Développement objet C++

Projet : Prototype de voiture intelligente

Etudiant : Mahmod Alhabaj

Professeur référent : Labbaci Hemza

Année universitaire : 2018-2019

Table des matières

I. Présentation du projet

Durant l'enseignement Développement objet C++ de la L2 informatique, j'ai été amené à réaliser un prototype de voiture intelligente, capable de se déplacer en autonomie dans un circuit grâce aux données fournies en temps réel par une caméra, et des capteurs.

Ce projet s'est déroulé de février au 6 mai 2019. Il a été proposé au professeur référent M. LABBACI Hemza qui l'a approuvé en début de 2ème semestre.

J'ai choisi ce projet car je le trouve très complet. En effet, j'ai pu m'initier à l'électronique, le traitement des signaux des capteurs, l'aspect réseau avec l'utilisation des Sockets sous C++ et finalement l'IA avec OpenCV4.

Les notions abordées sont :

-C/C++ et développement objet.

-OpenCV 4 pour le traitement d'images et IA.

-Raspberry Pi 3B sous Raspbian-Stretch.

-Traitement d'informations et utilisation de différent types de capteurs (ultrason, infrarouge...).

-Création et utilisation des Sockets sous C/C++ Windows et linux.

-Gestion d'Energie et conception/réalisation de circuits électriques.

-Réalisation d'un module permettant à la voiture de se recharger (par induction) en temps réel sur le circuit (Recherches et expériences effectuées mais module non construit).

II. Développement et réalisation

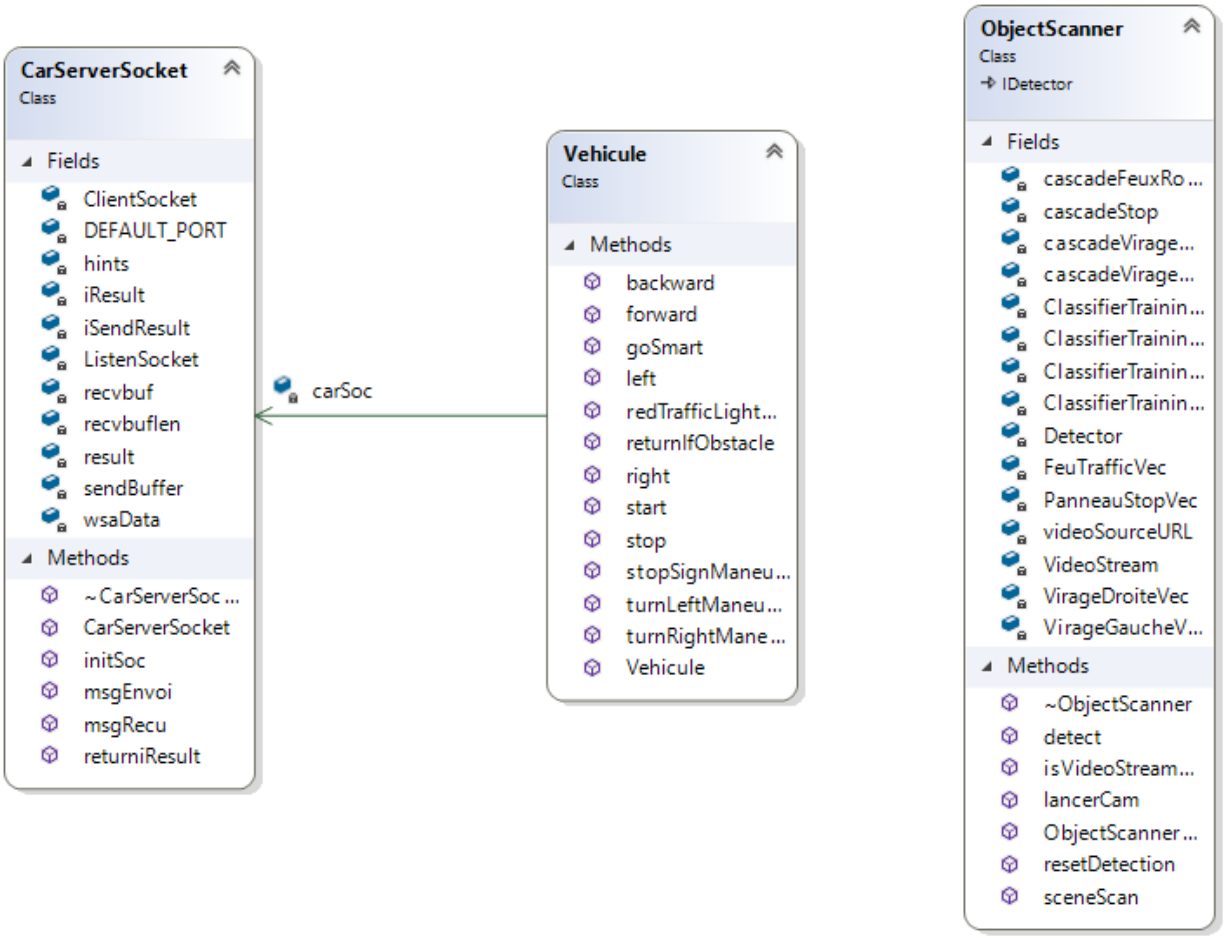
Lors de la conception, j'envisageais de mettre la totalité du traitement d'informations et la prise des actions dans la Raspberry. Puis, d'après plusieurs tests, j'ai constaté que les performances fournies par le Raspberry n'étaient pas au rendez-vous, surtout pour la partie IA

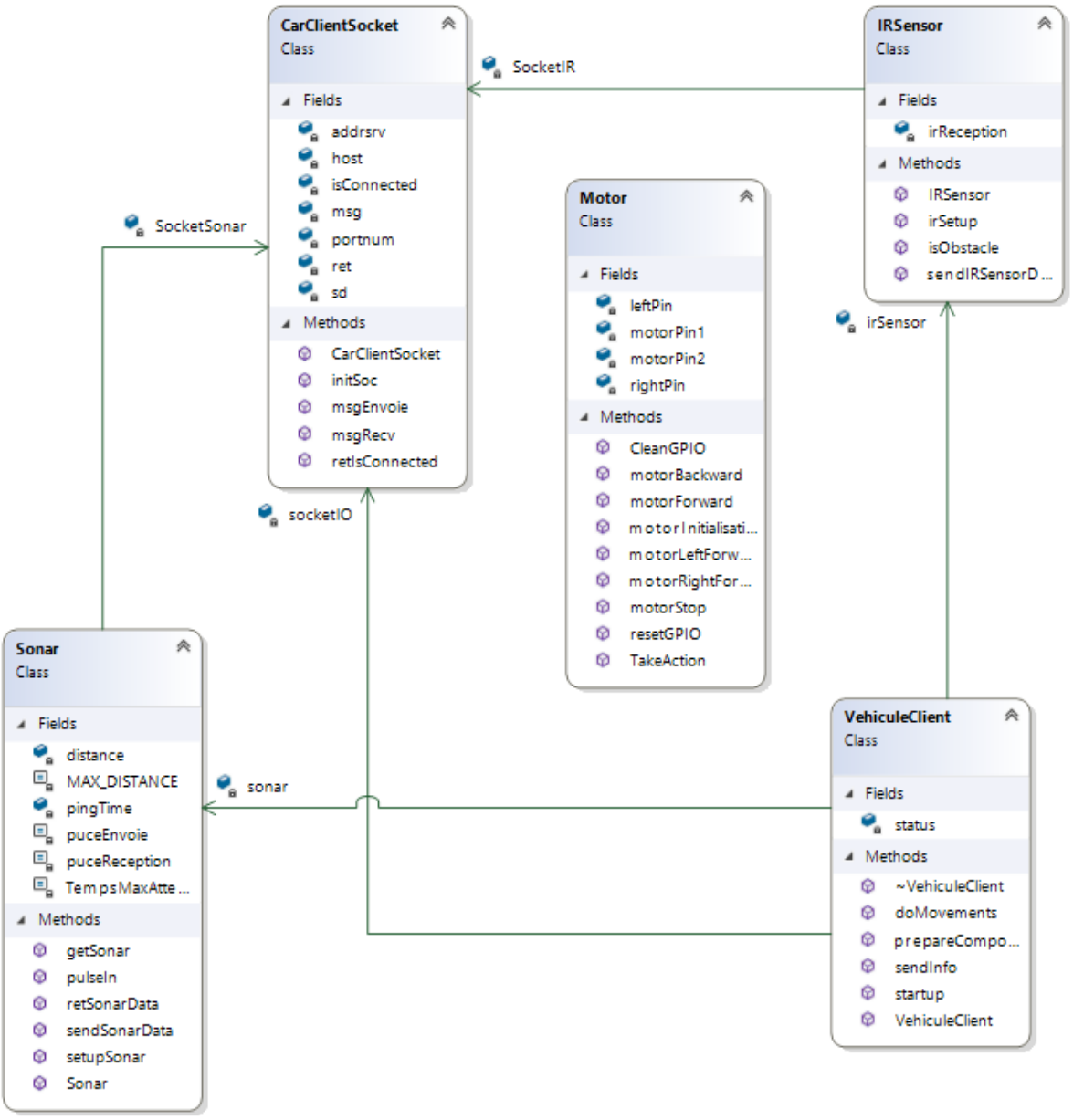
Le fonctionnement se divise en deux parties :

