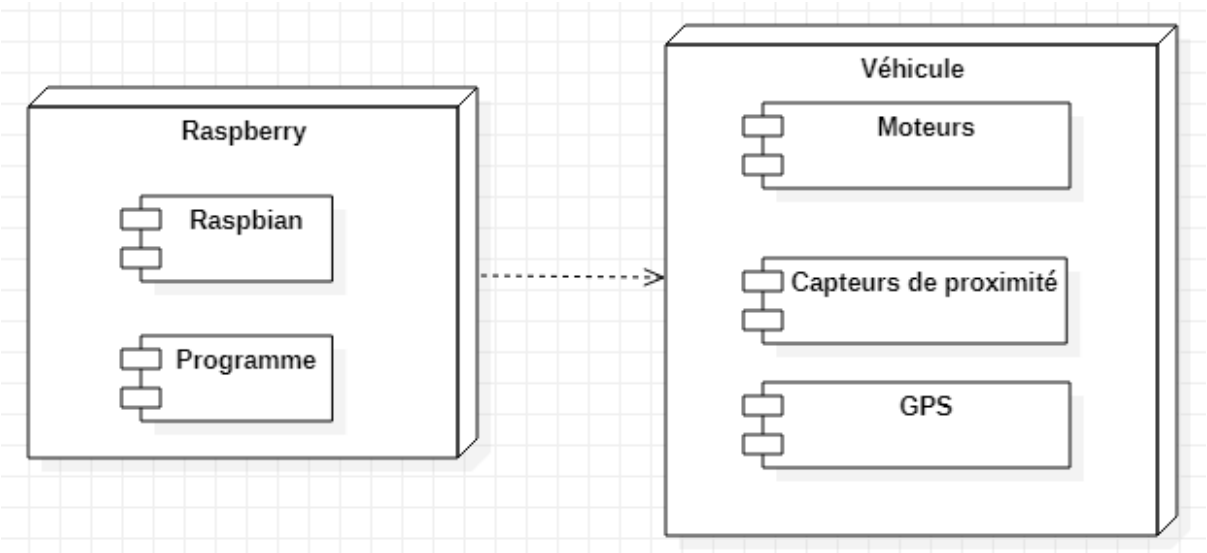
1.2) Diagramme de cas d'utilisation



1.3) Diagramme de séquence



1.4) Diagramme de déploiement



2) Spécification

2.1) Contraintes

2.1.1) Contraintes matérielles

- 2 Cartes de pilotage moteur Syren25
- Carte CAN I2C
- 4 Moteur 12V
- Capteur de proximité SHARP GP2Y0A02YK0F
- GPS
- Routeur wifi TL-WR841N
- Batterie externe

2.1.2) Contraintes logiciels

- Linux Raspbian Jessie 8
- C++

2.1.3) Contrainte réseau

- Wifi

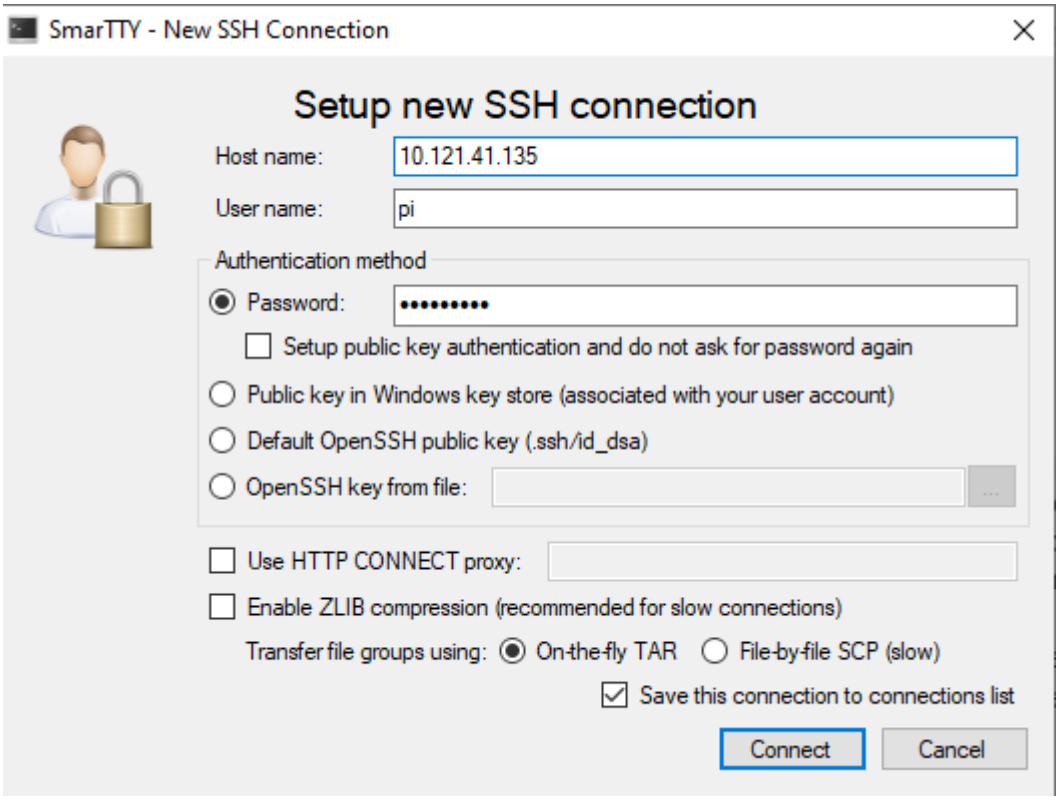
2.2) Performance

- Le GPS a une précision de 2.5M et un temps de rafraîchissement de 1 seconde et 26 seconde au premier démarrage à froid
- Le capteur de proximité mesure de 0 à 150 cm avec un temps de rafraîchissement de 38.3ms
- Les moteurs sont alimentés en 12V continue avec un RPM de 200

3) Activités de découverte

3.1) Installation du Raspberry Pi

Pour l'installation du raspberry nous avons gravé l'iso de l'os Jessie 8 pour raspberry sur une micro carte sd brancher avec un adaptateur micro sd -> usb sur un pc windows avec le logiciel Win32. Nous avons ensuite brancher cette micro sd avec l'os qui vient d'être gravé dans le raspberry.
Le raspberry est donc fonctionnel il nous reste plus qu'à connecter un câble ethernet et trouver l'ip donner au raspberry par le dhcp avec la commande: "ip a" . La dernière chose à faire est de se connecter au raspberry avec SmarTTY pour pouvoir utiliser la cross complications à distance.

