Rapport technique ProTIS

PICHARD Grégoire – ACAR Ozan – BOISSIERE Wiliam – DUVERGER Romain
Institut d'Optique Graduate School

7 mai 2020

Nous attestons que ce travail est authentique et que toute source extérieure sera citée.

Table des matières

Introduction

Le projet que nous avons réalisé s'intitule « Apprentissage de la programmation ». Il a pour but d'initier les gens à la programmation à travers la résolution d'un labyrinthe analogique-numérique. Nous avons donc séparé les deux parties de ce projet. La partie numérique sert d'interface utilisateur, où celui-ci peut programmer ses déplacements, et la partie analogique sert à résoudre le labyrinthe en temps réel à l'aide d'une platine amovible selon 2 axes.

Il était attendu aux termes de ce projet que l'utilisateur puisse résoudre le labyrinthe, et que le pointeur situé sur la platine atteigne chaque position en moins de 20s avec une précision correcte.

Les problématiques du projet ont donc été de développer les labyrinthes, de contrôler la platine de déplacement selon les spécifications et d'assembler les parties analogique et numérique.

Pour répondre à cette problématique, nous allons dans un premier temps développer le choix du matériel utilisé, puis nous développerons les parties analogique et numérique, et nous conclurons.

1 Choix du matériel utilisé

L'une des problématiques du projet a été de faire communiquer les parties analogique et numérique. Pour répondre à ce problème, nous avons une carte de prototypage Nucléo, qui nous permet, par le biais d'un protocole de communication RS232, de relier parties analogique et numérique. Nous avons donc choisi le logiciel de programmation en conséquence.

Le choix s'est porté sur le logiciel Python pour plusieurs raisons. Premièrement les connaissances de ce langage de programmation étaient déjà acquises. Deuxièmement le logiciel possède une bibliothèque qui s'intitule Pyserial, pour laquelle la communication entre l'ordinateur est un périphérique externe est très simple. Cette communication se fait par le biais d'un protocole RS232, ce qui est bien adapté pour la carte Nucléo. Enfin, le logiciel Python permet de développer des interfaces graphiques développées assez facilement grâce à la bibliothèque Pygame.

Concernant l'application en elle-même, le schéma suivant illustre son mode de fonctionnement :

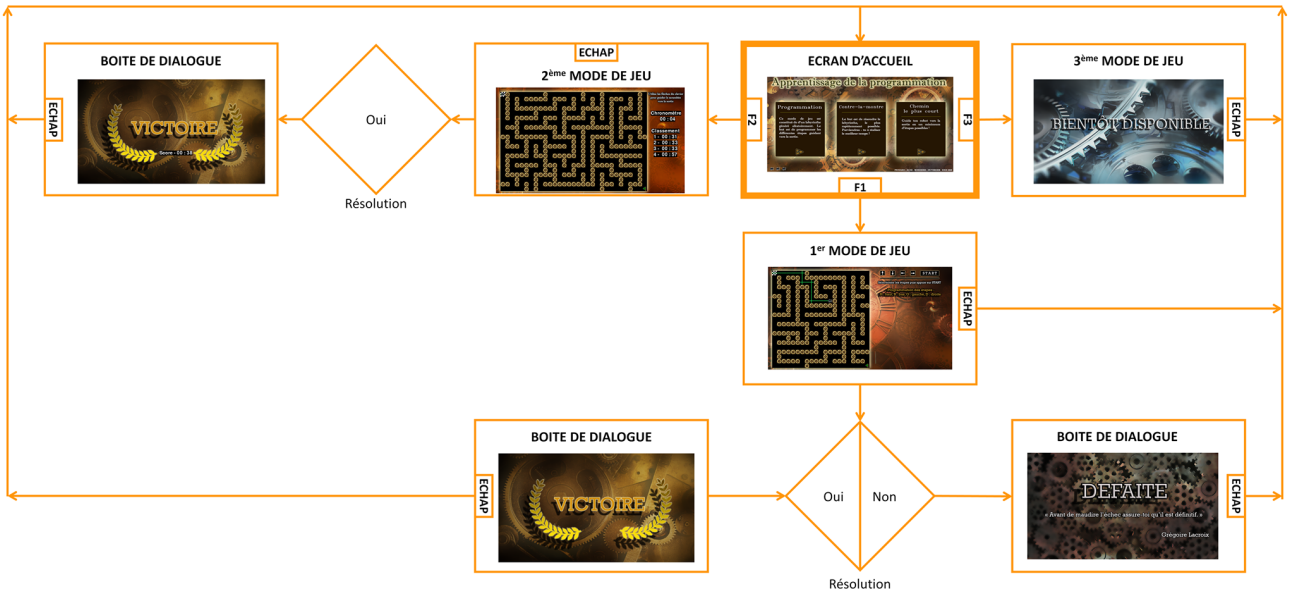

FIGURE 1 – Synoptique de l'interface utilisateur

L'objectif de cette application est que l'utilisateur ne se lasse pas du jeu au bout de 5 minutes. Pour cela, 3 modes de jeu ont été développés, comme le montre la figure ci-dessus. Le premier