-La partie ordinateur externe qui peut représenter le serveur ou bien l'ordinateur de bord de la voiture.

-La partie Raspberry et la voiture RC qui représentent la voiture ; elle-même sert comme vecteur de traitement d'informations de capteurs, puis d'envoi vers l'ordinateur de bord pour finalement attendre les instructions qui seront envoyées par celui-ci.

Voici le diagramme de classes de l'ensemble du programme :




CarClientSocket
Class
Fields
addrsrv
host
isConnected
msg
portnum
ret
sd
Methods
CarClientSocket
initSoc
msgEnvoie
msgRecv
retIsConnected
SocketIR
IRSensor
Class
Fields
irReception
Methods
IRSensor
irSetup
isObstacle
sendIRSensorD...
SocketSonar
Motor
Class
Fields
leftPin
motorPin1
motorPin2
rightPin
Methods
CleanGPIO
motorBackward
motorForward
motorInitialisati...
motorLeftForw...
motorRightFor...
motorStop
resetGPIO
TakeAction
irSensor
socketIO
Sonar
Class
Fields
distance
MAX_DISTANCE
pingTime
puceEnvoie
puceReception
TempsMaxAtte...
Methods
getSonar
pulseIn
retSonarData
sendSonarData
setupSonar
Sonar
sonar
VehiculeClient
Class
Fields
status
Methods
~VehiculeClient
doMovements
prepareCompo...
sendInfo
startup
VehiculeClient

1. Ordinateur de bord

L'ordinateur de bord externe va permettre le traitement des informations fournies par la voiture.

a) TRAITEMENT D'IMAGES ET DETECTION D'OBJETS

1- Présentation des méthodes utilisées

Le but premier du projet étant de pouvoir reconnaître les différents panneaux, une analyse de ces objets est requise afin de comprendre et savoir choisir la méthode de détection la plus adaptée à la situation.

Généralement, un panneau est un objet fixe de forme fixe, c'est-à-dire qu'il ne change pas de forme, de sens ou d'apparence au cours du temps (pour faciliter le déroulement du projet, nous n'utiliserons que les panneaux suivant le système européen).

De plus, des panneaux comme le STOP possèdent une forme particulière, ce qui signifie que les algorithmes de détection de formes (cercle, rectangle…) sont à éviter.

OpenCV4 fournit plusieurs techniques permettant la détection d'objets dans une image.

En tenant compte des capacités du matériel disponible, et le besoin du meilleur taux d'images analysées par seconde, la détection d'objets en ***utilisant les caractéristiques pseudo-Haar & Cascade de classificateurs*** est la plus adaptée.

La technique des caractéristiques pseudo-Haar a été introduite par Paul Viola et Michael Jones en 2001 dans l'article « Robust Real-time Object Detection » et elle consiste à prendre une image, dans laquelle nous allons définir des zones rectangulaires adjacents. Ensuite, nous calculons la somme des intensités des pixels de l'image dans ces zones. La caractéristique pseudo-Haar sera donnée en faisant la différence entre les rectangles noirs et blancs (on calcule la somme de pixels délimités par la zone sombre que l'on soustrait à la somme des pixels délimités par la zone claire).

(1) (2) (3) (4)

Dans openCV4, le « Cascade Classifier » est un ensemble de classificateurs simples « stages » qui sont appliqués successivement sur une région d'intérêt (ici un panneau STOP par exemple) dans une image. La détection se fait ici en rejetant toutes les zones où on ne trouve pas l'objet recherché. Si un objet est susceptible d'être trouvé par le premier classificateur, le suivant va rentrer en action, jusqu'au dernier.

Nous utilisons cet outil afin de réduire au maximum les temps de recherche. C'est-à-dire qu'au lieu de chercher dans chaque portion de l'image des similitudes avec l'objet, faire le test et les calculs avec disons plus de 50000 caractéristiques, nous allons séparer ces caractéristiques en plusieurs « stages ». Les premiers « stages » sont les moins précis et contiennent donc le moins d'informations, et plus on valide un « stage », plus le suivant contiendra de caractéristiques. Dès qu'un « stage » échoue, nous passons à la prochaine partie de l'image.

La partie de l'image qui aura passé tous les « stages » est une zone contenant l'objet. Ainsi, nous aurons gagné énormément en temps de calculs (presque 50% de gains de performance d'après certaines sources).

2- Entraînement des cascades

L'entraînement de ces classificateurs se fait en 3 étapes :

D'abord nous allons réunir un maximum d'images (au mieux une centaine) contenant l'objet recherché (par exemple un panneau « virage à droite ») et nous créons un fichier texte recensant les images avec leurs noms.

On crée également un fichier « .vec » regroupant ces images à l'aide du programme « sample_creation » de openCV4.

Ensuite, on refait la même chose dans un autre dossier dans lequel nous allons cette fois-ci mettre une centaine d'images dites « négatives », c'est-à-dire ne contenant pas cet objet.

Pour faciliter la création de la liste des noms des fichiers, on peut utiliser une ligne de commande pour automatiser l'action : « dir /b *.jpg *.png >ListeNegatives.txt ».

La prochaine étape est la plus longue ; en effet, nous allons lancer le programme « haartraining » fourni par OpenCV4, en lui passant en paramètres le nombre de « stages » à faire, la mémoire vive allouée à l'entraînement, le nombre de fichiers positifs, négatifs, les listes des images et le « .vec » crée précédemment.

Voici à quoi peut ressembler cette ligne de commande sous Windows :

```
haartraining.exe -data cascades -vec vector/vector.vec -bg negative/bg.txt -npos 15 -nneg 3020 -nstages 15 -mem 1024 -mode ALL -w 24 -h 24
```

Cette opération peut prendre plus ou moins longtemps (des minutes, des heures, voire des jours) en fonction du nombre d'images fournies. On remarquera que plus on donne d'images positives/négatives, plus cela prendra du temps, de mémoire, mais aussi plus la détection sera précise.

Finalement, nous utilisons le programme « haarconv » qui va assembler les différents stages en un seul pour obtenir un .xml final fonctionnel.

```
haarconv.exe cascades myhaar.xml 24 24
```

Bien entendu, il faut refaire ces étapes pour chaque objet. Dans notre cas présent, on obtiendra un .XML pour chaque panneau.

