Offre de projet - OFFRE 2021_B002

Robot Véronica

en route pour Mars

Sommaire

Introduction **6**
Description du projet 6
Cahier des charges 6
Contraintes générales 6
Performances attendues 6
Déplacement 6
Mesure 6
Autonomie 6
Ergonomie 6
Architecture système et composants **7**
Architecture du système 7
Composants 7
Carte d’acquisition 7
Émetteur et récepteur 8
Moteur 8
Pont en H 8
Batterie 9
Convertisseur de tension 9
Capteur de température 9
Capteur de pression 9
Architecture de communication **10**
Objectifs de la communication 10
Protocole de communication 10
Communication de l’opérateur au robot 10
Communication du robot à l’opérateur 10
Mise en place 10
Architecture électronique 10
Architecture logicielle 11
Tests 12
Architecture du pilotage **13**
Mise en oeuvre 13
Généralités 13
Précautions à mettre en place 13
Protocole de communication 13
Architecture logicielle 14
Généralités 14
Entrées et sorties utilisées 14

Entrées 14
Sorties 14
Réception des données 14
Contrôle des données de consignes 14
Généralités 14
Test 1 : taille du message 15
Test 2 : contrôle du premier caractère de chaque consigne 15
Test 3 : contrôle du deuxième caractère de chaque consigne 15
Test 4 : contrôle des données de déplacement de chaque consigne 16
Test 5 : contrôle des valeurs de vérification 16
Réécriture des données 16
Déplacement du robot 16
Généralités 16
Fonctionnement de la sous-fonction 17
Asservissement en vitesse 18

Réalisation d'une interface 19
Description 19
Architecture de l'interface 19
Modes de commande 20
Protocole de communication 21
Architecture logicielle 21

Performances énergétiques 22
Nécessité 22
Protocole 22
Résultats 22
Axes d'amélioration 22

Conclusion 23

Annexes 24
Annexe 1 : Structure principale de la fonction de déplacement 24
Annexe 2 : Structure de la sous-fonction Reception() 26
Annexe 3 : Structure de la sous-fonction Verification() 27
Annexe 4 : Structure de la sous-fonction Ecriture() 29
Annexe 5 : Structure de la sous-fonction Deplacement() 30
Annexe 6 : Structure de la sous-fonction Asservissement() 33
Annexe 7 : Schéma de la structure en plexiglas 34
Annexe 8 : Programmes d'émission et réception 35
Annexe 9 : Code de l'Interface Homme Machine 36

1. Introduction

1.1. Description du projet

Cette documentation porte sur la réalisation d'un robot, guidé à distance, permettant l'exploration au sol de Mars pour détecter la présence d'eau. Baptisé Veronica, le robot se déplacera par tronçons et collectera des données environnementales qui seront transmises sur Terre. Le projet comporte également la réalisation d'un interface logiciel.

1.2. Cahier des charges

1.2.1. Contraintes générales

Lors de son parcours, le robot devra être en mesure de se déplacer en ligne droite et d'effectuer des rotations sur son axe en respectant des consignes envoyées depuis la Terre, via une interface graphique. De plus, il effectuera à intervalle régulier des relevés environnementaux, tels que des mesures de pression atmosphérique et de température. Ces données seront ensuite transmises à un centre d'analyse de façon périodique.

1.2.2. Performances attendues

1.2.2.1. Déplacement

Les performances attendues sur le déplacement impliquent une erreur maximale de **2 cm** sur la position du robot et une erreur maximale de **3°** sur son angle. Veronica devra également atteindre une vitesse comprise entre **10 cm/s** et **30 cm/s**.

1.2.2.2. Mesure

Les mesures devront être réalisées tous les **10 cm** et envoyées au centre toutes les **10 minutes**.

1.2.2.3. Autonomie

Le robot devra pouvoir effectuer un parcours de **1 km** avant de procéder à toute recharge.

1.2.2.4. Ergonomie

L'interface graphique à développer devra pouvoir être utilisée sans formation préalable et affichera les données recueillies en fonction de la distance ou du temps.

2. Architecture système et composants

2.1. Architecture du système

Le système se compose du robot à proprement parler et du poste d'émission.

Le robot est constitué d'une plaque en plexiglas renforcé par une armature métallique. Le contact avec le sol se fait par 2 roues motrices directement montés sur l'axe de leur propre moteur sur le dessus de la plaque et d'une roue folle placé à l'avant du robot sous la plaque. La batterie est fixée au centre du robot, sous la plaque. Tous les autres composants sont fixés sur la plaque. Le montage intégral est effectué de tel manière que le centre de gravité du robot se situe au plus près du centre du robot, afin d'en améliorer l'équilibre.