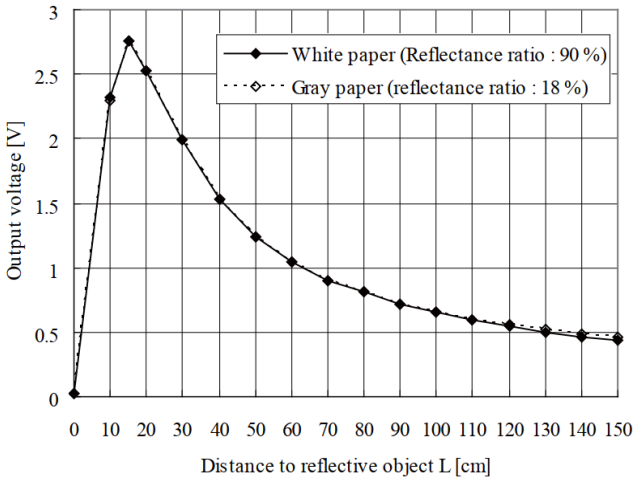


3.2) Capteur de proximité (Mathieu)

3.2.1) Analyse et mise en place

Le capteur de proximité nous permettra d'éviter les obstacles que pourrait rencontrer le véhicule en cours de route.

Nous connectons le capteur de proximité à un Convertisseur Analogique Numérique avec communication i2c.
Fonction de conversion de Volt à cm:



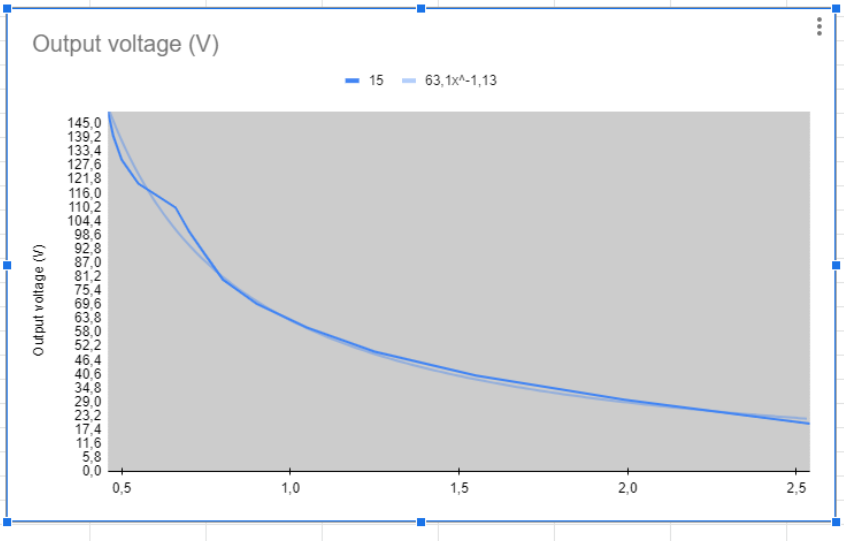
On remarque sur cette courbe, que pour un voltage donné, nous avons deux distances possibles:

- Une distance inférieur à 15 cm
- Une distance supérieur à 15 cm

On a pris la décision de ne pas prendre en compte la première possibilité (car le robot s'arrêtera automatiquement s'il détecte un obstacle à moins de ~20cm, donc "impossible" que la distance à un obstacle soit inférieur à cette valeur), afin de trouver une fonction de conversion volt à centimètre.

On a utilisé un tableur qui nous permet de retracer le graphique et d'en sortir une équation.

Distance (cm)	Output voltage (V)
15	2,73
20	2,54
30	1,99
40	1,55
50	1,25
60	1,05
70	0,9
80	0,8
90	0,75
100	0,7
110	0,66
120	0,55
130	0,5
140	0,475
150	0,46

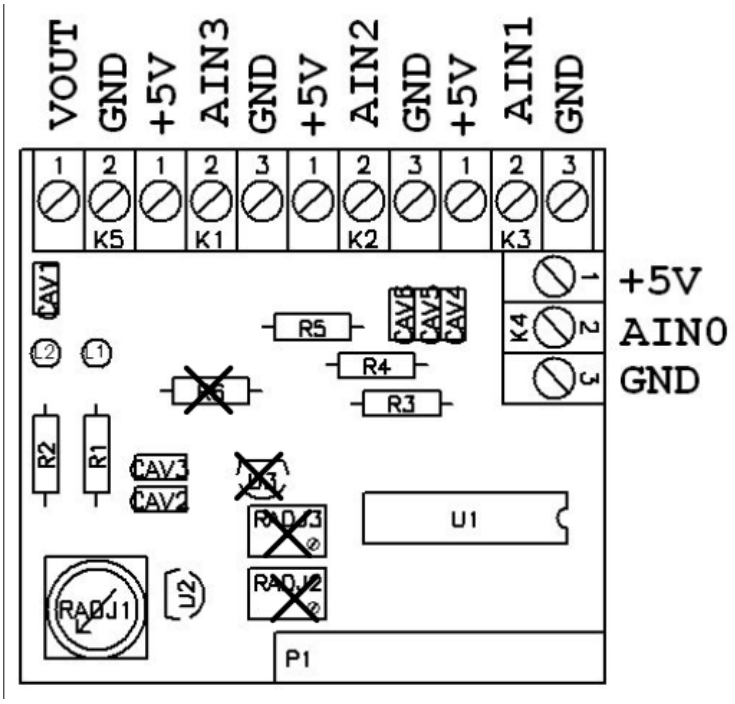


Le tableur nous propose l'équation d'une courbe de tendance de type "Série de puissance" :
Distance = 63,1 * volt^-1,13

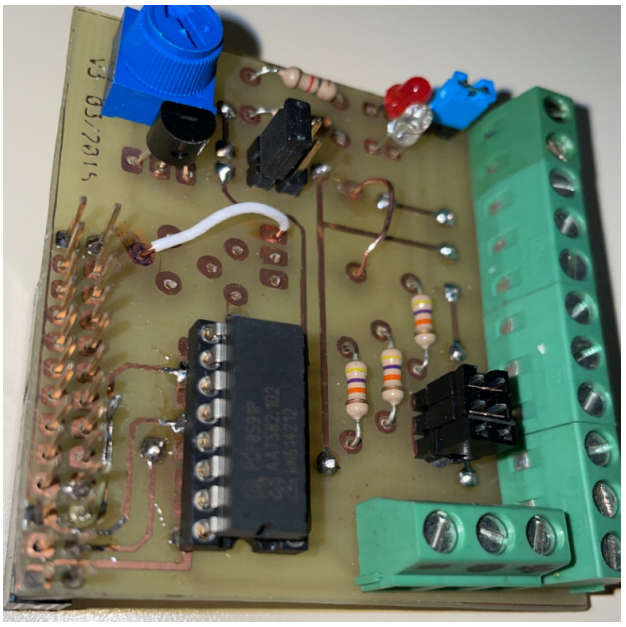
On remarque également qu'on a un voltage pouvant aller au dessus de 2,5V, hors notre carte CAN à pour Vcc 2,5V.