mode de jeu consiste en une résolution de labyrinthe aléatoire. Si ce mode de jeu est sélectionné, un labyrinthe de 21x21 cases est généré aléatoirement. L'utilisateur devra donc programmer tous ses mouvements à l'aide de l'interface graphique. Une fois tout le programme lancé, le personnage effectue les déplacements programmés. Deux options s'offrent ensuite au personnage : soit l'utilisateur a correctement implémenté les déplacements, soit le personnage n'atteint pas la sortie du labyrinthe car l'utilisateur a mal implémenté les déplacements. Dans les deux cas, une boite de dialogue apparait une fois que le personnage a fini de se déplacer. S'affiche alors « Victoire » s'il est parvenu à sortir du labyrinthe, ou « Défaite » suivi d'une citation (générée aléatoirement parmis cinq choix) s'il n'y est pas parvenu. On revient ensuite au menu principal en cliquant sur Echap.

Le deuxième mode de jeu est un contre la montre. Dans ce mode de jeu, un seul labyrinthe est généré, et le but est alors d'en sortir le plus vite possible. Une fois le labyrinthe terminé, le temps est affiché dans une fenêtre de dialogue, et on revient au menu principal en cliquant sur Echap. A noter que les 5 meilleurs temps sont affichés sur le menu du deuxième mode de jeu.

Le troisième mode de jeu doit consister en la résolution d'un labyrinthe par le chemin le plus court. Malheureusement, par manque de temps ce mode de n'a pas pu être développé. C'est la raison pour laquelle si on le sélectionne on ouvre une boite de dialogue avec écrit « Bientôt disponible ». On pourra ensuite revenir au menu principal en cliquant sur Echap.

Intéressons-nous maintenant à la partie analogique. Le choix du labyrinthe analogique s'est porté sur une platine de déplacement, car cet outil était disponible avant le début du projet. Une photo de la platine est située ci-dessous :

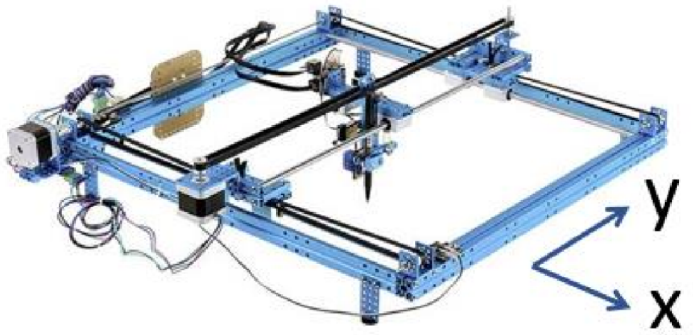

FIGURE 2 – Illustration de la platine de déplacement. Les deux bras de la platine peuvent se déplacer selon les axes x et y

Cette platine est munie de deux moteurs pas à pas. C'est donc une configuration idéale pour contrôler avec précision le pointeur de cette platine.

2 Fonctionnement de la partie logicielle

Comme expliqué précédemment, l'interface utilisateur va être constituée de trois modes de jeu : un mode axé sur la programmation de chacune des étapes pour accéder à la sortie du labyrinthe, un mode contre-la-montre et un dernier niveau qui n'a pas encore été finalisé.

Nous avons décidé de réaliser l'interface graphique à l'aide de python en utilisant la bibliothèque pygame. Cette dernière permet notamment d'associer des actions aux clics sur les différents boutons de la sourie ou aux différentes touches de l'ordinateur. Dans les parties suivantes nous allons décrire les différentes fonctions du codes qui nous a permis d'implémenter l'interface :

2.1 Gestion générale

Nous avons partagé notre codes en quatre scripts interconnectés :

Le script intitulé "constante.py" contient les variables globales du programme. On y retrouve le chemin vers le dossier de l'interface (ce qui va permettre de transmettre le code très facilement sur un autre ordinateur) et surtout le chemin des différentes images constituant l'interface. Ainsi si l'on souhaite utiliser une autre image on aura juste à changer le chemin dans ce script (une seule fois).