3- Implémentation dans C++

Nous cherchons à réaliser une Class permettant de se connecter à la caméra de bord et récupérer les images en temps réel pour la détection. A la fin de la détection, la fonction va assigner une valeur qui identifie l'objet détecté (cf. image ci-dessus) à une variable globale, appelée « ScannerStatus ».

```
static int ScannerStatus = -1;
//////////////////////////
const int NothingDetected = 1;
const int stopDetected = 3;
const int trafficLightsDetected = 4;
const int turnRightDetected = 5;
const int turnLeftDetected = 6;
```

Les états définissant ScannerStatus 1

On commence d'abord par donner les chemins d'accès vers les fichiers .XML et on crée des vecteurs pour chaque objet :

```
std::vector<cv::Rect> PanneauStopVec, FeuTrafficVec, VirageDroiteVec, VirageGaucheVec;
std::string ClassifierTrainingStop = "C:/Users/mhaba/OneDrive/Desktop/StopSign.xml"; //URL VERS LES DONNEES DE LENTRAINEMENT stop.
std::string ClassifierTrainingFeuRouge = "C:/Users/mhaba/OneDrive/Desktop/TrafficLights.xml"; //URL VERS LES DONNEES DE LENTRAINEMENT feu rouge.
std::string ClassifierTrainingVirageDroite = "C:/Users/mhaba/OneDrive/Desktop/DirectionRight.xml"; //URL VERS LES DONNEES DE LENTRAINEMENT stop.
std::string ClassifierTrainingVirageGauche = "C:/Users/mhaba/OneDrive/Desktop/DirectionLeft.xml"; //URL VERS LES DONNEES DE LENTRAINEMENT feu rouge.
```

On ouvre chaque fichier avec la fonction « findFile » et on définit les pointeurs sur Cascade Classifieur :

```
string fichierXmlCascadeStop = samples::findFile(this->ClassifierTrainingStop);
string fichierXmlCascadeRouge = samples::findFile(this->ClassifierTrainingFeuRouge);
string fichierXmlVirageDroite = samples::findFile(this->ClassifierTrainingVirageDroite);
string fichierXmlVirageGauche = samples::findFile(this->ClassifierTrainingVirageGauche);
```

```
cv::Ptr<cv::CascadeClassifier> cascadeStop;
cv::Ptr<cv::CascadeClassifier> cascadeFeuxRouge;
cv::Ptr<cv::CascadeClassifier> cascadeVirageDroite;
cv::Ptr<cv::CascadeClassifier> cascadeVirageGauche;
```

```
cascadeStop = makePtr<CascadeClassifier>(fichierXmlCascadeStop);
cascadeFeuxRouge = makePtr<CascadeClassifier>(fichierXmlCascadeRouge);
cascadeVirageDroite = makePtr<CascadeClassifier>(fichierXmlVirageDroite);
cascadeVirageGauche = makePtr<CascadeClassifier>(fichierXmlVirageGauche);
```

Ensuite, on crée un pointeur de type Ptr< DetectionBasedTracker::IDetector> pour chaque string des fichiers XML des objets à détecter. Ptr<T> est un pointeur sur l'objet T. La particularité de ce pointeur est que l'objet sera automatiquement détruit quand il n'est plus mentionné par aucun Ptr.

```
Ptr<DetectionBasedTracker::IDetector> detectStopPrincipale = makePtr<ObjectScanner>(cascadeStop);
Ptr<DetectionBasedTracker::IDetector> detectFeuxRougePrincipale = makePtr<ObjectScanner>(cascadeFeuxRouge);
Ptr<DetectionBasedTracker::IDetector> detectVirageDroitePrincipale = makePtr<ObjectScanner>(cascadeVirageDroite);
Ptr<DetectionBasedTracker::IDetector> detectVirageGauchePrincipale = makePtr<ObjectScanner>(cascadeVirageGauche);
```

« DetectionBasedTracker::IDetector » est la class/Objet qui applique la détection d'objets avec caractéristiques pseudo-Haar (fonctionnement expliqué précédemment).

IDetector requiert deux « détecteurs », un principal, et l'autre dit « tracking ». On crée alors le deuxième :

```
Ptr<DetectionBasedTracker::IDetector> DetecteurStopTrack = makePtr<ObjectScanner>(cascadeStop);
Ptr<DetectionBasedTracker::IDetector> DetecteurFeuxRougeTrack = makePtr<ObjectScanner>(cascadeFeuxRouge);
Ptr<DetectionBasedTracker::IDetector> DetecteurVirageDroiteTrack = makePtr<ObjectScanner>(cascadeVirageDroite);
Ptr<DetectionBasedTracker::IDetector> DetecteurVirageGaucheTrack = makePtr<ObjectScanner>(cascadeVirageGauche);
```

On initialise l'objet « Parameters » (on laisse les paramètres par défaut) :

```
DetectionBasedTracker::Parameters params;
```

On crée un objet « DetectionBasedTracker » pour chaque panneau/objet à détecter en passant en paramètres les deux détecteurs (principal et tracking) et l'objet « params ».

```
DetectionBasedTracker DetectorStop(detectStopPrincipale, DetecteurStopTrack, params);
DetectionBasedTracker DetectorFeuxRouge(detectFeuxRougePrincipale, DetecteurFeuxRougeTrack, params);
DetectionBasedTracker DetectorVirageDroite(detectVirageDroitePrincipale, DetecteurVirageDroiteTrack, params);
DetectionBasedTracker DetectorVirageGauche(detectVirageGauchePrincipale, DetecteurVirageGaucheTrack, params);
```

Ensuite, nous allons initialiser deux objets de type Mat (matrice), qui en OpenCV4 représentent l'image. Le premier objet Mat sera pour les frames reçus par caméra, et le deuxième sera pour les frames modifiés.

Nous effectuons des modifications sur les images reçues en flux vidéo :

Chaque image sera transformée en niveaux de gris « COLOR_BGR2GRAY » grâce à la fonction « cvtColor » qui prend l'image dans le Mat Frame reçu, le modifie puis l'injecte dans le Mat Frame Modifié.

```
Mat FrameInitiale;
Mat FrameModifiee;

while (waitKey(30) < 0)
{
    VideoStream >> FrameInitiale;
    cvtColor(FrameInitiale, FrameModifiee, COLOR_BGR2GRAY);
```

Désormais, nous pouvons lancer la fonction « process (Frame Modifiée) » sur les différents Detectors instanciés précédemment. Cette fonction va donc chercher les zones d'intérêt dans chaque image et donc essayer de trouver l'objet.

Dans le cas où un objet est trouvé, on applique la fonction « getObjects (vecteur objet) » qui va retourner dans le vecteur correspondant à chaque objet, les coordonnés de celle-ci dans l'image.

Nous affichons alors l'objet, entouré d'un rectangle crée à l'aide de la fonction « rectangle » d'OpenCV.

```
VideoStream >> FrameInitiale;
cvtColor(FrameInitiale, FrameModifiee, COLOR_BGR2GRAY);
////////////Feu Rouge:
DetectorFeuxRouge.process(FrameModifiee);
DetectorFeuxRouge.getObjects(FeuTrafficVec);
/////////////////Virags Droite:
DetectorVirageDroite.process(FrameModifiee);
DetectorVirageDroite.getObjects(VirageDroiteVec);
///////////////Virages Gauche:
DetectorVirageGauche.process(FrameModifiee);
DetectorVirageGauche.getObjects(VirageGaucheVec);
///////////////PanneauStop:
DetectorStop.process(FrameModifiee);
DetectorStop.getObjects(PanneauStopVec);



for (size_t i = 0; i < PanneauStopVec.size(); i++)
{
    rectangle(FrameInitiale, PanneauStopVec[i], Scalar(10, 100, 0));
}
for (size_t i = 0; i < FeuTrafficVec.size(); i++)
{
    rectangle(FrameInitiale, FeuTrafficVec[i], Scalar(50, 400, 10));
}
for (size_t i = 0; i < VirageDroiteVec.size(); i++)
{
    rectangle(FrameInitiale, VirageDroiteVec[i], Scalar(50, 400, 10));
}
for (size_t i = 0; i < VirageGaucheVec.size(); i++)
{
    rectangle(FrameInitiale, VirageGaucheVec[i], Scalar(50, 400, 10));
}
```

Ainsi, dès que la taille d'un vecteur est supérieure à zéro, on sait qu'il existe un objet dans le champ visuel de la voiture, et donc on modifie la valeur globale « ScannerStatus » présentée précédemment.

```
if (this->PanneauStopVec.size() > 0)
{

    resetDetection();
    //cout << "\033[1;31m \n------------\STOP DETECTE\n---------- \033[0m\n" << endl;
    ScannerStatus = stopDetected;
}
else if (this->FeuTrafficVec.size() > 0)
{

    resetDetection();
    //cout << "\033[1;31m \n------------\TRAFFIC LIGHT DETECTE\n---------- \033[0m\n" << endl;
    ScannerStatus = trafficLightsDetected;
}
else if (this->VirageDroiteVec.size() > 0)
{

    resetDetection();
    //cout << "\033[1;31m \n------------\VIRAGE DROIT DETECTE\n---------- \033[0m\n" << endl;
    ScannerStatus = turnRightDetected;
}else if (this->VirageGaucheVec.size() > 0)
{

    resetDetection();
    //cout << "\033[1;31m \n------------\VIRAGE GAUCHE DETECTE\n---------- \033[0m\n" << endl;
    ScannerStatus = turnLeftDetected;
}
else if (this->PanneauStopVec.size() <= 0 && this->FeuTrafficVec.size() <= 0
    && this->VirageDroiteVec.size()<=0 && this->VirageGaucheVec.size()<=0)
{
    resetDetection();
    ScannerStatus = NothingDetected;
```