On a décidé d'enlever différents éléments de la carte, et d'y souder d'autres fils de cuivre, afin de passer sur du 5V.

Les éléments à enlever sont ceux barrés sur le schéma suivant:



Résultat:



La carte CAN i2c renvoie une valeur codée sur 8 bits, donc comprise entre 0 et 255 (2^8 -1), 0 correspondant à 0V, et 255 à 5V.

Donc pour obtenir la tension, on doit d'abord utiliser la fonction de conversion suivante:

Volts = (5 * valeur) / 255

On branche le capteur sur le port 1 de la carte comme ceci.

Une fois le raspberry pi branché, on cherche l'adresse de la carte i2c avec la commande suivante: i2cdetect -y 1

On obtient l'adresse de la carte i2c: 0x48.

Maintenant, on veut récupérer depuis une commande shell les valeurs renvoyées par le capteur de proximité. La commande utilisée est la suivante:
i2cget -y 1 [adresse de la carte] [numéro de port sur la carte]
Donc: i2cget -y 1 0x48 0x01

```
pi@raspberrypi:~ $ i2cget -y 1 0x48 0x01
0x4c
pi@raspberrypi:~ $ i2cget -y 1 0x48 0x01
0x62
pi@raspberrypi:~ $ i2cget -y 1 0x48 0x01
0x6c
pi@raspberrypi:~ $ i2cget -y 1 0x48 0x01
0x83
pi@raspberrypi:~ $ i2cget -y 1 0x48 0x01
0x8c
```

On va essayer de retrouver les valeurs en cm:

On passe de l'hexadécimal à décimale: 0x4c = $(76)_{10}$

On convertit 76 en volt: tension = (5 * 76)/255 = 1,49V

On convertit la tension en distance: distance = 63,1 * 1,49^-1,13 = 40,2 cm

Les mesures sont bonnes.

3.2.2) Le programme

Avec toutes nos informations, on créer un programme:

a) La conversion du nombre codé sur 8 bits à volt:

```
7   #define A_CONVERSION 38.2
8   #define PUISSANCE_CONVERSION -0.879
9   #define LIMITVOLT 0.4
10  #define BITS 8
11  #define VOLT_MAX 5
12  #define ADRESS 0x48
13
14
15  double conversionBitToVolts(double voltsBit)
16  {
17      double volts = (VOLT_MAX * voltsBit) / pow(2,BITS);
18      return(volts);
19  }
```

b) La conversion de Volt à cm:

```
21  double conversionVoltToDistance(double volts)
22  {
23      //distance = 63,1 * volts^-1,13
24      if(volts < LIMITVOLT){
25          return 0;
26      }
27      double distance = A_CONVERSION * pow(volts, PUISSANCE_CONVERSION);
28      return(distance);
29  }
```

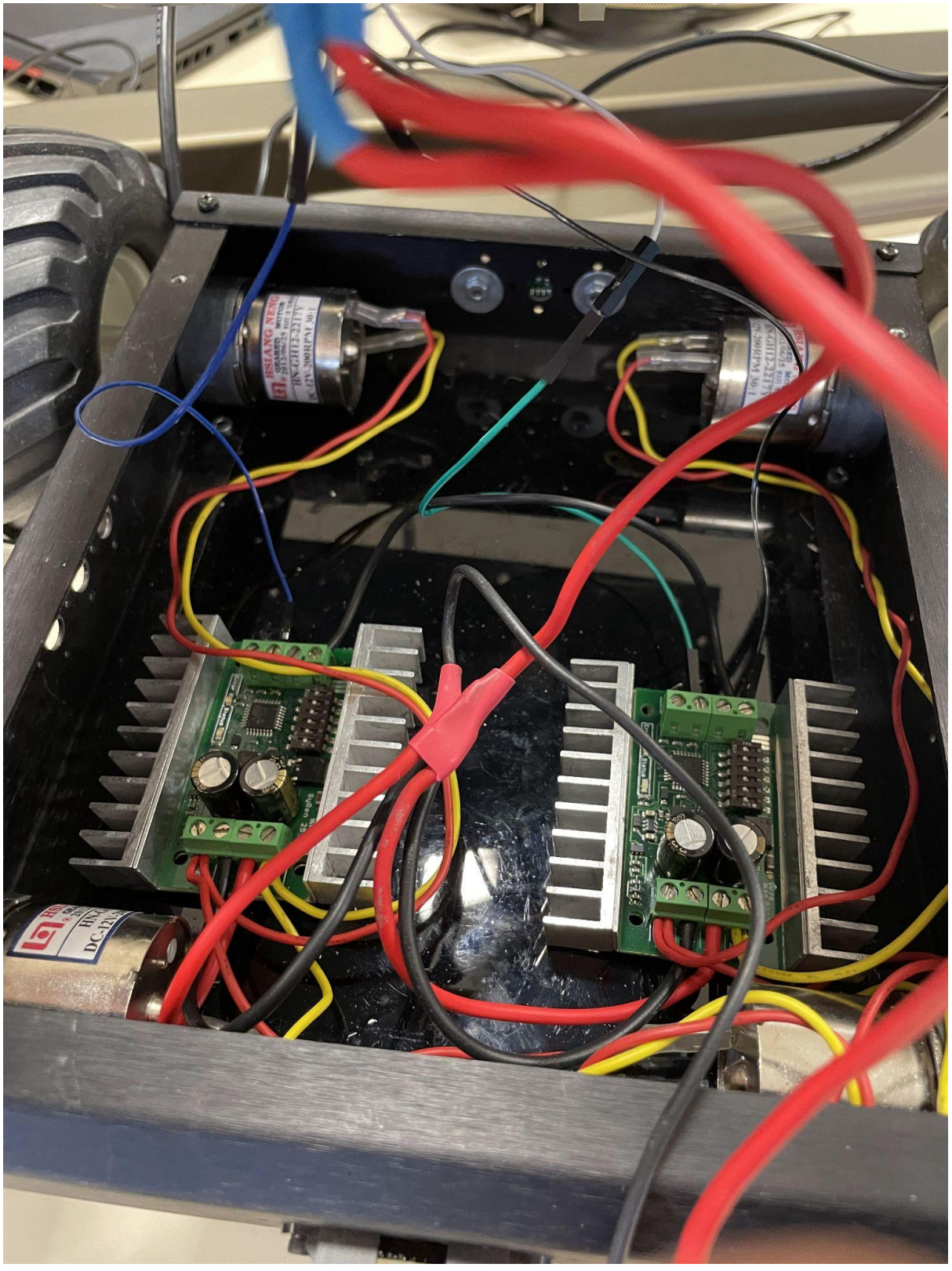
c) Programme principal:

```
32  int main()
33  {
34      int fd;
35      int voltsBit = 0;
36      double voltage = 0;
37      double distance = 0;
38      wiringPiSetup();
39      fd = wiringPiI2CSetup(ADRESS);
40
41      while (1)
42      {
43          //Lecture en volt
44          voltsBit = wiringPiI2CReadReg8(fd,1);
45          //Conversion nombre sur 8 bits à volts
46          voltage = conversionBitToVolts(voltsBit);
47          //Conversion de volt à cm
48          distance = conversionVoltToDistance(voltage);
49
50          printf("distance=%.2lf cm\n",voltsBit,voltage,distance);
51          delay(500);
52      }
53      return 0;
54  }
```