Deux autres scripts, "classes.py" et "generer.py" contiennent des fonctions qui seront utiles pour le code principal ("labyrinth.py"). Leur utilité est donc surtout de donner de la clarté au code principal.

Pour interconnecter ces différents scripts, on les importe depuis le script principal ("labyrinth.py") en utilisant la commande suivante :

```
from classes import *
from constantes import *
from generer import*
```

Pour créer l'interface sous pygame, on utilise un système de boucle infini avec la commande **while** (comme en TP d'électronique). On a une boucle principale qui fait tourner le jeu, associée dans le code à la variable **continuer** qui a une valeur unitaire temps que l'on ne quitte pas le jeu. On quitte le jeu si on appuie sur la croix en haut à droite (c'est l'évènement **QUIT**). Dans le code, cela correspond à un condition **if** qui met alors la variable **continuer** à 0 :

```
for event in pygame.event.get():
    # si l'utilisateur quitte, on met la variable qui continue le jeu
    # a 0 pour fermer la fenetre
    if event.type == QUIT :
        continuer = 0
```

Le but des évènements (clics de souris ou appuis sur les touches de clavier) est de naviguer entre ces différentes boucles **while** (par des conditions). On prévoit donc une boucle **while** pour l'accueil, pour les modes de jeu, une pour l'affichage de la defaite et une l'affichage de la victoire. La navigation entre les différents modes de jeu se fera par des conditions à l'intérieur de la boucle jeu.

Une fois que l'on a compris la gestion des évènements, il faut comprendre la façon dont pygame gère les images. Le simple exemple suivant permet de comprendre le fonctionnement : si on veut déplacer un personnage sur un fond (background) du coin supérieur droit au coin inférieur gauche, on procède de la façon suivante :

- On colle le fond
- On colle par dessus le personnage sur le coin supérieur droite
- On recolle le fond par dessus (donc on ne voit que le fond)
- On colle le personnage sur le coin inférieur gauche

C'est de cette manière que l'on va simuler le déplacement du robot dans le labyrinthe.

La commande pour afficher une image à la coordonnée (**x**, **y**) avec pour origine le coin supérieur gauche de l'image et **x**, **y** positifs est :

```
accueil = pygame.image.load(image_accueil).convert() # on charge l'image
fenetre.blit(accueil, (x,y)) # on l affiche
```

(ici **image_accueil** est la variable de "constante.py" contenant le chemin de l'image à charger)

Pour générer des images aléatoires (c'est la cas par exemple pour le mode 1 ou l'image de la défaite contient une citation générée aléatoirement), on crée un nombre aléatoire à l'aide de la bibliothèque **random**. On tranforme ce nombre en chaine de caractère et on le concatène à un chemin de base. Les images sont par exemple "image_1", "image_2", "image_3" et le chemin de base sera en ".../image_". On génère ici un nombre entre 1 et 3. Dans le code :

```
defaite = pygame.image.load(fond_echec+str(alea)+'.png').convert()
# str(alea) correpond au nombre aleatoire change en chaine de caractere
# + permet la concatenation
# avec alea = radom.randint(1,5) dans le code
fenetre.blit(defaite, (0,0)) # affichage
```

2.2 Utilisation de classes

Dans le script "classes.py", on utilise des classes permettant de gérer l'affichage du labyrinthe dans les deux modes (programmation et contre-la-montre) et de gérer les déplacements du personnage (un scarabée).

La classe **Niveau** contient plusieurs attributs :
- un attribut permettant de générer une matrice contenant le labyrinthe par lecture d'un fichier .txt. Ce fichier .txt contient des **0** indiquant les cases libres et des **m** indiquants les murs :

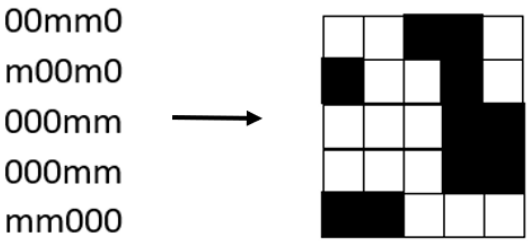

FIGURE 3 – Fichier .txt à gauche et labyrinthe correspondanat à droite (case noire = mur)

- un attribut qui permet d'afficher le labyrinthe en chargeant une image du mur et en la collant sur le fond pour chaque case **m** de la matrice précédemment générée (par calcul des coordonnées).