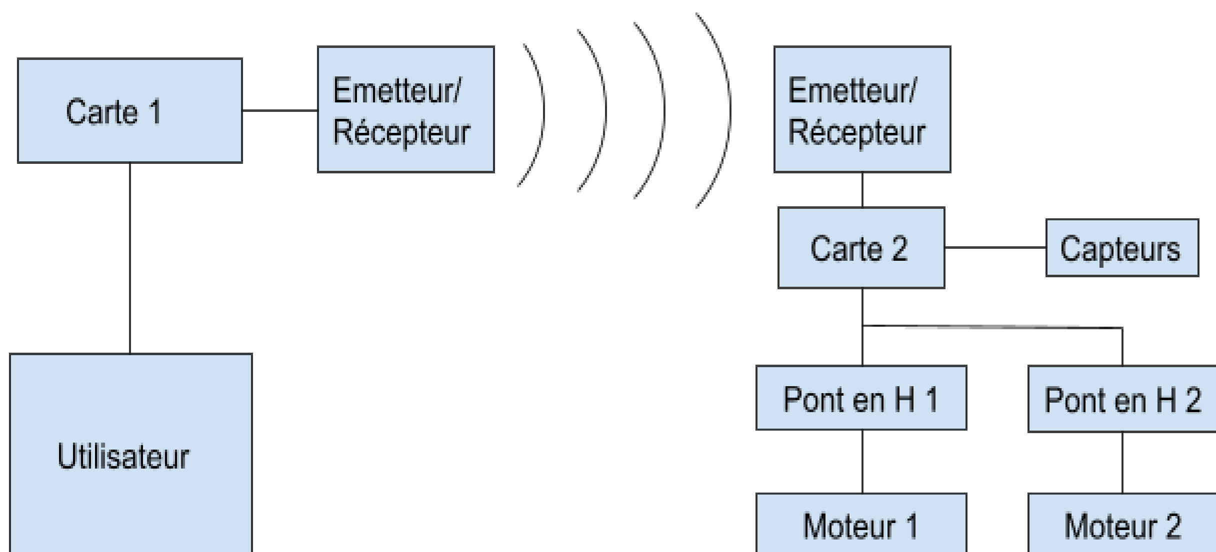


Figure 1 : Schéma simplifié de la structure du robot et de sa commande

Chaque roue est commandée séparément. Le fait de commander chaque moteur séparément permet d'avoir plus de degrés de liberté notamment lors des virages ou chaque roue doit tourner à une vitesse propre. On communique avec le robot à l'aide d'un système de communication radio onde à distance. Les ponts en H permettent d'adapter la tension d'alimentation à celle de la carte qui est de 5V. Enfin, l'utilisateur est censé piloter le tout à l'aide d'une interface homme-machine.

2.2. Composants

2.2.1. Carte d'acquisition

La carte d'acquisition utilisée est une carte Nucléo L476RG, équipée d'une connectique USB et pilotée via des fonctions réalisées sous MBED Compiler. La carte peut fonctionner via une alimentation externe, type batterie, ou être alimenté par son cordon USB,

les tension d'alimentation tolérées suivant le type d'alimentation choisi sont : 3.3 V, 5 V et de 7 à 12 V.

2.2.2. Émetteur et récepteur

La communication à distance est assurée par une paire de puces Kappa-M868 dont le schéma est donné au paragraphe 3.3.1. Ces deux puces assurent chacune l'émission et la réception. Elles sont alimentées par une tension continue 3.3 V.

2.2.3. Moteur

Le robot est équipé de deux moteurs à courant continu, tous munis d'un motoréducteur. Sur chaque moteur on dispose de deux encodeurs qui permettent de connaître la position de l'axe principal du moteur.
Les caractéristiques principales du moteur sont la tension de fonctionnement standard (12 V) et le rapport de réduction (19.225).

2.2.4. Pont en H

Le pont en H, vendu par Digilent et de référence PmodHB5, permet de réguler la tension appliquée au moteur. Il permet de transmettre les informations du codeur et de piloter le moteur en envoyant le sens de rotation et la puissance à appliquer. La tension d'alimentation du moteur est appliquée séparément de la régulation.

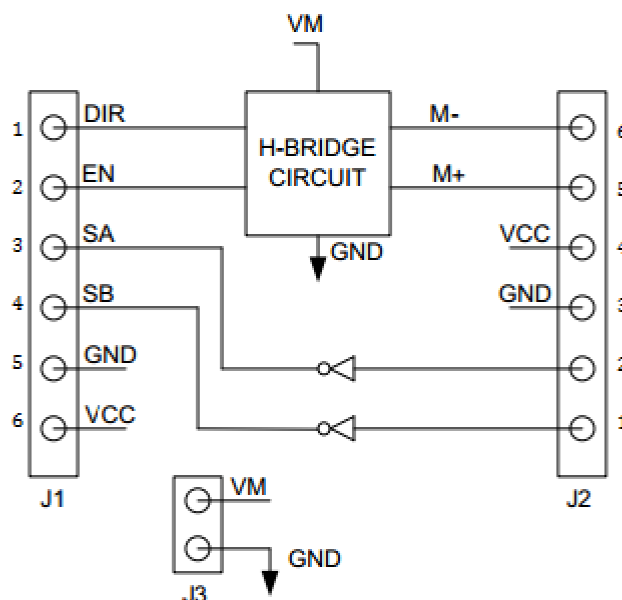


Figure 2 : Diagramme du pont en H. Le moteur est branché du coté J2, les commandes sont envoyées par J1 et l'alimentation du moteur se fait par J3.

2.2.5. Batterie

On utilise des batteries classiques, délivrant une tension d'alimentation comprise en 6.5 V et 8 V suivant l'état de charge. La capacité des batteries est de 4000 mAh.