Nous obtenons donc une fonction « sceneScan() » capable d'analyser le flux vidéo à plus de 15 images/seconde. Cette fonction sera exécutée indépendamment du reste du programme afin d'avoir un Scan continu sur les objets rencontrés par la voiture. Cette exécution sera effectuée donc dans un thread différent.

```
ObjectScanner s1;
thread t(&ObjectScanner::sceneScan, s1);
```

sceneScan() exécutée sur un thread différent

b) SOCKETS ET COMMUNICATION AVEC LA VOITURE

Nous cherchons à communiquer avec le Raspberry dans la voiture, pour envoyer des instructions/commandes de déplacements et recevoir des informations sur l'environnement entourant la voiture pour traitement.

Remarque : Cette partie n'existerait pas s'il était possible d'implémenter le program dans le Raspberry, mais, comme expliqué précédemment, les performances sont très limitées.

Il existe deux parties pour les communications. Ici nous allons voir le côté serveur, puis nous traiterons le côté voiture dans la partie correspondante.

La partie Serveur (Windows) utilise la librairie « Winsock » et « Winsock 2 » fournie par Microsoft et incluse dans Windows Studio (Librairies de base C++).

1- Initialisation du Socket

Tout d'abord, nous créons un objet « WSADATA » qui contient les informations sur l'implémentation du Socket sous Windows.

On initialise cet objet Winsock avec « WSAStartup » qui retourne zéro si l'initialisation s'est bien déroulée, sinon d'autres codes erreurs sont retrouvables sur docs.microsoft.com .

```
iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
if (iResult != 0) {
    printf("Error init Socket WSA : %d\n", iResult);
    return 1;
}
```

Cette fonction prend en paramètres la version de Winsock requise et l'adresse de l'objet WSADATA créée précédemment.

Ensuite, nous allons définir les réglages du Socket pour correspondre à un socket de serveur.

-ai_family = AF_INET qui signifie l'utilisation du protocole IP de la famille IPv4.

-hints.ai_socktype = SOCK_STREAM signifie que c'est un socket de stream.

-hints.ai_protocol = IPPROTO_TCP signifie le choix du protocole réseau choisi , ici TCP.

Nous allons maintenant appeler la fonction « getaddrinfo » qui va prendre en compte les paramètres du socket définis précédemment et le port choisi. Cette fonction retournera zéro s'il n'y a pas d'erreurs, sinon un code erreur. Finalement on crée un socket vide.

```
//les params de la socket:
ZeroMemory(&hints, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
hints.ai_flags = AI_PASSIVE;

// Initialisaation address local du serveur et le PORT.
iResult = getaddrinfo(NULL, DEFAULT_PORT, &hints, &result);
if (iResult != 0) {
    printf("error fonction getaddrinfo: %d\n", iResult);
    WSACleanup();
    return 1;
```

```
// Creation de socket:
ListenSocket = socket(result->ai_family, result->ai_socktype, result->ai_protocol);
if (ListenSocket == INVALID_SOCKET) {
    printf(" error: %ld\n", WSAGetLastError());
    freeaddrinfo(result);
    WSACleanup();
    return 1;
}
```

A cette étape, nous procédons à l'initialisation du protocole TCP dans le socket. On appelle la fonction « bind » en passant en paramètres le socket créé et les paramètres du socket.

```
// Initialisation du protocole TCP du Socket:
iResult = ::bind(ListenSocket, result->ai_addr, (int)result->ai_addrlen);
if (iResult == SOCKET_ERROR) {
    printf("error bind(): %d\n", WSAGetLastError());
    freeaddrinfo(result);
    closesocket(ListenSocket);
    WSACleanup();
    return 1;
}
```

Nous avons donc lié le Socket à l'adresse IP et le port du server. Maintenant le serveur doit écouter sur le port et attendre pour qu'une connexion s'engage.

Pour écouter sur un socket, nous utilisons la fonction « listen » qui prend en paramètre le socket et un nombre de connexions simultanées en queue.

```
iResult = listen(ListenSocket, SOMAXCONN);
if (iResult == SOCKET_ERROR) {
    printf("Error listen(): %d\n", WSAGetLastError());
    closesocket(ListenSocket);
    WSACleanup();
    return 1;
}
```

Désormais, pour accepter les connexions, nous allons créer un socket temporaire, et utiliser la fonction « accept ».

```
// Lors qu'un client tent de se connecter:
if (ClientSocket = accept(ListenSocket, NULL, NULL)) printf("\nclient initialisee \n");
if (ClientSocket == INVALID_SOCKET) {
    printf("accept failed with error: %d\n", WSAGetLastError());
    closesocket(ListenSocket);
    WSACleanup();
    return 1;
}
```

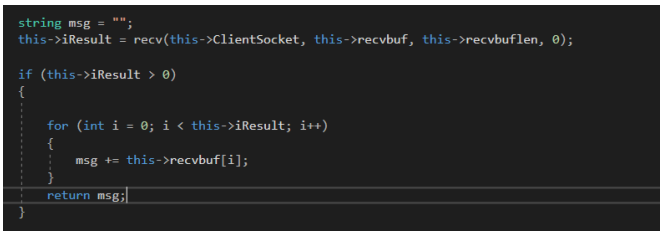
Donc le socket est initialisé et le serveur est prêt à recevoir et envoyer des données.

2- Envoi et réception (Serveur)

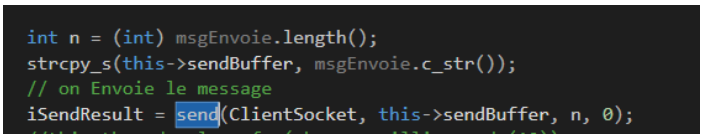
Afin de recevoir les informations, nous créons la fonction « msgRecu ». Cette fonction permet de recevoir les données émises par le client.

Pour faire ceci, nous utilisons la fonction « recv » de Winsock. Elle prend en paramètres un socket, un buffer d'arrivée qui va stocker temporairement les données du message dans un tableau de chars et un entier représentant la taille du message.

Finalement on convertit le tout dans un String qui sera retourné par la fonction.



Pour l'envoi, la méthode « msgEnvoi » va utiliser la fonction « send » qui prend en paramètres le socket, ainsi qu'un buffer en forme d'un tableau de chars et la taille de ce tableau.



En utilisant ces méthodes d'envoi et de réception, on arrive à une latence réseau quasiment nulle grâce à la simplicité des données envoyées, ainsi que l'efficience des méthodes employées.

Dès lors que nous pouvons analyser les images, détecter les objets, recevoir les données et les envoyer, nous pouvons donc procéder à la prise de décision.

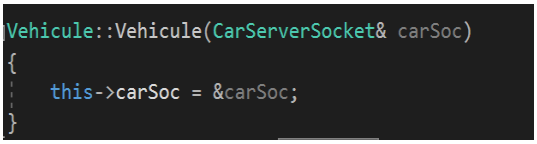
c) PRISE DE DECISION

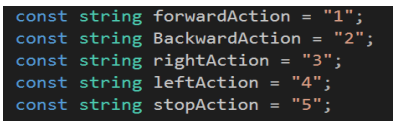
Cette prise de décision sera assurée par la class Véhicule. Cette class est composée de 3 types de méthodes :

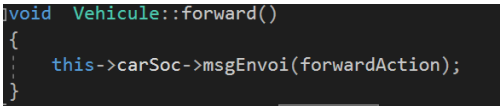
1- Commandes de déplacements

Les fonctions de déplacements principaux telles que : avancer, reculer, tourner à gauche, tourner à droite et arrêt du véhicule.