La classe **Perso** qui dépend notamment de la dimension du labyrinthe. En effet, l'attribut **déplacer** permet de calculer les coordonnées de déplacements dans le labyrinthe en fonction de la direction choisie et de la structure du niveau (on ne peut pas se déplacer sur un mur). La direction importe pour le calcul des coordonnées (**self.x** et **self.y**) mais aussi pour afficher la bonne image du personnage (stockée dans **self.direction**). Il y a en effet quatre images possibles :

FIGURE 4 – Quatre sens donc quatre images du personnage

Cette classe est très importante notamment pour tracer le chemin du scrabée à chaque instant. Avec ce système de collage (expliqué en haut), il est difficile de tracer le chemin du scarabée car à chaque instant il faut tracer le bon chemin du départ jusqu'au scarabée. Pour cela, la solution est d'utiliser des listes pour retenir à la fois la direction (2 directions de chemin possibles : vertical ou horizontal) et les coordonnée des cases déja visitées. Ce qui devient complexe est alors de "coller" les images au bon moment dans le code principal "labyrinth.py".

FIGURE 5 – Deux directions de chemin possibles : horizontal ou vertical

2.3 Fonctions implémentées

2.3.1 Génération aléatoire d'un labyrinthe

Le dernier script, "generer.py" contient quelques fonctions utiles pour se simplifier la tâche dans la programmation :

Il permet dans un premier temps de générer un labyrinthe aléatoire au format .txt (comme expliqué précédemment) de la taille souhaitée. Nous avons exploité l'algorithme de Wikipedia sur l'exploration exhaustive (article : Modélisation mathématique de labyrinthe) :

Exploration exhaustive [modifier | modifier le code]

On part d'un labyrinthe où tous les murs sont fermés. Chaque cellule contient une variable booléenne qui indique si la cellule a déjà été visitée ou non (i.e. les cellules visitées sont celles qui appartiennent au chemin du labyrinthe en cours de construction).

Au départ, toutes les cellules sont marquées comme non visitées (*faux*).

On choisit arbitrairement une cellule, on stocke la position en cours et on la marque comme visitée (*vrai*).

Puis on regarde quelles sont les cellules voisines possibles et non visitées.

S'il y a au moins une possibilité, on en choisit une au hasard, on ouvre le mur et on recommence avec la nouvelle cellule.

S'il n'y en pas, on revient à la case précédente et on recommence.

Lorsque l'on est revenu à la case de départ et qu'il n'y a plus de possibilités, le labyrinthe est terminé.

FIGURE 6 – Génération d'un labyrinthe par exploration exhaustive

Voici comment nous avons adapté le problème : On part tout d'abord d'une grille vide. Pour définir une grille sous python on considère qu'une case est un **0** entouré de quatres murs (des **1**). De manière générale les **0** seront les indices paires selon les lignes et les colonnes (indexation commence à zéro en python) :

```
>>> M # exemple d'une grille 5 * 5
array([[0, 1, 0, 1, 0],
       [1, 1, 1, 1, 1],
       [0, 1, 0, 1, 0],
       [1, 1, 1, 1, 1],
       [0, 1, 0, 1, 0]])
```

Cette matrice initiale est dupliquée (matrice **M2**) dans le code pour avoir une matrice qui retient si une case a été visitée ou non (dans le code, on a choisit de remplacer le **0** d'une case par un **3** si la case a été visitée).

Pour appliquer l'algorithme de Wikipedia, on a créé une sous-fonction **coord_voisin** qui permet de connaître les indices des cases voisines d'une case choisie. La sous-fonction **voisins_pos** permet de vérifier pour une liste de cases voisines celles qui peuvent encore être visitées, c'est à dire celles qui n'ont pas encore été visitées. Ces deux fonctions retourne donc une liste de coordonnées de cases.

Casser un mur dans l'algorithme de Wikipedia revient donc ici à remplacer un **1** par un **0**. Pour choisir un voisin aléatoirement pour une case donnée on utilise la fonction **random.randint** sur le nombre de voisins possibles. Un fois que l'on est revenu à notre case de départ (elle-même choisie

aléatoirement), la matrice **M** contient le labyrinthe à tracer. On l'écrit alors dans un fichier.txt en remplaçant les **1** par des **m**. Ce fichier sera lu par la classe **Niveau** comme expliqué précédemment.