2.2.6. Convertisseur de tension

Ce composant, de référence L7805CV et conçu par STMicroelectronics, permet de convertir la tension envoyée de la batterie en une tension de 5V et un courant pouvant aller jusqu'à 1,5A, servant à alimenter les ponts en H, l'émetteur récepteur et les capteurs de température.

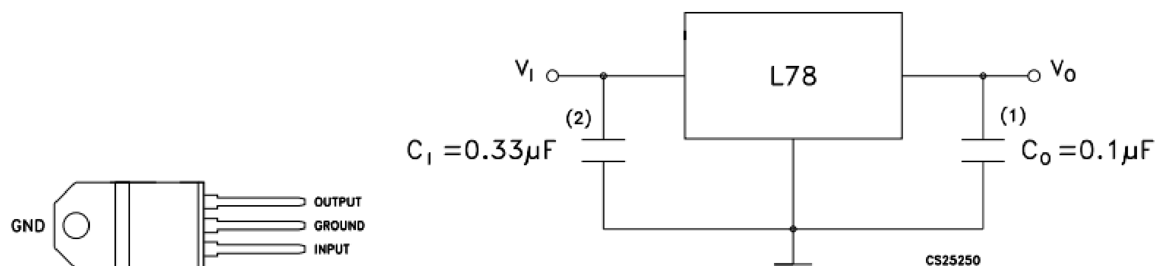


Figure 3 : Bornes du composant et circuit électrique utilisé. Source : Documentation constructeur. (1) : Bien que ce condensateur n'est pas nécessaire pour la stabilité, il améliore la réponse transitoire. (2) : Nécessaire si le régulateur est placé à une distance appréciable du filtre de l'alimentation

2.2.7. Capteur de température

Ce capteur de température, de référence LM35 et vendu par Texas Instruments, permet d'obtenir une tension image de la température, comprise entre 0V pour une température de -40°C et allant jusqu'à 110°C avec une sensibilité de 10 mV/°C. La précision de mesure est de ±1°C. Il est alimenté en 5V.

2.2.8. Capteur de pression

Le capteur de pression, de référence MPX5100AP, permet d'obtenir une tension image de la pression comprise entre 0 et 5V. Etant donnée que la valeur maximale de tension applicable à la carte Nucléo est de 3,3V, on utilise un pont diviseur de tension, en prenant comme résistances 27kΩ et 44kΩ. Le capteur est alimenté en V_S =5V.

La relation donnée par le constructeur liant la tension de sortie et le capteur de pression est $V_{out} = V_S(0.009P + 0.04)$. La précision de mesure dépend de la température.

Seules les bornes 1, 2 et 3 sont utiles pour ce capteur. Elles correspondent respectivement à la tension de sortie, à la masse et à la tension d'alimentation.

3. Architecture de communication

3.1. Objectifs de la communication

Le système doit permettre une liaison entre l'opérateur et le robot, dans les deux sens. Le robot doit recevoir les consignes de déplacement envoyées par l'opérateur, et l'opérateur doit recevoir les données émises par le robot, regroupant des ensembles de mesures (température, pression).

3.2. Protocole de communication

3.2.1. Communication de l'opérateur au robot

Les messages transmis par l'opérateur au robot sont des chaînes de caractères du type "a*mot1mot2...motn*z" où le caractère"z" marque la fin du message et ''a'' son début, et chaque "*moti*" est une consigne.

Ces consignes sont formées d'un caractère donnant le type de la consigne, d'un espace, et d'une valeur associée à la consigne composée de 3 chiffres. Les 4 types de consignes sont "avancer" (caractère "u"), "reculer" (caractère "d"), "tourner à gauche" (caractère "l"), "tourner à droite" (caractère "r"). La valeur associée est, selon la consigne, une longueur en cm ou un angle en degrés. Par exemple, "u 400" signifie "avancer de 400 cm".

La longueur maximale des messages à été fixée provisoirement à 10 consignes, mais elle reste à préciser en fonction des futures contraintes (fréquence d'envoi des messages, durée souhaitée du parcours du robot en autonomie…).

3.2.2. Communication du robot à l'opérateur

Le cahier des charges demande une mesure tous les 10 cm, et un envoi de données toutes les 10 minutes (voir 1.2.2.2.). Cela signifie que le robot doit envoyer des données sous forme de tableaux de mesures. La forme des messages à envoyer reste à déterminer. Elle dépendra de la précision (nombre de chiffres, unité) des valeurs envoyées.

3.3. Mise en place

3.3.1. Architecture électronique

La transmission est réalisée avec une paire de puces KAPPA-M868 (voir documentation sur https://www.tme.eu/Document/546a6ab7ab8675a71bf45776b9edb666/KAPPA-M868-DTE.pdf).

L'un est pour le robot, l'autre pour l'opérateur. Ces composants communiquent via une liaison RS232. Le montage réalisé pour les utiliser est présenté sur le schéma ci-dessous.