En sachant que l'initialisation de la Class dépend de l'existence d'un socket (cf. image ci-dessus), l'implémentation des commandes de déplacement se fait par un envoi de message contenant l'entier attribué à chaque déplacement :







2- Gestion d'obstacles

Le rôle de la fonction « returnIfObstacle » est de recevoir les alertes envoyées par la voiture, qui envoie « 1 » si un obstacle est repéré. Cette opération sera traitée ultérieurement dans la partie voiture/client.

Donc si un obstacle est présent, la fonction retourne 1, sinon zéro.

La voiture possède actuellement seulement un sonar, donc elle ne peut voir que devant, sans prendre en compte les voitures qui peuvent venir sur les côtés. Donc cette fonction évoluera lors des futurs ajouts d'autres détecteurs.

3- Manœuvres et actions

Chaque panneau où type d'obstacle possède une manœuvre. Ces manœuvres prédéfinies seront remplacées dans le futur par de la prise de décision IA avec du CNN.

Par exemple, la manœuvre du panneau Stop correspond à l'arrêt devant ce panneau, attendre deux secondes, puis s'il n'y a pas d'obstacles, continuer son itinéraire.

```
this->stop();
Sleep(2000);
while (returnIfObstacle() == 1) {
    cout << "Obstacle detectee, On reste sur place" << endl;
}
this->forward();
```

Les autres manœuvres ont le même pattern :

(detection → action→detection→action).

4- Prise de décisions

La fonction « goSmart » est en partie le chef d'orchestre. Elle va prendre en compte toutes les données, comme le «ScannerStatus » actuel, la potentielle présence d'obstacles, et en fonction de ceci, prendre l'action nécessaire dans chaque situation possible (manœuvres, arrêts…etc). Cette méthode pourra être également remplacée par une prise de décisions faite par l'IA ayant eu un entraînement sur plusieurs pistes et scénarios.

Ainsi, la partie serveur est terminée. Le serveur est désormais prêt à analyser l'environnement de la voiture, et la diriger.

2. Véhicule et Raspberry.

La voiture, commandée par l'ordinateur de bord, va parcourir le chemin, tout en gardant l'ordinateur de bord au courant de ce qui se passe autour d'elle.

a) SOCKETS ET COMMUNICATIONS AVEC LE SERVEUR

La communication Voiture-Ordinateur de bord est très similaire au cas inverse expliqué précédemment. En effet, étant sur linux, Winsock n'est plus utilisé mais la librairie fournie par le système « sys/socket.h » va prendre le relais. L'ensemble des fonctions est presque identique des deux côtés, mais ce qui change est qu'ici, nous sommes en mode « Client » et donc on se connecte sur une adresse IP spécifique au serveur et le port de celui-ci. Ceci se fait en utilisant la fonction « connect» et les paramètres qui sont aussi presque identiques à la partie Serveur.

```
if ((host = gethostbyname(hostname)) == NULL) {
    herror("gethostbyname");
    this->isConnected = false;
    exit(-1);
}

bcopy(host->h_addr, (char *)&addrsrv.sin_addr, host->h_length);
addrsrv.sin_family = AF_INET;
addrsrv.sin_port = htons(portnum);

connect(sd, (const struct sockaddr*)&addrsrv, sizeof(addrsrv));
this->isConnected = true;
return true;
```

Les fonctions d'envoi et de réception restent les mêmes également.

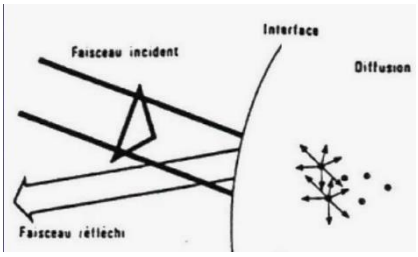
b) WIRING PI, CAPTEURS ET DEPLACEMENTS

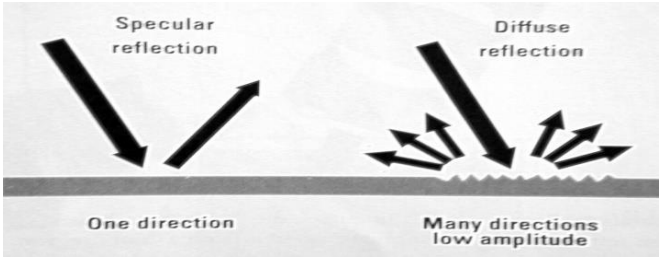
WiringPi est une librairie permettant au Raspberry de manipuler des modules électroniques (des Leds, des capteurs, des moteurs...etc) via ses pins GPIO.

1- Configuration et fonctionnement du Sonar (Ultrason)

Une onde sonore est une onde mécanique. Elle nécessite un milieu matériel pour pouvoir se déplacer. Lorsqu'une onde passe d'un milieu à un autre, une partie de l'énergie incidente est transmise et l'autre partie est réfléchie au niveau de l'interface séparant ces deux milieux.

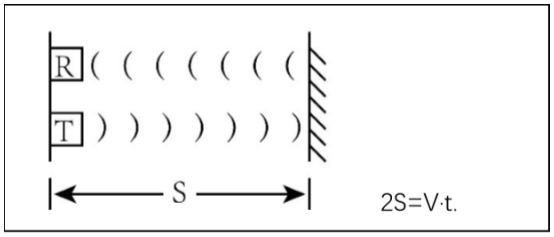
Nous savons également qu'un faisceau réfléchi repart avec un angle identique à l'angle d'incidence. Nous remarquons par ailleurs, que si la réflexion a lieu sur une surface irrégulière, elle sera omnidirectionnelle et de faible amplitude.





Le faisceau transmis ne conserve sa direction initiale que dans le cas où il est perpendiculaire à l'interface. Dans tous les autres cas il est partiellement dévié.

Le fonctionnement du module ultrasonique suit donc ces principes physiques. En effet, le module est composé d'un émetteur qui va convertir les signaux électriques (énergie électrique) en ondes sonores (énergie mécanique) et d'un récepteur qui sera chargé de faire l'inverse.



Pour trouver la distance entre le module et un obstacle potentiel, nous allons émettre une onde et en même temps démarrer le chronomètre. Cette onde va se propager jusqu'à rencontrer un objet. Dès que l'onde touche l'objet, elle est réfléchie et retourne donc vers l'endroit où elle est émise. C'est à ce moment là que le récepteur va détecter cette onde et va arrêter le chronomètre.

Nous savons que la vitesse du son dans l'air et d'environ 340m/s. Alors, on applique la formule :

*Vitesse = 2*distance / temps.*

Ici on cherche la distance, en connaissant la vitesse et le temps (chronométrés) :

*Distance = (Vitesse * temps)/2 .*

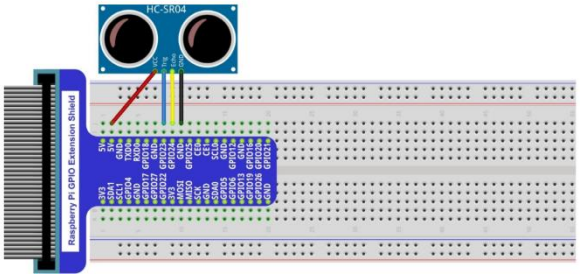
L'interface du Sonar est composée de 4 pins :

VCC	➔	Vcc est le pin d'alimentation. Il a besoin de 5V et 12mA pour fonctionner.
Trig	➔	Trig est le pin traitant l'émetteur.
Echo	➔	Echo est le pin traitant le récepteur.
GND	➔	GND est le pin terre.

Ce module mesure une distance minimale de 2cm, et va aller jusqu'à 200 cm. Dans notre application, nous irons jusqu'à 60cm pour diminuer les erreurs de calcules et le taux de pertes d'ondes.

Nous allons allouer et connecter les pins entrée et sortie aux GPIO 23 et 24.

```
class Sonar {
private:
    const int puceEnvoie = 23;
    const int puceReception = 24;
    const int MAX_DISTANCE = 60; //50 à cause des pertes et des faux-positives.
```



La méthode « setupSonar » sera chargée d'initialiser les ports dans leurs modes respectifs :

```
int Sonar::setupSonar()
{
    if (wiringPiSetupGpio() == -1) {

        ("Erreur wiringPi Setup !");
        return 1;
    }
    pinMode(puceEnvoie, OUTPUT);
    pinMode(puceReception, INPUT);
}
```

On sait également qu'une onde peut être perdue, donc, pour éviter d'attendre que chaque onde revienne obligatoirement, nous allons mettre un seuil, un temps maximum d'attente avant de laisser tomber l'onde :

TempsMax d'attente = 2*MAX_DISTANCE/100/340*1000000.