3.3) Les moteurs

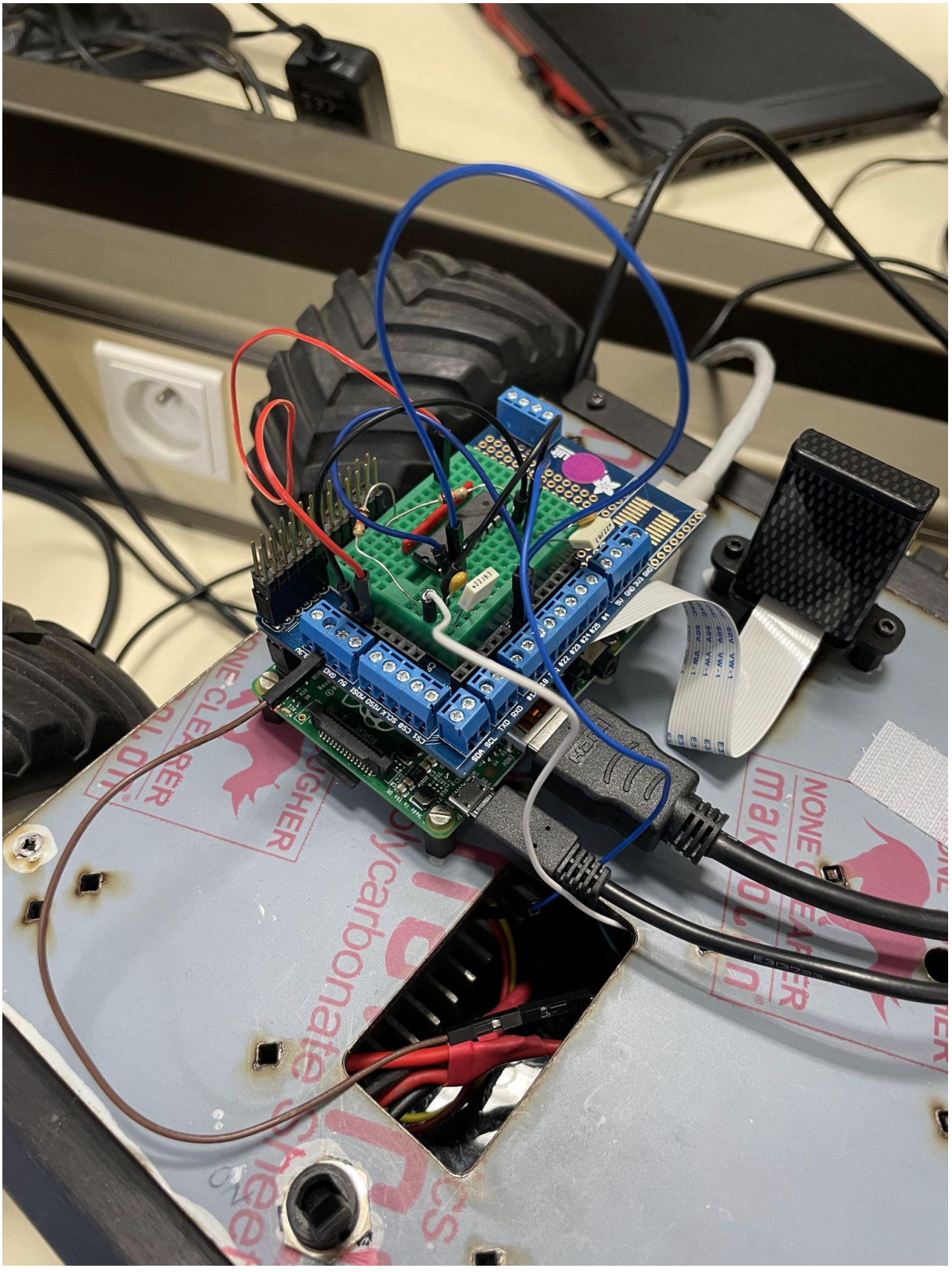
3.3.1) Analyse et mise en place

Afin de faire fonctionner les 4 moteurs, on les a reliés à 2 cartes de pilotage moteur Syren25 et on décide de les contrôler en PWM .



Pour le branchement entre les cartes de pilotage moteur et le raspberry on à ajouté une carte prototype avec un uln2803 .

Le câblage entre les cartes de pilotage moteurs et l'uln2803 : on doit partir du S1 de chaque carte de pilotage moteur et venir se brancher sur un GPIO du prototype pour ensuite passé du GPIO au port "input" de l'uln2803 qui a donc inversé les ports GPIO mais surtout passer la tensions de 3.3v à 5v et pour finir on ajoute un filtre passe bas qui sert a avoir la moyenne



3.3.2) Le programme

Pour contrôler les moteurs il faut envoyer un signal en PWM de 0 à 200 sur le port GPIO respectif de chaque carte de pilotage moteur.

```
#include <wiringPi.h>
#include <stdio.h>
#include <softPwm.h>

int main(){
    //configurations des moteurs en PWM
    int PWM_pinD = 1;//port gpio 0 inverser à cause du filtre passe bas
    int PWM_pinG = 0;//port gpio 1 inverser à cause du filtre passe bas
    wiringPiSetup();
    pinMode(PWM_pinD,OUTPUT);
    pinMode(PWM_pinG,OUTPUT);
    softPwmCreate(PWM_pinD,1,200);//contrôle roues droite
    softPwmCreate(PWM_pinG,0,200);//contrôle roues gauche

    //La voitrure avance pendant 5 secondes
    softPwmWrite (PWM_pinD, 60);
    softPwmWrite (PWM_pinG, 60);
    delay (5000) ;
    //La voitrure recule pendant 5 secondes
    softPwmWrite (PWM_pinD, 15);//contrôle roues droite
    softPwmWrite (PWM_pinG, 15);//contrôle roues gauche
    delay (5000);
}
```

3.4) GPS

3.4.1) Analyse et mise en place

On dispose d'une carte RPI GPS Add-on ainsi que d'un GPS ANTENNA GPS 900-1.
Le GPS a une précision de 2,5 mètres en extérieur et récupère des données toutes les 1 secondes.