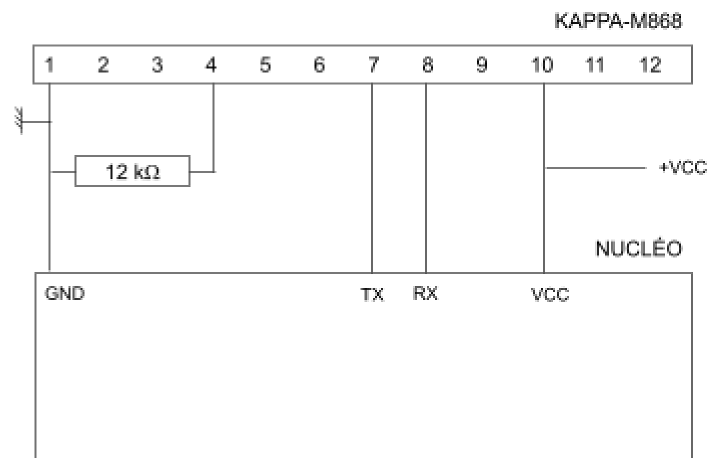


Figure 4 : Schéma du montage utilisé pour relier la puce KAPPA-M868 à la carte Nucléo du robot ou de l'opérateur

3.3.2. Architecture logicielle

L'émission consiste à envoyer la chaîne de caractères souhaitée avec un printf sur le port TX.

La réception est réalisée via une interruption avec la fonction IT_reception dont l'organigramme est donné ci-contre.
Les caractères sont transmis un à un, et stockés dans la variable "caractere". Des variables "debut" et "fin" sont initialisées à 0, et mises à 1 si les caractères de début ("a") et de fin ("z") sont reçus. Elles permettent de déterminer quand commence et quand finit la réception du message. Entre le début et la fin du message (lorsque debut = 1 et fin = 0), les caractères sont plaçés dans la chaîne de caractère "tab", dans le bon ordre grâce à un compteur "i" qui est incrémenté à chaque remplissage de tab.
Lorsque la fin du message est atteinte, les variables "debut", "fin", "i" sont remises à 0 et le message "tab" peut être traité.

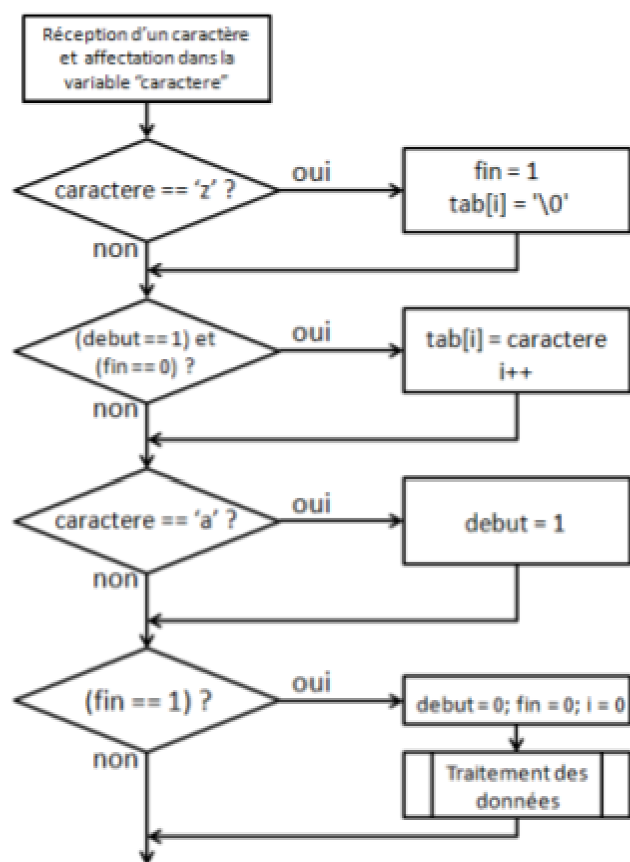


Figure 5 : Algorithme de vérification du message

L’ordre des 3 premières boucles permet de ne pas écrire les caractères de début et de fin du message (“a” et “z”) dans la chaîne tab. On obtient ainsi une suite de consignes de 5 caractères comme définies dans la partie 3.2.1.

Les programmes d’émission et de réception sont donnés en annexe 8.

3.3.3. Tests

La transmission a été testée avec l’envoi répété d’un message type (exemple dans le programme en annexe) et l’affichage dans teraterm du message reçu. On a pu constater que le message était entièrement et correctement restitué. On évalue l’avancement du système de communication au niveau TRL 4 (validation de la technologie en laboratoire, voir annexe 1). Cependant, les liaisons avec l’interface homme-machine et les programmes de contrôle des moteurs du robot posent des problèmes encore non-résolus. Le niveau TRL de la communication pourra être augmenté lorsqu’elle aura été intégrée au reste du robot.

4. Architecture du pilotage

4.1. Mise en oeuvre

4.1.1. Généralités

La motorisation du robot est assurée par deux roues motrices propulsées par des moteurs à courant continu, le troisième point d'équilibre consiste en une roue libre. Les moteurs peuvent recevoir des consignes similaires ou différentes afin d'assurer la translation ou la rotation de l'ensemble machine. Du fait de l'architecture du système, les deux moteurs ont leur sens de rotation par défaut opposés l'un à l'autre. De fait, pour faire avancer ou reculer le robot, on doit donc envoyer des consignes sur le sens de rotation de l'axe moteur différentes tandis que pour une rotation ces consignes seront les mêmes.