« 1/100/340*1000000 » ici est égale à 58.7 donc :

```
const int MAX_DISTANCE = 60; //50 à cause des p
const int TempsMaxAttente = MAX_DISTANCE * 60;/
```

La fonction « getSonar » va implémenter le fonctionnement de celui-ci. Elle va calculer le la distance requise et va la retourner en cm.

Nous allons d'abord dire à l'émetteur d'émettre une onde pendant 35 microsecondes. Ensuite, on appelle la fonction « pulseIn » qui va chronométrer le temps de retour du signal et donc écouter le récepteur. Quand le signal est reçu et est plus grand que 2, on applique la formule de la distance présentée précédemment et la fonction retourne la valeur de la distance.

```
double Sonar::getSonar()
{
    delay(35);

    digitalWrite(this->puceEnvoie, HIGH);
    delayMicroseconds(35);
    digitalWrite(this->puceEnvoie, LOW);
    pingTime = pulseIn(this->puceReception, HIGH, this->TempsMaxAttente);
    if (this->pingTime <= 1) {
        return -1.0; //trop proche ou trop loin
    }
    else {
        distance = pingTime * 340.0 / 2.0 / 10000.0;
        return distance;
    }
}
```

Finalement, la fonction « retSonarData » se charge du tri des valeurs et on ne garde que celles inférieures à 40 cm.

Si l'obstacle est à 40cm ou moins, elle retourne 1, sinon zéro.

2- Configuration et fonctionnement du capteur Infrarouge.

Le capteur infrarouge suit les mêmes principes que le capteur ultrason. La lumière est une onde et il y a réflexion.

La différence avec le sonar, est que ce module est déjà automatisé. Donc nous ne pouvons pas effectuer de calculs de distances. Nous pouvons seulement savoir si un obstacle est présent à quelques centimètres du capteur (la distance de détection est réglable via les potentiomètres intégrés au capteur). D'autres types de modules existent, mais ils ne sont pas à disposition pour ce projet.

Le module a besoin de 3.3V voire 5V pour un bon fonctionnement. Et on a un pin envoyant des données binaires, 0 pour obstacle détecté, sinon rien.

Ici il lui est attribué le port GPIO4. C'est un pin qui sera configuré en mode INPUT (réception).

```
int IRSensor::irSetup()
{
    if (wiringPiSetupGpio() == -1) {
        printf("Erreur WIring Pi IRSensor!\n");
        return 1;
    }

    pinMode(irReception, INPUT);
}
```

La fonction « isObstacle » retourne 1 si un obstacle est détecté, 0 sinon.

```
int IRSensor::isObstacle() {
    delay(25);
    if (digitalRead(irReception) == 0) {
        delay(25);
        if (digitalRead(irReception) == 0) {
            printf("OBSTACLE DETECTEE !! \n");
            return 1;
        }
        else {
            return 0;
        }
    }
}
```

3- Alimentation et gestion des moteurs.

La voiture est équipée de 2 moteurs. Le premier est chargé d'avancer et de reculer et le deuxième joue le rôle d'un « Servo Motor » pour positionner les roues avant en mode à gauche, centre et à droite.

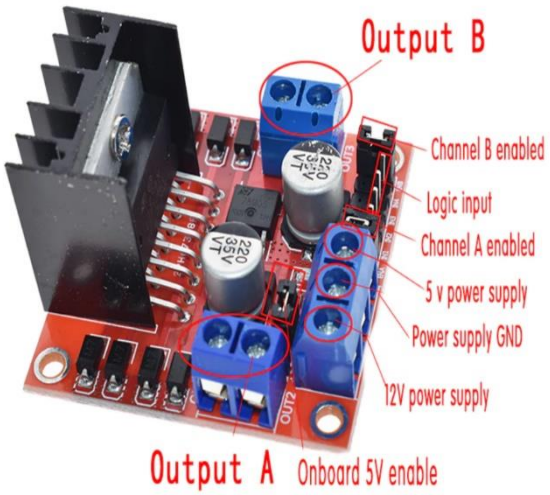
D'abord, il faut savoir qu'on ne peut pas connecter un moteur DC directement au Raspberry. Le moteur a en effet besoin de 9 Volts pour fonctionner, or, le Raspberry ne peut en fournir que 5 au maximum.

De plus, nous voulons alimenter deux moteurs à la fois, donc cela pourra court-circuiter l'appareil.

Il existe une multitude de façons pour faire fonctionner un moteur DC efficacement. Ici, nous utiliserons un «Dual H-Bridge ». C'est un ensemble de transistors pouvant contrôler deux moteurs à la fois.

- Le fonctionnement du pont en H :

Un «pont en H» est simplement un module capable d'inverser la polarité de la tension appliquée à un moteur à courant continu, contrôlant ainsi son sens de rotation.

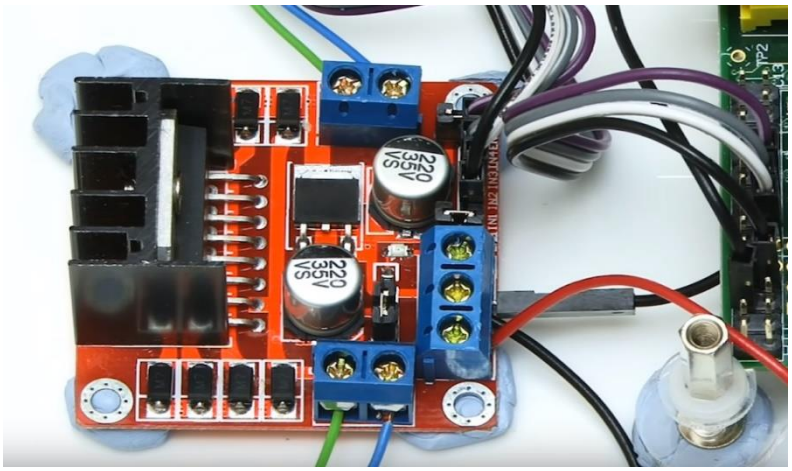


-Output A&B sont les points de connexion avec les moteurs.

-Les pins du « Logic Input » sont ceux qui seront connectés au Raspberry pour permettre de définir la polarité des moteurs. Pin1 et Pin2 représentent les pins OUT1 et OUT2 de la partie OUTPUT A. Pin3 et Pin4 sont liés avec OUTPUT B.

-Dans ce cas, nous utilisons le port 12V sur lequel nous branchons une pile 9V.

-Le port GND est à la fois connecté au Terre de la pile et au « Terre » sur Raspberry.



On initialise maintenant les moteurs dans « WiringPi ». Nous allons donner GPIO 5 et 6 pour avancer et reculer, ainsi que GPIO 12 et 13 pour gauche et droite.

```
    //Parametres Wiring Pi moteurs:
private:
    int motorPin1 = 5;   //forward
    int motorPin2 = 6;   //backward
    int leftPin = 12;    //gauche
    int rightPin = 13;   //droite
```

Pour l'initialisation, nous allons définir tous les pins en mode OUTPUT (car on envoie les données).

```
pinMode(motorPin1, OUTPUT);//set mode for the pin
pinMode(motorPin2, OUTPUT);
pinMode(leftPin, OUTPUT);
pinMode(rightPin, OUTPUT);
```

Ainsi, la fonction « motorInitialisation » retourne 1 s'il n'y a pas eu de problèmes.

Pour chaque commande de déplacement, il existe une combinaison de PINS à contrôler.

Par exemple, pour la commande « motorForward », nous allons changer l'état de chaque pin en « LOW » et le pin correspondant au pôle permettant au moteur d'aller vers l'avant en « HIGH » (on inverse les polarités). Nous obtenons donc ce genre de combinaison :

```
void Motor::motorForward()
{
    digitalWrite(motorPin2, LOW);
    digitalWrite(leftPin, LOW);
    digitalWrite(rightPin, LOW);
    digitalWrite(motorPin1, HIGH);

    printf("Forward...\n");
}
```

Cette méthode est utilisée pour chaque commande de déplacement possible.