3.4.2) Le programme

Nous avons déjà des fichiers sources fournies avec le GPS, permettant de récupérer les données directement.
Nous avons décidé de recréer une nouvelle classe utilisant les structures et les fonctions des fichiers sources fournies.

a) Méthode getData()

La méthode getData() renvoie un tableau 2D contenant les données GPS récupérées pendant son exécution.
Nous entrons en paramètre le tableau dans lequel seront stockées les valeurs, et nData qui correspond au nombre de données que nous voulons récupérer.

Sachant que le GPS récupère une donnée par seconde, si nData = 10, alors la méthode prendra 10 secondes à être exécutée.

```
20  double* CGPS::getData(double DataCP[][5], int nData)
21  {
22
23      for(int i=0; i<nData; i++)
24      {
25          // Methode qui met dans data (structure loc_t) differentes données
26          gps_location(fd,&data);
27          DataCP[i][0] = data.latitude;
28          DataCP[i][1] = data.longitude;
29          DataCP[i][2] = data.altitude;
30      }
31
32      //Renvoie le tableau 2D
33      return reinterpret_cast<double *>(DataCP);
34  }
```

b) Méthode getLocData()

Le GPS peut également nous fournir la position est-ouest et l'hémisphère.

Nous avons créé une méthode nous permettant de la récupérer indépendamment de la méthode getData() afin de renvoyer des caractères.

```
74  char* CGPS::getLocData()
75  {
76      char tabLocData[2] = {data.lat, data.lon};
77
78      return tabLocData;
79  }
```

4) Conception générale

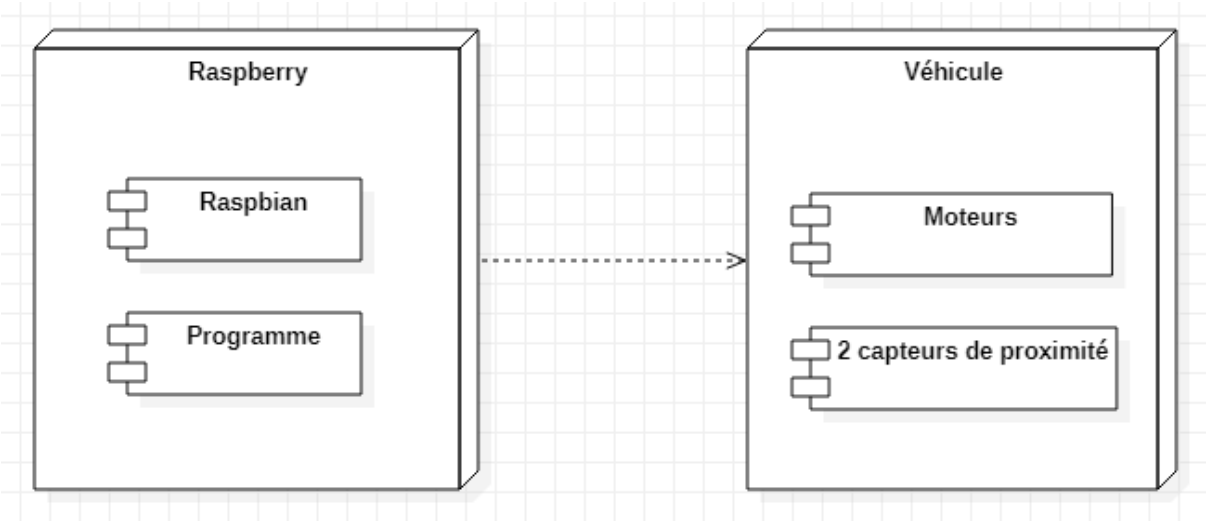
4.1) Solutions retenues

Dus à un manque de temps, et au manque de précision du capteur, nous avons décidé de l'écarter du projet.
Les données que le GPS nous fournit sont très instables et peu précises, de plus, le temps qu'il met à récupérer les données pose un autre problème de timing dans le programme (attente de 1 seconde sans que rien ne se passe).