4.1.2. Précautions à mettre en place

La mise en place d'un système de pilotage sera soumis à certaines contraintes de performances. Les précisions attendues sur la position et l'angle de l'ensemble obligent à réaliser un asservissement des consignes moteurs comme, par exemple, la vitesse de rotation. Même si les moteurs sont donnés pour identiques par le constructeur, leur réponse à une tension et un courant donné ne sont pas similaires. Certaines zones mortes peuvent impliquer qu'à un courant donné, un moteur puisse être actionné mais pas l'autre. De même, la répartition des masses sur la structure n'est pas égale de part et d'autre de l'axe de symétrie du robot, accentuant la nécessité de la mise en place d'un système d'asservissement.

L'envoi de consignes sur de longues distances peut présenter des risques de corruptions des données. Pour pallier cela, avant tout déplacement de Veronica et dès réception d'un message, une série de tests seront effectués afin de vérifier l'intégrité du message et la cohérence des consignes. Ces tests ne pourront pas éviter toute corruption mais au moins en réduire la portée. Le robot n'initialisera pas sa phase de déplacement avant la fin de cette phase de contrôle.

4.2. Protocole de communication

Le traitement des données relatives au pilotage suivra le protocole mis en place par le système de communication, soit une dimension de message maximale de cinquante caractères correspondant à dix consignes à suivre, chacune composée de cinq caractères.

4.3. Architecture logicielle

4.3.1. Généralités

Le pilotage est réalisé par la fonction **PROTIS_PilotageMoteurFinal_V3**, dont la structure principale est donnée en annexe 1. Cette fonction permet de contrôler le déplacement du robot et regroupe diverses sous-fonctions. Après réception du message comportant les consignes, la fonction s'articule en trois phases : une première phase de contrôle des données de consigne, une deuxième de réécriture et une troisième phase de déplacement. Chacune de ces phases sont elle-même découpée en plusieurs sous-phases.

Le programme est réalisé sous MBED.

4.3.2. Entrées et sorties utilisées

4.3.2.1. Entrées

Comme expliqué dans l'en-tête de la fonction, celle-ci utilise plusieurs ports de la carte d'acquisition. Concernant les informations reçues par la carte, soit les entrées, on utilise un total de six ports.

Quatre ports digitaux servent à recevoir les informations renvoyées par les encodeurs des deux moteurs, ils sont référencés D4, D5, D8 et D9.

Les deux derniers ports sont des ports série TX/RX et servent à la réception des consignes de déplacement, ils sont référencés PC10 et PC11.

4.3.2.2. Sorties

Les sorties sont nécessaires au pilotage direct du moteur. Pour cela on utilise deux ports digitaux, D2 et D6 et deux sorties de puissance D3 et D7.

4.3.3. Réception des données

La partie réception suit le protocole de communication et est réalisée par le partie communication. Les sorties et entrées correspondent à celle utilisée par le reste des commandes

4.3.4. Contrôle des données de consignes

4.3.4.1. Généralités

Cette phase de contrôle sera assurée par un ensemble de tests sous condition permettant de vérifier que la taille de la chaîne de caractère, la valeur du premier et du deuxième caractère de chaque consigne ainsi que la valeur du déplacement souhaité soit en

adéquation avec le protocole mis en place. Chaque test renverra une valeur sur la variable notée *check_i*, avec *i* représentant l'indice du test (1 pour le test 1 par exemple). Ce sont ces valeurs qui, une fois testées, permettent de mettre à jour la valeur de la variable *Flag*. On affecte la valeur false à cette variable en entrée de fonction.

La structure de la sous fonction de contrôle est donnée en annexe 3.

4.3.4.2. Test 1 : taille du message

Ce premier test contrôle la longueur du tableau de caractère *Message*. Si les consignes envoyées n'ont pas été altérées lors de leur envoi, de leur réception ou par des perturbations lors de leur transmission, ce tableau devrait comporter des caractères d'intérêt jusqu'à un index multiple de 5, en comptant le caractère de fin de message.

Pour cela on réalise un test conditionnel sur la valeur *Stop* dans laquelle on a stocké la valeur de position du caractère de fin de message. On applique à cette valeur l'opérateur modulo, retournant le reste d'un division entière, et on compare le résultat à celui souhaité, à savoir 0. Lorsque le résultat obtenu est 0, on assigne à la variable locale *check_1* la valeur 1 sinon cette variable reste égale à 0.

4.3.4.3. Test 2 : contrôle du premier caractère de chaque consigne

Ce deuxième test contrôle la valeur du premier caractère de chaque consigne. Le tableau *Message* peut contenir jusqu'à dix consignes de cinq caractères chacune, on va donc tester les premières valeurs de ces consignes en commençant à l'index 0 du tableau et en le parcourant par pas de 5 et ce tant que la valeur d'index est inférieure à la variable *Stop* qui représente l'index de fin des consignes.

On réalise un test conditionnel en deux temps, et en premier lieu on va s'intéresser au tableau d'entier nommé *check_21*, de taille dix. En utilisant une boucle conditionnelle, on teste si les caractères ciblés de *Message* ont bien pour valeur *u*, *d*, *l* ou *r* et on remplit le tableau de vérification par des 1 si la condition est remplie. Si cette dernière n'est pas rempli le tableau conserve la valeur qui lui a préalablement alloué à cet index (ici 0). En second lieu, on réalise une boucle afin d'affecter à une seconde valeur *check_22* la somme de *check_21* prise entre l'index 0 et l'index égale à *Stop*.