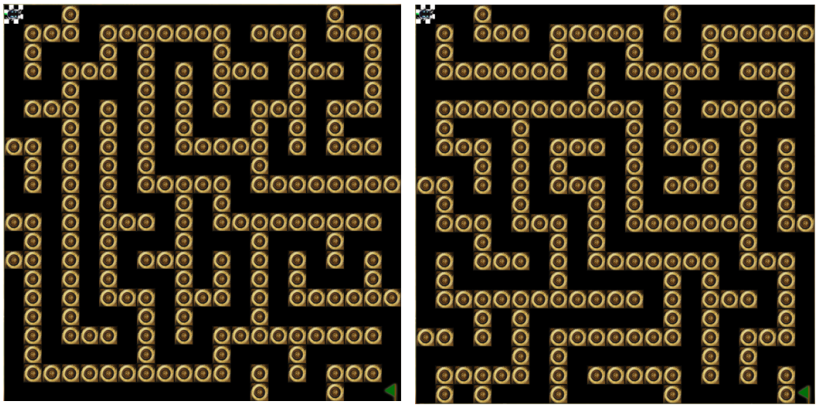

FIGURE 7 – Exemple de deux labyrinthes générés aléatoiremement

On note qu'avec ce type d'algorithme, il n'y a qu'un chemin possible pour parvenir à la sortie. Par conséquent il faudrait trouver un autre algorithme avec plusieurs chemins possibles pour le troisième mode (pour que la notion de chemin le plus court est un sens).

2.3.2 Gestionnaire du classement des meilleurs temps dans le mode contre-la-montre

Pour gérer le temps dans le mode contre-la-montre on utilise la ligne de code suivante :

```
temps = round(pygame.time.get_ticks()/1000-tps0,0) # temps en ms donc /1000 on arrondit 0 chiffre apres la virgule
```

Cette derniere permet de récupérer le temps en seconde et nous avons créé une fonction dans "générer.py", **minute** qui affiche le temps dans le format (minutes : secondes). Cette fonction fournit donc en sortie une chaîne de caractère. Dans "labyrnthe.py", les lignes de commandes suivantes permettent d'afficher du texte :

```
font = pygame.font.Font('freesansbold.ttf', 32)
text = font.render("chaine de caractere", True, white, black)
textRect = text.get_rect()
fenetre.blit(text, (900,700)) # position sur le fond
```

C'est très utile pour afficher les chronomètres mais aussi les commandes de l'utilisateur dans le premier mode de jeu.

Les différents chronomètres sont en fait écrits dans un fichier "score.txt" puis lus et affichés avec les lignes de commandes que nous venons de rappeler. Dans "labyrinth.py" nous avons posé une condition pour que ce fichier ne contiennent pas plus de cinq lignes (chaque ligne correspond à un chronomètre). Si ça devient le cas, alors on procède de cette façon sur python :
- on ouvre le fichier "score.txt" en lecture
- on récupère toutes les lignes
- on ouvre le fichier en écriture puis on le ferme (ce qui a pour conséquence de l'effacer)
- on ouvre le fichier en écriture et on écrit les trois premières lignes que l'on avait récupérées

Pour que ces trois lignes correspondent aux meilleurs temps, on a écrit une fonction **tri** dans "générer.py" qui trie au préalable le fichier "score.txt".

3 Fonctionnement de la partie matérielle du prototype

L'objectif du projet était de construire un dispositif d'apprentissage ludique de la programmation, incluant un robot dont le déplacement au sein d'un labyrinthe est programmable à l'aide d'une application. La partie logicielle décrite dans le paragraphe n'est donc pas suffisante, il fut également nécessaire d'envisager la construction d'une partie matérielle capable d'interagir avec la partie logicielle, de recevoir des instructions de la part de cette dernière, et d'agir conformément à ces instructions.

Il est important de préciser que la solution présentée dans ce paragraphe n'est qu'une ébauche qui n'a pu être ni réalisée en pratique, ni testée, en raison de la situation de confinement en vigueur au moment du projet, et de l'absence de matériel et d'appareil de mesures qui en a résulté. La seule chose qu'il fut possible de faire fut d'imaginer comment il aurait été possible de construire un petit robot capable de se déplacer en fonction des instructions de l'utilisateur, quels composants et outils auraient pu être utilisés pour cela, comment ces composants auraient pu être reliés et/ou programmés, etc... C'est pourquoi nous ne décrirons ici que la constitution et le principe de fonctionnement de la solution envisagée. Etant donné qu'aucun essai n'a pu être réalisé, il est impossible de garantir le bon fonctionnement de la solution proposée, ni l'exactitude des câblages. Cependant, il est pertinent d'expliquer quels tests auraient pu être réalisés pour s'assurer du bon fonctionnement des différents éléments de la partie matérielle. Les programmes et schéma de câblage des protocoles de test n'ont pas été joints, car ils n'ont pas pu être mis en application, et nous ignorons s'ils sont réellement fonctionnels.