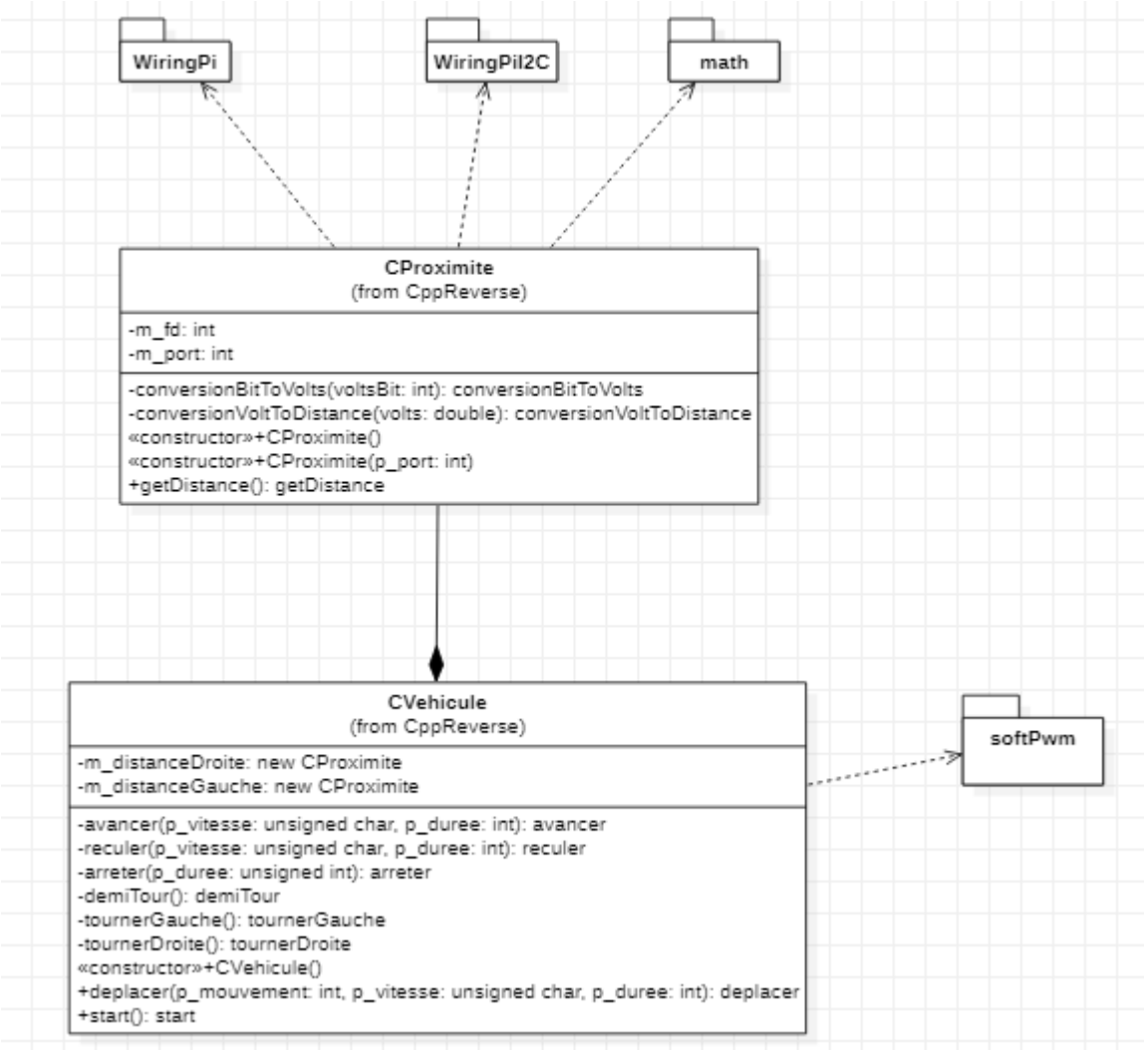
Nous allons alors simplement faire une voiture autonome, qui évite les obstacles à l'aide de ses capteurs de proximité.

Un des problèmes rencontrés est que le capteur de proximité détecte uniquement les obstacles exactement devant lui. Nous avons décidé de rajouter un second capteur afin d'avoir un plus grand champ de vision.

4.2) Diagramme de déploiement final



4.3) Diagramme de classe



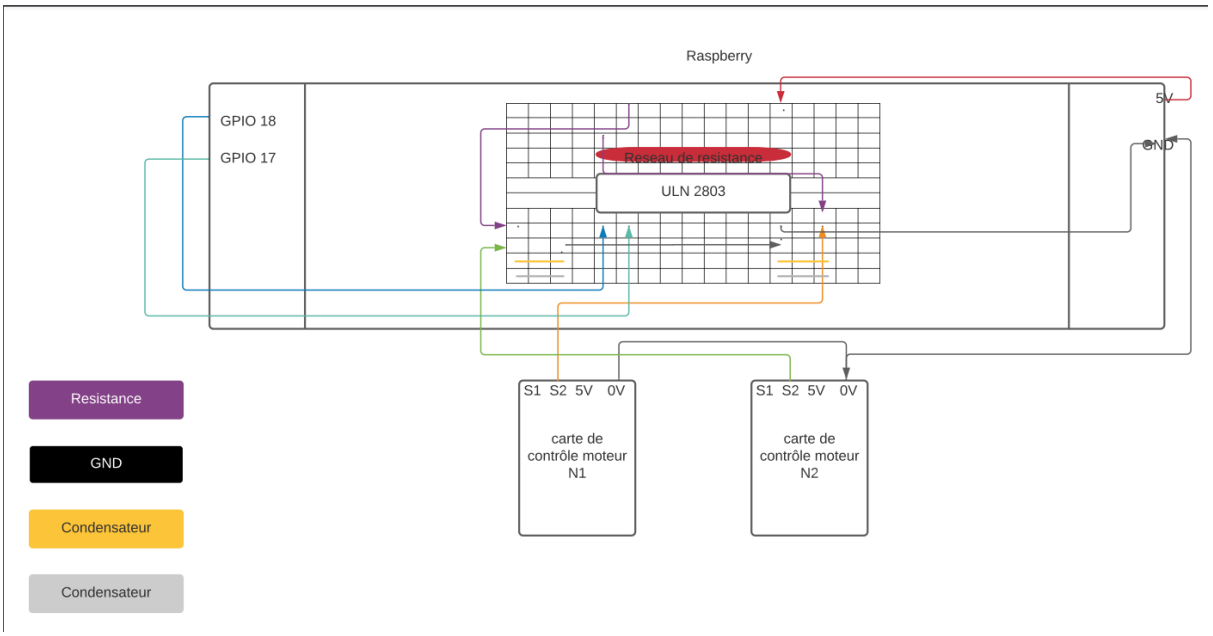
5) Conception détaillée et réalisation

5.1) Explication détaillée

Notre véhicule dispose de deux capteurs de proximité mis à l'avant afin de détecter les obstacles potentiels. Si un obstacle est détecté, alors la voiture s'arrête et change de direction pour continuer son chemin.
Nous allons donc créer deux classes:

- CProximite qui permettra de gérer un capteur de proximité
- CVehicule qui permettra de contrôler la voiture, avec la gestion d'obstacle et les déplacement

5.2) Schéma électrique



5.3) Réalisation des classes

5.3.1) Classe CProximite

Analyse de la classe:
On a besoin des attributs suivants:

- m_fd (int) qui correspond à l'ID de l'appareil
- m_port (int) qui correspond au numéro de port utilisé sur la carte i2c.