C'est la valeur de *check_22* que l'on va tester pour mettre à jour *Flag*.

4.3.4.4. Test 3 : contrôle du deuxième caractère de chaque consigne

Ce deuxième test contrôle la valeur du deuxième caractère de chaque consigne. Si les consignes de départ n'ont pas été modifiées par l'envoi, la réception ou pendant la transmission, ce deuxième caractère doit être un espace.

Comme pour le test 2, on réalise un test conditionnel en deux temps, ici on comparera la valeur du deuxième caractère à la valeur attendue et on remplira le tableau d'entier *check_31* et la variable locale *check_32* en conséquence.

C'est la valeur de *check_32* que l'on va tester pour mettre à jour *Flag*.

4.3.4.5. Test 4 : contrôle des données de déplacement de chaque consigne

Ce troisième test contrôle la valeur du déplacement donné par chaque consigne. Le déplacement relatif à une consigne est donné par les troisième, quatrième et cinquième caractères de cette consigne composée de cinq caractères. Celui en position trois correspond au chiffre des centaines, en position deux les dizaines et le dernier aux unités.

Pour ce test on va utiliser un variable temporaire notée *check_Cons* qui prendra la valeur du déplacement imposé par un consigne. Cette variable sera comparé à deux valeurs en fonction du type de déplacement, soit *0* et *999* pour une translation et *0* et *180* pour une rotation. Comme pour les deux tests précédents, on affecte la valeur 1 à un index du tableau d'entier *check_41* si les conditions sont remplies par une consigne et on les parcourt toutes. On affecte ensuite une valeur à la variable *check_42* en conséquence.

C'est la valeur de *check_42* que l'on va tester pour mettre à jour *Flag*.

4.3.4.6. Test 5 : contrôle des valeurs de vérification

Ce dernier contrôle permet de mettre à jour la variable *Flag* en fonction des valeurs de vérification obtenues (*check_1*, *check_22*, etc.). Si la valeur *check_1* vaut 1 et que *check_22=check_32=check_42=Stop/5*, alors la variable *Flag* peut être mise à true et ainsi le reste du programme peut s'exécuter.

4.3.5. Réécriture des données

La sous-fonction *Ecriture()* est assez simple et permet d'écrire le message reçu par le robot en une ou plusieurs consignes à suivre. Pour ce faire on utilise deux nouvelles variables, *ConsDis* et *ConsDep*, toutes deux des tableaux de dimension dix, la première étant un tableau de double et la seconde de caractères.

Sur chaque index de *ConsDep*, on va stocker le caractère marquant le type de déplacement que devra suivre le robot soit 'u' pour avancer, 'd' pour reculer, etc.

ConsDis contient dans chaque index la distance ou l'angle à parcourir relatif au type de déplacement souhaité pas le même index du tableau *ConsDep*.

Les deux tableaux ne pourront accueillir que dix consignes maximum, en accord avec le protocole mis en place.

4.3.6. Déplacement du robot

4.3.6.1. Généralités

Pour comprendre le fonctionnement de la sous-fonction, il faut avoir en mémoire les composants utilisés, soit deux moteurs à courant continu équipé d'un moto-réducteur, chacun piloté par un pont en H permettant la mise en marche, via la sortie *en*, et le choix du sens de

rotation du moteur, via *dir*. Chaque moteur est également équipé de deux encodeurs, noté *Sa* et *Sb*, qui permettent de quantifier la rotation de l'axe principal du moteur. Ces encodeurs sont placés à 90° l'un de l'autre, si bien que les signaux d'intérêt qu'il délivre seront déphasés de 90°, autorisant une plus grande précision de la position de l'axe. Il est à noter que si pour un moteur l'encodeur *Sa* et actionné avant l'encodeur *Sb*, c'est l'inverse pour l'autre moteur du fait de l'architecture du système.

Cette sous-fonction assure le déplacement du robot dans le respect des consignes qui lui ont été imposées. Son squelette principal est une instruction *switch/case*, enveloppé d'une boucle *for* permettant de parcourir l'ensemble des consignes et comprenant une boucle *while* permettant la mise en place d'une condition dans chaque cas possible.

Même si la sous-fonction peut sembler complexe de par sa taille, il s'agit en fait de la même structure répété quatre fois, une pour chaque type de déplacement et appelé grâce au *case*.

4.3.6.2. Fonctionnement de la sous-fonction

On va s'intéresser ici au fonctionnement de la sous-fonction de déplacement dans le cas où l'on reçoit une seule consigne, on ne parlera donc par de la boucle *for* et on ne présentera qu'un type de déplacement, les autres étant semblables en tout point.