La solution qui nous a parue la plus simple et efficace pour réaliser le déplacement en temps réel du robot dans le labyrinthe fut d'utiliser la platine de translation XY Plotter de la marque Makeblock, platine déjà disponible au LENSE. Le choix d'un tel élément peut être justifié par plusieurs raisons. D'une part, cette platine normalement destinée au dessin intègre un « plotter » dont la position est commandée par deux moteurs pas à pas, l'un permettant une translation horizontale (axe X), et l'autre suivant la verticale (axe Y). La surface de la platine peut donc faire office de labyrinthe, le plotter peut être vu comme le robot à guider, et le moteurs peuvent être utilisés pour faire suivre au robot la trajectoire souhaitée par l'utilisateur. De plus, utiliser un élément déjà presque « tout prêt » tel que la platine évite de devoir concevoir un robot et un labyrinthe soi-même de A à Z. Cela permet donc de gagner du temps compte tenu de la durée limitée de notre projet, et des difficultés liées au confinement. Une question se pose alors : comment faire communiquer la partie logicielle et la partie matérielle de façon que le robot se positionne là où l'utilisateur le souhaite ? D'une part, les deux moteurs de la platine peuvent être aisément contrôlés à l'aide d'un microcontrôleur capable de recevoir et de transmettre des instructions de déplacement, tel qu'une carte Nucléo par exemple. Pour contrôler les moteurs, ce microcontrôleurs devra être couplé à un driver (tel qu'un L293D ou bien un L298). De plus, il est possible de transmettre les instruction définies dans l'interface utilisateur au microcontrôleur pilotant les moteurs par l'intermédiaire d'une liaison série RS232. Ce type de liaison est pertinent dans la mesure où elle peut s'intégrer très bien au code de la partie logicielle, car il existe une bibliothèque Python (Pyserial) permettant de gérer une liaison RS232.

Le fonctionnement de la partie matérielle du prototype est représenté dans le schéma bloc ci-dessus. Il peut être résumé comme suit. Dans un premier temps, l'utilisateur spécifie grâce à l'interface utilisateur la nouvelle case du labyrinthe où il souhaite positionner le robot. Grâce à la donnée de la case où était positionnée le robot à l'instant d'avant, et à celle de la nouvelle case où le robot doit être positionné, une partie du programme Python peut alors déterminer le sens, la direction et la longueur du déplacement nécessaire dans le système de coordonnées XY commun à la platine et au labyrinthe. Après que le déplacement à réaliser ait été correctement défini, la même partie du code Python se charge de déterminer quel moteur doit agir (celui contrôlant la translation suivant X, ou celui contrôlant la translation suivant Y), ainsi que le sens et la valeur (en nombre de pas) de la

rotation que ce moteur doit effectuer pour positionner le robot à la case souhaitée. Cela suppose bien entendu de connaître la correspondance entre l'angle de rotation du moteur en nombre de pas et la longueur de translation parcourue.

C'est ensuite qu'intervient une seconde partie du code Python, écrite à l'aide des fonctions de la bibliothèque Pyserial, et dédiée à la gestion de la communication RS232 entre la partie logicielle et la partie matérielle du prototype. Cette partie du code prend en entrée l'identifiant du moteur sur lequel il faut agir, le sens de rotation, et la valeur de la rotation, et envoie ces 3 données à la carte Nucléo pilotant les moteurs. Les 3 données envoyées transitent par un câble USB reliant la carte Nucléo au PC sur lequel est installée l'interface utilisateur. Une fois que ces instructions ont été réceptionnées par la carte Nucléo, celle-ci se charge d'activer les bonnes entrées du driver (LM293D, LM298, etc...) contrôlant les moteurs, et ce de façon adaptée, afin que le moteur sélectionné puisse tourner du nombre de pas demandé, et ce dans le sens souhaité. En faisant ainsi tourner son axe, le moteur va mettre en mouvement le robot par l'intermédiaire de la courroie associée au moteur, et le robot pourra bien être positionné à la case définie par les instructions de l'utilisateur. En reproduisant tout ce processus pour chaque case faisant partie du parcours du robot, le robot peut alors suivre le trajet défini par l'utilisateur et sortir du labyrinthe.