Et des méthodes suivantes:

- conversionBitToVolts(voltsBits: int) qui convertit la valeur du capteur en volt
- conversionVoltToDistance(volts: double) qui convertit les volt en cm
- getDistance() qui renvoie la distance
- CProximite(m_port: int) qui est le constructeur avec en paramètre le numéro de port sur la carte i2c du capteur correspondant

Code source:

CProximite.h

```
#define ADRESS 0x48
#define A_CONVERSION 38.2
#define PUISSANCE_CONVERSION -0.879
#define LIMITVOLT 0.4
#define BITS 8
#define VOLT_MAX 5

class CProximite
{
    private:
        int m_fd;
        //Numero du port sur la carte I2C
        int m_port;

        // Methode pour convertir le nombre sur 8 bits a volt
        double conversionBitToVolts(int voltsBit);
        //conversion des volts en distance
        double conversionVoltToDistance(double volts);

    public:
        CProximite();
        //Constructeur avec parametre
        CProximite(int p_port);
        //Destructeur
        ~CProximite();

        //Methode qui renvoie la distance
        double getDistance();
};
```

Constructeur:

```
CProximite::CProximite(int p_port)
{
    m_port = p_port;
    wiringPiSetup();
    m_fd = wiringPiI2CSetup(ADRESS);
}
```

Méthode conversionBitToVolts:

```
// Conversion nombre sur 8 bits a volts
double CProximite::conversionBitToVolts(int voltsBit)
{
    double volts = (VOLT_MAX * voltsBit) / pow(2,BITS);
    return(volts);
}
```

Méthode conversionVoltToDistance

```
28  //conversion des volts en distance
29  double CProximite::conversionVoltToDistance(double volts)
30  {
31      //Calcul de la distance: 63,1 * volts^-1,13
32      if(volts < LIMITVOLT){
33          return 0;
34      }
35      double distance = A_CONVERSION * pow(volts, PUISSANCE_CONVERSION);
36      return(distance);
37  }
```

Méthode getDistance

```
39  //Renvoie la distance detectée par le capteur
40  double CProximite::getDistance()
41  {
42      //Lecture en volt
43      int voltsBit = wiringPiI2CReadReg8(m_fd,m_port);
44      //Conversion nombre sur 8 bits a volts
45      double voltage = conversionBitToVolts(voltsBit);
46      //Conversion de volt a cm
47      double distance = conversionVoltToDistance(voltage);
48      return distance;
49  }
```

5.3.2) Classe CVehicule

Analyse de la classe:
On a besoin des attributs suivants:

- m_distanceDroite (CProximite) qui est l'objet correspondant au capteur sur la droite de la voiture
- m_distanceGauche (CProximite) qui est l'objet correspondant au capteur sur la gauche de la voiture

Et des méthodes suivantes:

- avancer(p_vitesse, p_duree) fait avancer la voiture selon les paramètres
- reculer(p_vitesse, p_duree) fait reculer la voiture selon les paramètres
- arreter(p_duree) arrête le véhicule pendant la durée indiquée
- demiTour() fait faire un demi tour au véhicule
- tournerGauche() tourne à gauche de 90°
- tournerDroite() tourne à droite de 90°
- deplacer(p_mouvement, p_vitesse, p_duree) méthode publique pour deplacer le véhicule selon les paramètres
- start() lance le véhicule et son fonctionnement autonome

Code source:

CVehicule.h

```
 9    #define PIN_DROITE 1
10    #define PIN_GAUCHE 0
11
12    #define AVANT 0
13    #define ARRIERE 1
14    #define ARRET 2
15    #define DEMI_TOUR 3
16    #define DROITE 4
17    #define GAUCHE 5
18
19    class CVehicule
20    {
21    private:
22        // Fait avancer le vehivule en avant selon une vitesse et une durée
23        bool avancer(unsigned char p_vitesse, int p_duree);
24        // Fait reculer le vehicule en arrière selon une vitesse et une durée
25        bool reculer(unsigned char p_vitesse, int p_duree);
26        // Arrête le vehicule selon une durée
27        bool arreter(unsigned int p_duree);
28        // Le vehicule fait un demi-tour
29        bool demiTour();
30        // Fait tourner le vehicule à Gauche
31        bool tournerGauche();
32        // Fait tourner le vehicule à Droite
33        bool tournerDroite();
34
35        CProximite *distanceGauche;
36
37        CProximite *distanceDroite;
38
39    public:
40        //Constructeur
41        CVehicule();
42        //Destructeur
43        ~CVehicule();
44        // Fait deplacer le vehicule selon le mouvement la vitesse et la durée
45        bool deplacer(int p_mouvement, unsigned char p_vitesse, int p_duree);
46        // Lancement du vehicule
47        void start();
48    };
```

Méthode deplacer

```
81  // Methode qui permet de deplacer le vehicule
82  bool CVehicule::deplacer(int p_mouvement, unsigned char p_vitesse, int p_duree)
83  {
84      switch(p_mouvement) {
85          case AVANT:
86              avancer(p_vitesse, p_duree);
87              break;
88          case ARRIERE:
89              reculer(p_vitesse, p_duree);
90              break;
91          case ARRET:
92              arreter(p_duree);
93              break;
94          case DEMI_TOUR:
95              demiTour();
96              break;
97          case DROITE:
98              tournerDroite();
99              break;
100         case GAUCHE:
101             tournerGauche();
102             break;
103         default:
104             return 0;
105     }
106     return 1;
107 }
```

Méthode avancer

```
109 // Fait avancer le vehivule en avant selon une vitesse et une duree
110 bool CVehicule::avancer(unsigned char vitesse, int p_duree)
111 {
112     if(vitesse>60 || vitesse == 0){
113         return 0;
114     }
115     softPwmWrite (PIN_GAUCHE, vitesse+20);
116     softPwmWrite (PIN_DROITE, vitesse+20);
117     delay(p_duree);
118
119     return 1;
120 }
```

Les méthodes renvoient 0 en cas d'erreur de fonctionnement (mauvais paramètre(s) par exemple), ou 1 si tout s'est passé comme prévu.

5.4) Réalisation de programme d'exemple

5.4.1) Fait le tour du "pâté de maison"

Ce programme fait simplement déplacer le véhicule autour d'un "pâté de maison".

```
#include "CVehicule.h"
#include "CProximite.h"


#define AVANT 0
#define ARRIERE 1
#define ARRET 2
#define DEMI_TOUR 3
#define DROITE 4
#define GAUCHE 5

int main()
{

    // creation objet CVehicule en dynamique
    CVehicule *vehicule1=new CVehicule();

   // Lance le vehicule
    vehicule1->deplacer(ARRET,1,5000);
    vehicule1->deplacer(AVANT,50,3600);
    vehicule1->deplacer(GAUCHE,1,1);
    vehicule1->deplacer(AVANT,50,6200);
    vehicule1->deplacer(GAUCHE,1,1);
    vehicule1->deplacer(AVANT,50,4600);
    vehicule1->deplacer(GAUCHE,1,1);
    vehicule1->deplacer(AVANT,50,6200);
    vehicule1->deplacer(GAUCHE,1,1);
    vehicule1->deplacer(AVANT,50,1600);

    return (0);
}
```

5.4.2) Gestion d'obstacle

L'objectif de ce programme est que la voiture avance jusqu'à détecter un obstacle.
Lorsque le capteur nous renvoie la valeur 0, il n'y a en réalité aucun obstacle en vue, c'est pour cela que nous nous arrêtons à une distance comprise entre 1 et 21 cm.

```
#include "CProximite.h"
#include "Cvehicule.h"
int main()
{

      CProximite *distance1=new CProximite();
      Cvehicule *vehicule1=new Cvehicule();

      while(1)
      {
        double distance = distance1->getDistance();
        //printf("distance=%lf\n",distance);
        printf("distance=%lf cm.\t", distance);
        if(distance < 21 && distance > 1)
        {
            printf("Entrer IF\n\n");
            delay(50);
            printf("\nDistance apres le demitour= %lf cm.\n",distance1->getDistance());
            vehicule1->arreter(2000);
        }
        else
        {
            printf("ELSE %lf\n",distance);
            vehicule1->avancer(30,50);
        }
        delay(50);
        printf("fin des conditions\n");
      }

    return (0);
}
```

*Dans ce programme, les méthodes arreter() et avancer() sont publiques.