Toutes les variables utiles ont d'abord été mises à zéro à l'entrée dans le *main*, c'est variables existent pour les deux moteurs avec l'indication *_D* ou *_G* suivant le moteur auquel elles sont liées. Elles permettent notamment de comptabiliser l'angle effectué par l'axe moteur en sortie du motoréducteur suivant les informations des encodeurs avec *Angle_Sa* et *Angle_Sb* et de calculer le déplacement le déplacement associée en fonction des dimensions des roues avec *Pos_Sa* et *Pos_Sb*. Deux variables sont quant à elle communes au deux moteurs, il s'agit de *Angle_Final* et *Pos_Final* qui quantifient la totalité du déplacement effectué par le robot à un instant *t*. Ce sont ces deux données que l'on va comparer à la valeur de consigne dans l'instruction *while* afin de stopper le déplacement lorsque la consigne est atteinte. Deux autres variables existent également, *old_Sa* et *old_Sb*, mais ne servent qu'à stocker la valeur de *Sa* ou Si à *t-1* et permettre ainsi de la comparer à celle obtenue à *t* afin d'effectuer un test avant de quantifier le déplacement.

Lors de la quantification du mouvement qu'a effectué le robot, il faut prendre une précaution particulière concernant le déphasage de 90° sur l'un des encodeurs d'un de chacune des roues, comme mentionné précédemment.

Pour prendre l'exemple d'une avancée en ligne droite, piloté par le caractère 'u' et la distance associé, on rentre dans la boucle while si la position cible n'est pas encore atteinte pour on fixe un rapport cyclique pour chacun des moteurs et enfin on règle le sens de rotation, ici dans le sens horaire pour l'un des moteurs et dans le sens trigonométrique pour l'autre. Enfin on incrémente les valeurs de déplacement grâce aux encodeurs et on en calcule la moyenne pour obtenir la valeur *Pos_Final* que l'on compare en permanence à la consigne. Lorsque cette valeur atteint la valeur cible, on sort de la boucle et on remet toutes les variables utilisées à leur valeur par défaut, ici 0, on arrête les moteurs et on positionne

l'indicateur *Flag* à false, permettant ainsi l'attente de la réception d'un nouveau paquet de données.

Enfin cette sous-fonction fonctionnera de paire avec la sous-fonction d'asservissement qui pilotera la valeur du rapport cyclique à injecter dans l'un des moteurs.

4.3.7. Asservissement en vitesse

Avant toute chose, il faut signaler que la fonction est à un stade très primaire de développement, certaines des grandeurs utilisées sont encore à définir correctement. La sous-fonction est donc présente dans la fonction principale mais non utilisée pour le moment.

Comme il l'a été précisé dans l'introduction de cette partie, les performances attendues en termes de mobilité nécessitent de réaliser un asservissement du déplacement du robot. Il a été choisi ici de réaliser un asservissement en vitesse des moteurs via leur rapport cyclique. Celui-ci sera réalisé sur la moteur gauche avec pour cible le moteur droit. Même si ce type d'asservissement ne nous permettra pas d'atteindre une vitesse cible, il assurera néanmoins un déplacement homogène de Veronica. La sous-fonction est donnée en annexe 7.

On réalise un asservissement type PID qui fournira une valeur de correction à appliquer au rapport cyclique dirigeant le moteur de gauche. La partie proportionnelle sera proportionnelle à la différence pure entre le rapport cyclique réel du moteur droit et celui mesuré du moteur gauche à un coefficient *Kp* prêt. Pour l'intégrateur on s'interessera à la somme de ces erreurs, toujours à un facteur prêt, ici *Ki*. Et enfin le dérivateur sera lié à la différence entre la valeur d'erreur obtenue à N et celle à N-1, on lui appliquera le facteur *Kd*.

Pour le moment toute *Kp*, *Ki* et *Kd* sont fixés égale à zéro afin d'éviter un modification du rapport cyclique lors d'une entrée involontaire dans la boucle.

Pour ce qui est de l'implémentation, il est envisagé d'associer la sous-fonction à une interruption qui se produira à intervalle régulier, décidé par la fréquence d'échantillonnage que l'on choisira.

5. Réalisation d'une interface

5.1. Description

Pour pouvoir piloter le robot, il faut lui envoyer toutes les consignes établies selon le protocole de communication précédent. Cela se fait par le biais d'une Interface Homme-Machine, qui permettra d'envoyer les commandes au robot en passant par l'intermédiaire d'une carte Nucléo.

L'interface doit être simple d'utilisation. Elle doit être suffisamment intuitive pour être prise en main rapidement et sans difficulté majeure. Pour cela, la saisie des commandes doit être suffisamment claire pour éviter une fausse manipulation. L'interface sera réalisée à l'aide du logiciel MatLab App Designer.

5.1.1. Architecture de l'interface

L'interface se décompose principalement en 2 parties distinctes : une partie "envoi de données" et une partie "visualisation des mesures".