Finalement, la méthode « TakeAction » va exécuter la commande de mouvement correspondant au string qu'elle a reçu en paramètre.

```
void Motor::TakeAction(string action)
{
    int n = std::stoi(action);
    cout << "ACTION RECU: " << n << endl;
    switch (n) {
    case 1: Motor::motorForward();  break;
    case 2: Motor::motorBackward(); break;
    case 3: Motor::motorRightForward(); break;
    case 4: Motor::motorLeftForward(); break;
    case 5: Motor::motorStop(); break;
    default: Motor::motorStop(); cout << "Commande deplacement invalide" << endl; break;
```

L'ensemble de méthodes créées, sera utilisé dans la class Véhicule, qui a un simple rôle, notamment la fonction « startup » qui a un but de réunir ces méthodes et les exécuter, c'est-à-dire l'initialisation des capteurs, des moteurs et d'établir la connexion avec le serveur.

III. Recharger une voiture électrique en roulant

Comment peut-on recharger une voiture électrique en roulant ?

Peut-on produire de l'électricité avec un aimant ?

I- Induction :

Qu'est-ce que l'induction électromagnétique ?

L'induction électromagnétique est le processus par lequel un courant est induit, autrement dit "généré" ou "créé", par un champ magnétique variable.

Comment est-elle décrite ?

L'induction électromagnétique est décrite par deux lois :

La Loi de Faraday :

Elle relie la variation du flux magnétique traversant un circuit à la valeur de la force électromotrice ε induite dans ce circuit selon la formule suivante :

$$\varepsilon = -d\Phi/dt$$

-La force électromotrice correspond à la différence de potentiel aux bornes du circuit.

La loi de Lenz :

Tandis que la loi de Faraday donne l'amplitude de la FEM produite, la loi de Lenz renseigne sur le sens que va prendre le courant. Selon cette loi, le sens du courant est toujours tel qu'il va s'opposer à la variation du flux qui lui a donné naissance. Ainsi, tout champ magnétique créé par un courant induit sera de sens opposé à la variation du champ initial.

La loi résultante est appelé ***LENZ-FARADAY***, d'où le « - » qu'on retrouve au début de la formule.

II- Les applications expérimentales :

Afin de démontrer le phénomène, nous avons réalisé plusieurs types d'expériences :

N°1 :

L'expérience de Faraday : induction par un aimant en mouvement dans une bobine :

Matériel :

-Aimant 

-Bobine 

-Ampèremètre : Appareil mesurant l'intensité du courant. 

Manipulation :

On branche la bobine sur l'ampèremètre puis on agite l'aimant au centre de la bobine :

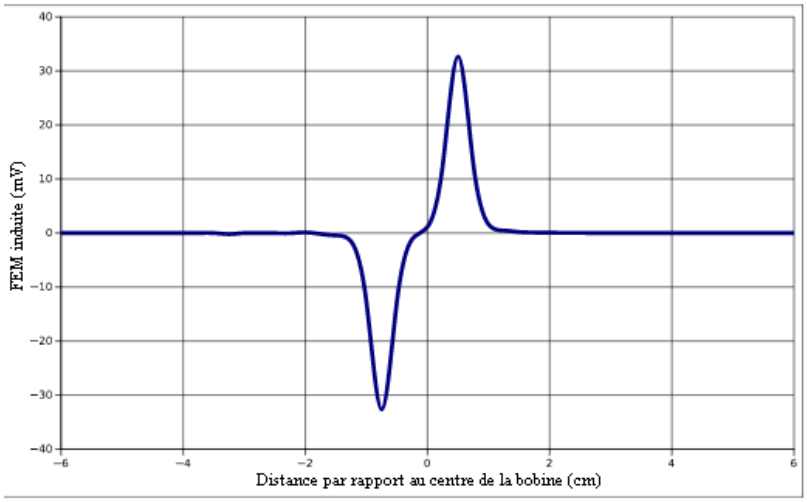
-On observe une faible intensité de courant de l'ordre du micro-ampère.

- On mesure une tension qui augmente de plus en plus jusqu'à ce qu'elle atteigne une valeur Max vers le centre, puis la tension change directement de signe qu'elle garde jusqu'à sa sortie de la bobine. (Signe opposé à celle obtenue en entrant dans la bobine).

Remarque : Plus on agite l'aimant, plus on arrive à avoir une intensité de courant élevé (mais qui reste néanmoins dans le même ordre de grandeur).

-Quand l'aimant est immobile, aucune tension n'est mesurée.

L'exemple de FEM mesurée peut être représenté par le graphique ci-dessous :

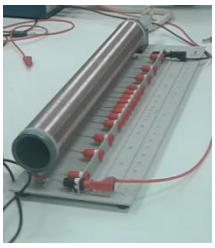


Même si l'aimant immobile produit un champ magnétique important, on n'obtient pas de FEM car celle-ci est le résultat de la variation du flux à travers la bobine.

Expérience N°2 :

Le but de cette expérience est de voir comment créer un champ magnétique dans une bobine.

Matériel :



Bobine

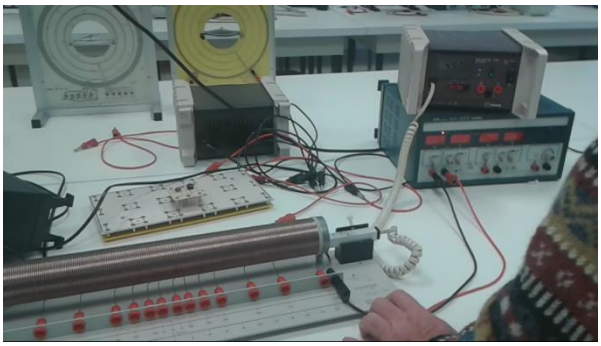
Sonde du Tesla mètre

Tesla Mètre

Générateur

Manipulation :

Dans cette expérience, nous avons branché aux extrémités de la bobine un générateur pour créer un champ magnétique. Nous observons sur le tesla mètre une mesure de l'ordre de 0-10 mT.



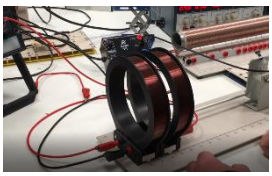
-Dans un second temps, on a voulu multiplier ce champ par deux mais, dans ce cas de manipulation, il faut faire attention car nous pouvons annuler le champ. On a constaté qu'en doublant notre nombre de spires, on double également le champ magnétique.

-Si on inverse le courant venant du générateur, on verra un changement de signe de la valeur indiquée sur le Tesla Mètre :

Expérience n°3 :

But : Allumer une led grâce à l'induction (courant alternatif) :

Matériel :

Deux bobines : 

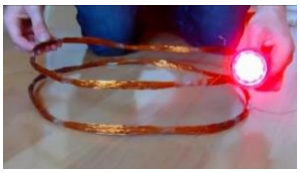
Une led : 

Générateur : 

Manipulation :

Dans cette expérience nous avons voulu allumer une led grâce à deux bobines, une bobine émettrice et une bobine réceptrice ; nous avons essayé dans un premier temps avec un courant continu mais l'intensité du courant était de l'ordre du microampère. Alors nous sommes passés par un courant alternatif.

Pour créer ce courant, nous devons faire bouger la bobine réceptrice, nous avons remarqué que la led rouge qu'on utilise s'allume.



Dans un second temps, nous avons essayé de mettre les deux bobines en émettrices, en n'oubliant pas de faire attention à ne pas annuler le champ.

Nous remarquons de ce fait que l'intensité du courant est plus élevée ; donc nous avons décidé d'ajouter une deuxième led ; nous arrivons donc à allumer ces deux led, voire trois en même temps.

Comment appliquer ce principe dans le rechargement des véhicules électriques ?