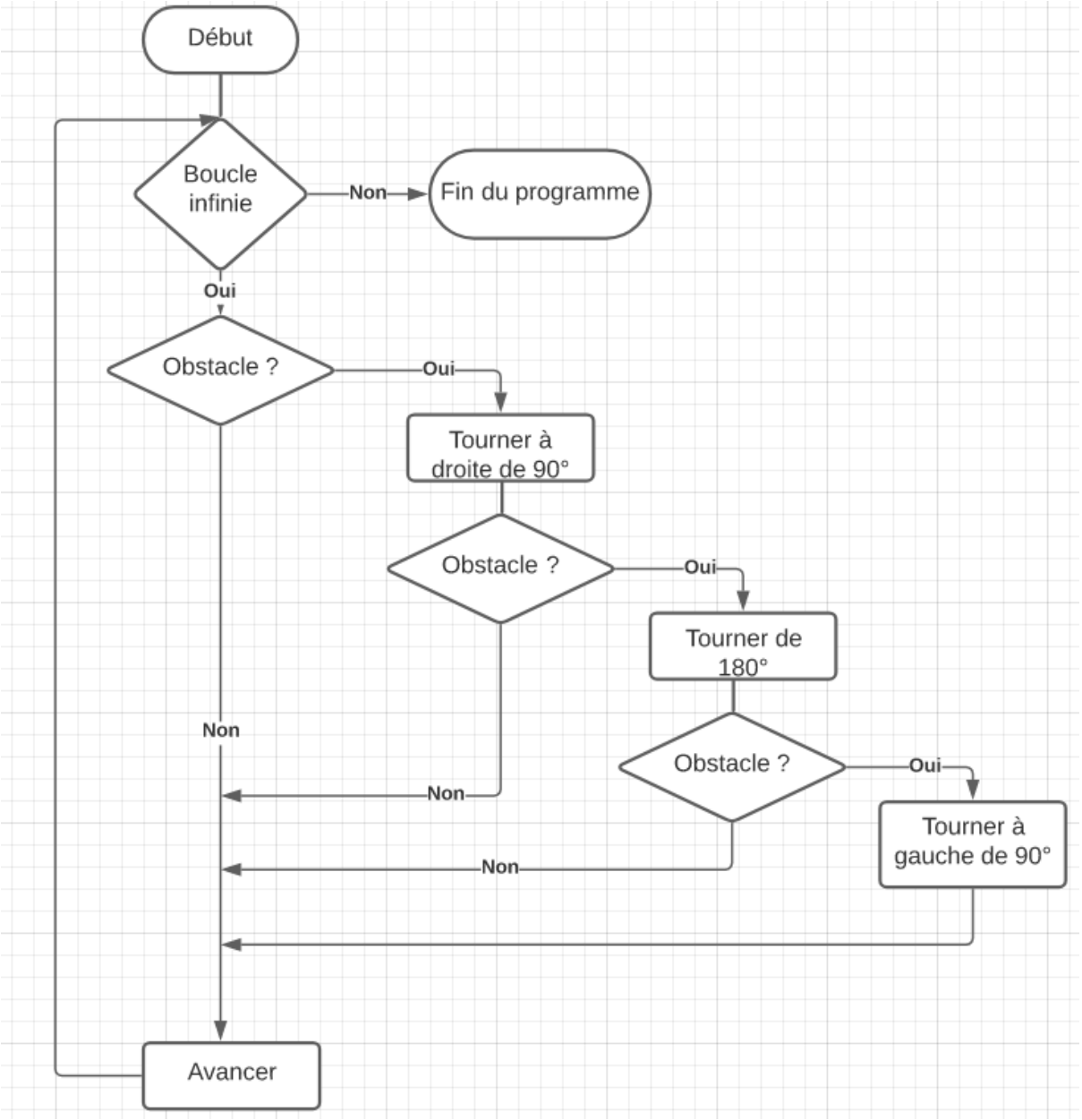
6) Intégrations des modules

Pour l'intégration finale, nous avons créé le système de gestion d'obstacle, l'objectif est que le véhicule ne soit jamais bloqué peu importe la situation.
Le système est simple, on avance sauf:

- Si il y a un obstacle, on tourne à droite
- Si on fait encore face à un obstacle on fait demi-tour
- Si nous avons encore un obstacle, on tourne à gauche

Avec ce système, la voiture essaiera toutes les directions (devant, droite puis gauche) avant de retourner sur ses pas.

Algorigramme du système:



7) Problèmes rencontrés et les solutions

Nous avons rencontré plusieurs problèmes pendant la conception du projet, les principaux concernent le gps qui ne fonctionnait pas en intérieur, qui ralentissait le programme à cause du temps de réponse pour récupérer les données qui varient (de 1 seconde en temps normal, à plusieurs dizaines de secondes de temps en temps). Ces différents problèmes nous ont poussé à prendre la décision d'écarter le GPS de notre projet, et de se concentrer à créer une voiture autonome avec ses capteurs de proximité.

Nous avons eu plusieurs fois le même problème avec la carte i2c, qui avait des ponts mal soudés, ce qui rendaient les données illisibles.

Un autre problème cette fois-ci lié au capteur de proximité, les câbles étaient dessoudés et donc on recevait de mauvaises données, nous avons donc totalement ressoudé les câbles.

8) Conclusion

En conclusion, notre projet s'est transformé en voiture autonome avec ses deux capteurs de proximité.
La partie algorithmie a sûrement été la plus longue et la plus difficile, principalement dû aux différents problèmes du GPS pour lesquels on ne trouvait pas de réelles solutions et qui ducoup nous bloquait dans l'avancement global du projet. Mais aujourd'hui la voiture est bel et bien autonome grâce au programme qu'on a créé au fil des dernières semaines.