Pour pallier l'impossibilité de construire la partie matérielle, il avait été envisagé au cours du projet de réaliser une simulation de son fonctionnement sous la forme d'une animation 3D, permettant de voir le robot se déplacer dans le labyrinthe dès lors que l'utilisateur ordonne un déplacement. Dans cet objectif, nous avions songé à transposer la partie matérielle sur le logiciel Simulink, car ce logiciel comporte une bibliothèque (Simscape) permettant de simuler le fonctionnement d'un moteur pas à pas. Cependant, cette possibilité a par la suite été abandonnée, pour plusieurs raisons. D'une part, la bibliothèque Simscape était incompatible avec les versions de MatLab installées sur nos PC. De plus, un programme Simulink faisant appel aux objets de la bibliothèque Simscape ne permet que de simuler le fonctionnement d'un moteur pas à pas, mais pas le déplacement du robot dans le labyrinthe qui en résulte. De ce fait, il aurait été impossible d'avoir une animation dans laquelle on voit le robot évoluer en temps réel à la suite de chaque action de l'utilisateur, et la solution Simulink perd ainsi de son intérêt. Enfin, le programme Simulink aurait été incapable de recevoir des données envoyées au format RS232 par la partie logicielle, et d'interagir avec cette dernière, car rien sous Simulink ne permet de faire cela. Dès lors, il aurait fallu reproduire « manuellement » dans Simulink tous les déplacements spécifiés par l'utilisateur, ce qui n'a aucun intérêt compte tenu que le déplacement doit s'effectuer de façon automatisée.

Voyons maintenant quels protocoles de test auraient pu être mis en œuvre pour s'assurer du bon fonctionnement de chaque élément de la partie matérielle du prototype. Les différents tests envisagés sont récapitulés dans le tableau ci-après.