III- L'induction dans la voiture électrique et son fonctionnement

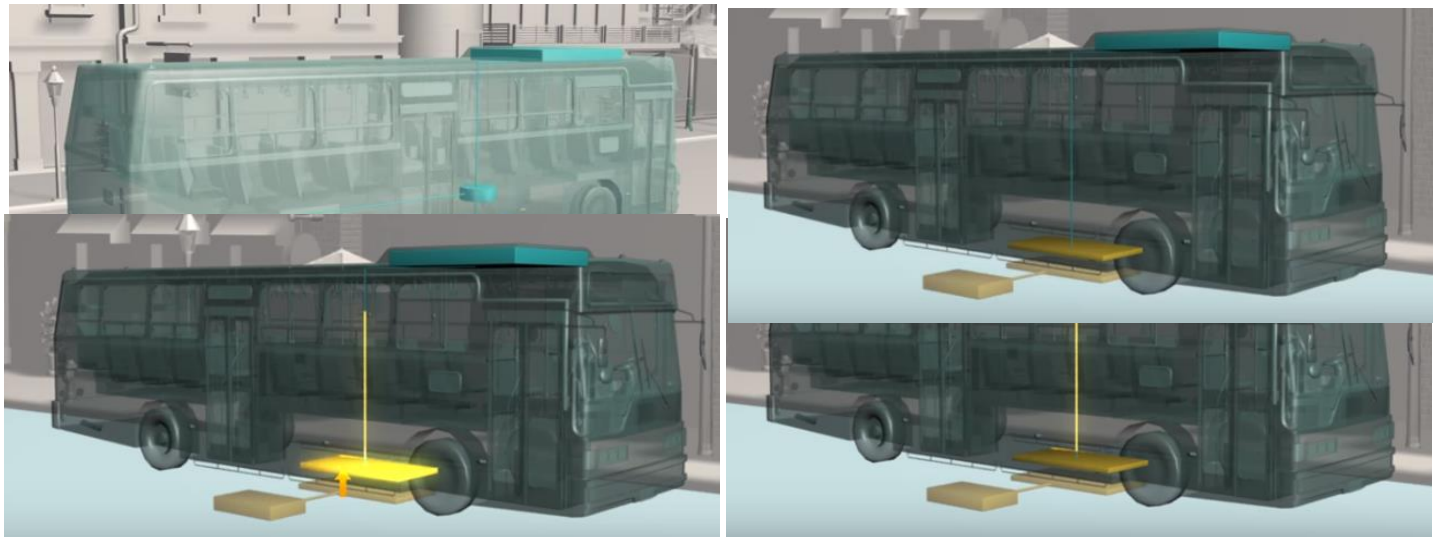
A) Recharge par induction en étant immobile : induction aux places de parking et stations d'arrêt :

L'idée n'est pas tout à fait nouvelle car on parle de la **recharge par induction** depuis plusieurs années déjà. Cette nouvelle façon de recharger les batteries (qui intéresse les constructeurs mais aussi les particuliers) permet de recharger une batterie sans câble. Une fonctionnalité pertinente, quand on sait que le système de charge est l'un des gros freins à l'achat d'une voiture électrique. Effectivement le temps de cette recharge est une partie du problème ; Aussi le manque actuel d'infrastructures.

En effet la recharge par induction nécessite des infrastructures :

Elle passe par deux bobines. La première bobine est intégrée dans une place de parking ou dans une voie de circulation. La seconde est placée sous le plancher du véhicule électrique. La recharge de la batterie est donc plus simple, mais dépend d'infrastructures particulières. Pour des bus électriques en ville par exemple c'est possible puisque les recherches menées à ce sujet ont permis de proposer cette technologie dès 2014 pour des bus électriques en Allemagne, aux Pays-Bas et en Belgique (même si cela ne résout pas le problème de l'autonomie hors agglomération).

L'opérateur régional allemand teste une technologie de charge inductive sans fil qui permet aux bus d'être rechargés par des stations d'induction souterraines. Le système est fait de composants en bordure de route installés sous la surface de la route et de composants intégrés à la structure du véhicule. La technologie permet une conception de batterie beaucoup plus petite et plus légère, tout en prolongeant la durée de vie de la batterie et en réduisant la consommation d'énergie. Les bus peuvent être rechargés lorsqu'ils s'arrêtent au moment de laisser les passagers monter et descendre.



Sur le même principe, à Saragosse, en Espagne, des scientifiques ont mis au point des bornes de recharge dissimulées sous des places de stationnement.

Le système fonctionne là aussi avec des bobines qui sont dissimulées sous le sol de la borne de recharge. La bobine est alimentée par un réseau électrique. Lorsque le véhicule est placé correctement sur la borne, une connexion se crée entre la bobine et la voiture et la bobine transfert l'énergie au véhicule. C'est un système facile à monter qui peut charger des voitures, des vans ou des bus électriques comme vu précédemment.

Les scientifiques pensent que les technologies de recharge par induction électromagnétique sont plus fiables, plus résistantes au vandalisme et plus discrètes que les bornes classiques !

En agglomération, la recharge par induction fait donc de plus en plus parler d'elle, notamment car elle permet aussi de se passer des câbles. Mais le but ultime des constructeurs reste **: pouvoir recharger tout en roulant**. Recharger en roulant permettrait en effet de ne plus avoir à attendre, de ne plus dépendre des bornes de recharge, et régler le problème d'autonomie dans les zones non équipées pour la recharge. Techniquement, la recharge en roulant n'est pas impossible.

B) Recharge par induction en mouvement : un nouveau type de route

Sur le site de Satory à Versailles la société Qualcomm fait ses tests de recharge sans fil sur l'espace réservé aux constructeurs équipementiers notamment où leurs essais automobiles sont effectués. Qualcomm a donc équipé (sur une piste d'environ 100 mètres) en son centre, de petites plaques qui dissimulent le système de recharge sans fil.

Fonctionnement de la route :

Un premier boîtier est relié au réseau électrique et à un plateau posé sous les grandes plaques et qui fonctionnent par induction à une fréquence normée (environ 85 kHz). Ce plateau envoie donc l'énergie par induction à deux autres plateaux qui se trouvent sous la voiture (un à l'avant, l'autre à l'arrière) reliés à un autre boîtier qui convertit le courant pour l'envoyer directement dans les batteries de la voiture. Il est précisé que la distance maximale entre le sol et les plaques de la voiture est de 17,5 cm mais, que cette distance pourra être augmentée pour les véhicules utilitaires.

5 tests sont alors effectués :

Une voiture roulant à 70 km/h reçoit une charge de 10 kW. Pour le deuxième test la voiture s'élance un peu moins vite, à seulement 50 km/h et reçoit cette fois-ci une charge de 15 kW. Lors du troisième test, on envoie une charge de 20kW en effectuant un changement de voie possible grâce au temps de latence inférieur à 20 millisecondes. Pour le quatrième test il est simulé un mode bouchon. On voit que la voiture reçoit 20kW. Et enfin le dernier test consiste à envoyer deux voitures qui se suivent. Les voitures reçoivent alors chacune 15kW. Il y a encore un travail sur les distances de sécurité à effectuer mais on peut déjà constater qu'il y a une adaptation du système suivant si la voiture a besoin ou non de beaucoup d'énergie pour se recharger.

Le système de recharge marche aussi bien par temps de pluie, neige ou de plaques d'huiles sur le sol car si on veut introduire ce système sur certaines autoroutes il faudra être prêt à toute éventualité.

IV. Conclusion :

L'ensemble du programme, l'ordinateur de bord et la voiture, fournissent un système de navigation qui peut (avec quelques améliorations telles que l'intégration des CNN dans les déplacements) devenir autonome.

Nous avons également vu que grâce à l'induction, la recharge d'une voiture électrique est tout à fait envisageable, aussi bien immobile qu'en mouvement. Aujourd'hui nous n'en sommes qu'à la phase de test, mais cette méthode paraît très intéressante pour répondre au problème d'autonomie des voitures électriques.

Nous pouvons tout à fait remarquer la vitesse des progrès réalisés dans ce domaine. On peut prendre comme exemple TESLA avec leur nouvelle voiture model 3 qui reçoit des mises à jour. En effet la voiture possède une dizaine de caméras de bord et des capteurs, surveillant la route en temps réel et fournissant un système d'autopilote assez avancé. La voiture reçoit des mises à jour et de nouvelles fonctionnalités très fréquemment. Verrons-nous bientôt, la mise à jour permettant la reconnaissance des panneaux, la route et l'environnement. Cela nous mènerait vers le monde d'autopilote totalement autonome.