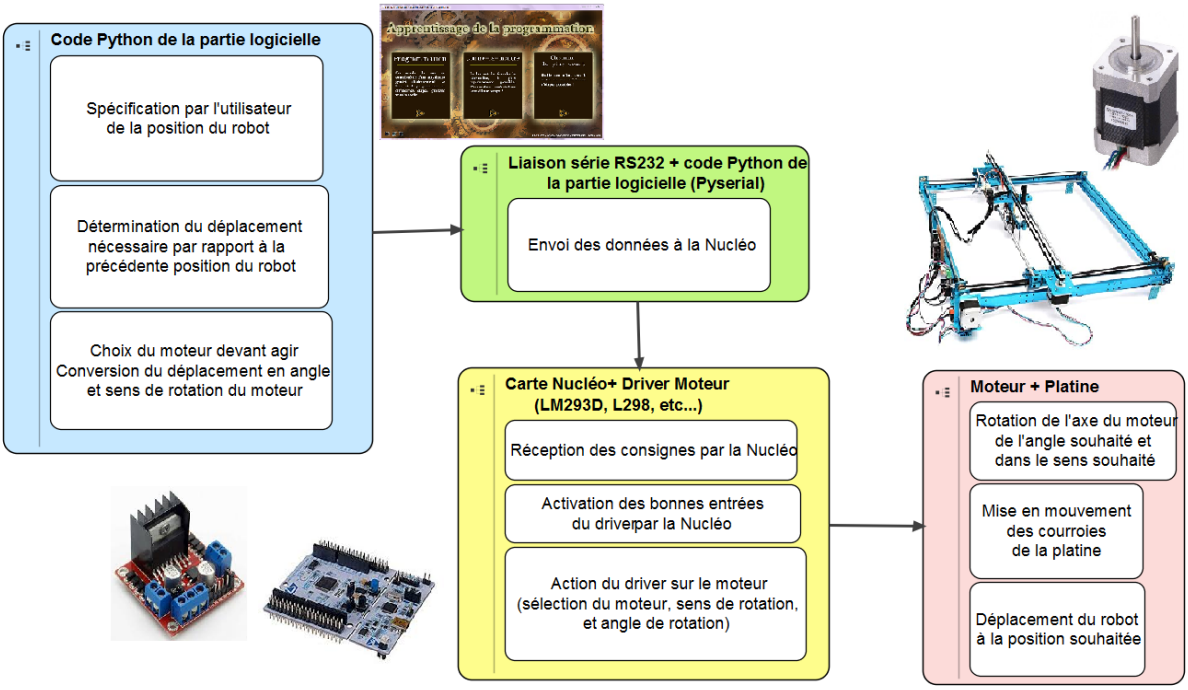

FIGURE 8 – Schéma bloc

Elément testé	Protocole de test
Liaison série RS232 entre le PC et la carte Nucléo	• Connecter la carte Arduino au PC par l'intermédiaire d'un câble USB • Ecrire à l'aide de la bibliothèque Pyserial un code Python envoyant des caractères à la carte Nucléo • Téléverser dans la carte Nucléo un code MBED ordonnant à la carte d'allumer une LED si le caractère reçu a telle valeur, ou de la laisser éteinte le cas échant
Compréhension par la carte Nucléo des données reçues concernant le sens de rotation et l'angle de rotation du moteur	• Connecter la carte Arduino au PC par l'intermédiaire d'un câble USB • Ecrire à l'aide de la bibliothèque Pyserial un code Python envoyant 3 données (moteur choisi, sens de rotation, angle de rotation en nombre de pas) définies manuellement • Téléverser dans la carte Nucléo un code MBED permettant à la carte de lire la série de 3 données reçues, et de la faire afficher dans un moniteur série tel que GTKTerm
Test des moteurs de la platine et du driver de contrôle des moteurs	• Câbler correctement la carte Nucléo, le driver et les moteurs • Téléverser dans la carte Nucléo un code MBED lui ordonnant de faire tourner le moteur d'un certain angle et dans un certain sens, définis manuellement • Vérifier que le moteur tourne bien comme attendu. Reproduire le test avec plusieurs valeurs d'angle de rotation et plusieurs sens
Totalité de la partie matérielle	• Définir manuellement le moteur choisi, ainsi qu'un angle et un sens de rotation dans la partie du code Python gérant la communication RS232 et l'envoi des données • Vérifier que le moteur agit bien comme souhaité

4 Retour d'expérience et bilan des acquis

4.1 Outils et concepts maîtrisés

Ce projet nous a permis d'aborder et de maîtriser plusieurs outils et concepts.

En programmation d'une part, nous avons dû utiliser Python et réussir à implémenter et mettre en pratique la bibliothèque Pygame. Nous avons fait face à de la programmation en vue de créer une interface graphique. Dans le passé, nous avons été peu confrontés à ce problème, l'interaction avec l'utilisateur se faisant généralement via l'invité de commande. Nous savons à présent former un rendu visuel agréable et interactif pour l'utilisateur. Toutefois, Pygame ne se limite pas à ça. Comme dans tout jeux-vidéos, nous avons appris à faire communiquer la souris avec notre interface graphique afin d'assurer la jouabilité. Nous avons également appris à ordonner et organiser notre code sous l'aspect modulaire souvent abordé pendant les cours de C. Pour une fois, nous avons bien compris en quoi cet aspect modulaire est très essentiel quand notre code est long et dense. Cette modularité nous a également permis d'introduire le concept de classes en programmation, que nous avions peu utilisé auparavant.

D'autre part, nous avons dû nous creuser la tête pour réussir à travailler en télé-travail sur la partie analogique. En effet, n'ayant pas accès au matériel, nous avons en grande partie étudié la partie théorique derrière l'aspect analogique. Ainsi, nous avons étudié la communication série RS232 qui permet de relier une carte nucléo et le langage Python. Cela nous permet alors d'utiliser et de contrôler en tension les différents moteurs sur la platine deux axes. Le gros du travail était de réussir à concevoir un schéma électrique clair de ce que nous pourrions faire si nous avions le matériel et nous avons utilisé pour cela le logiciel Fritzing. Nous avons également essayé d'apprendre à utiliser la bibliothèque Simulink disponible sous Matlab, que nous avons eu tout dabord beaucoup de mal à utiliser et à installer. Notre idée était de simuler la platine pour commencer à appréhender les différentes contraintes mécaniques. Néanmoins, nous avons très peu d'expériences dans ce genre de simulation et l'idée a donc été vite abandonnée. Cette partie nous a fait perdre pas mal de temps, mais nous gardons tout de même en temps quelques notions sur Simulink, peut-être que nous aurons à simuler de la mécanique plus tard.

Enfin, nous avons mis en œuvre lors de ce projet notre capacité à nous organiser en équipe, à répartir les tâches entre nous et dans le temps. Cette répartition a été essentielle pour avancer puisque notre projet est divisé en deux grandes parties : analogique et numérique. D'autre part, nous avons à présent l'expérience du travail à distance et nous savons utiliser différents outils permettant de travailler ainsi. Nous avons appris à utiliser un outil de partage de fichier et d'organisation de projet tel que Blackboard. Pour communiquer entre nous, nous avons en grande partie utilisé la plateforme ecampus et messenger. Le travail en télé-travail n'est pas toujours très simple, outre les difficultés rencontrés parfois en terme de connexion, la communication est moins claire et précise qu'en face à face, la productivité de chacun en est amoindri.

4.2 Bibliographie des documents consultés

— Bien évidemment le site du Lense avec ses nombreux tutos pour la partie analogique (commande de moteur, utilisation de la carte nucléo..)
— Openclassrooms pour apprendre à utiliser Pygame