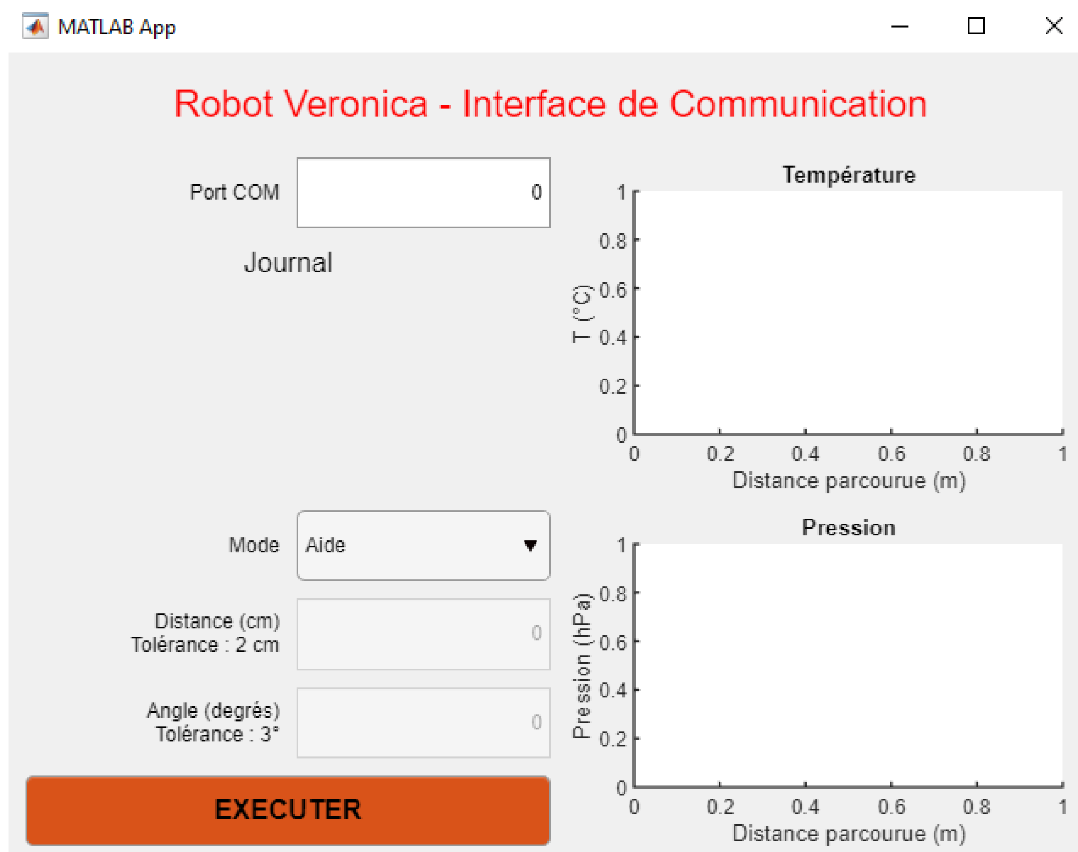


Figure 5 : Page de l'Interface Homme Machine

La partie "envoi de données", située à gauche, comprend plusieurs blocs distincts. On peut y voir de haut en bas les blocs suivants :

- Port COM : Permet de saisir le port de communication avec la carte Nucléo
- Journal : Liste toutes les actions effectuées, de la plus récente à la plus ancienne
- Mode : Permet de sélectionner le mode désiré (voir partie suivante pour plus de détails). Il est positionné par défaut sur Aide.
- Distance (cm) : Permet de saisir la distance à parcourir par le robot en ligne droite, souhaitée par l'utilisateur. Elle doit être comprise entre 0 et 999 cm. La saisie dans cette case est possible selon certains modes.
- Angle (degrés) : Permet de saisir l'angle de rotation à effectuer par le robot sur lui-même, souhaitée par l'utilisateur. Elle doit être comprise entre 0 et 180°. La saisie dans cette case est possible selon certains modes.
- Exécuter : Permet l'exécution de la consigne.

La partie "visualisation des mesures" située à droite, comprend 2 graphiques, reprenant les valeurs de température et de pression effectuées par le robot. Ces graphiques n'affichent pas de valeurs au début, puisqu'il faut attendre 10 minutes pour que le robot envoie ses mesures d'après le cahier des charges.

5.1.2. Modes de commande

Un mode de commande est un choix possible que peut faire l'utilisateur pour contrôler le robot et l'interface graphique. Par défaut, il est positionné sur Aide à l'ouverture de l'interface. Les modes possibles sont les suivants :

- Aide : Permet l'ouverture de la page d'aide, présentée sous la forme d'un LiveScript MatLab. Elle reprend ce qui est dit dans toute cette partie.
- Liste Ports COM : Permet d'afficher dans le Journal la liste des ports COM disponibles. Ce mode est particulièrement utile pour trouver le port à renseigner dans le bloc Port COM.
- Connecter : Permet la connexion de l'interface à la carte Nucléo par une liaison série RS232.
- Avancer : Permet de contrôler le déplacement du robot en marche avant. Le déplacement dépend de la valeur entrée dans la case Position (cm).
- Reculer : Même chose que Avancer, mais en marche arrière.
- Tourner à gauche : Permet de contrôler la rotation du robot sur lui-même, dans le sens antihoraire. L'angle de rotation dépend de la valeur entrée dans la case Angle (degrés).
- Tourner à droite : Même chose que tourner à gauche, mais dans le sens horaire.
- Autodestruction : Ce mode est une blague du développeur. Il n'a pas été implémenté.

A l'exception du mode Liste Ports COM, l'exécution de la commande écrit un message dans le journal, dépendant de la nature de la commande et de son succès.

5.2. Protocole de communication

Le protocole de communication est le même que celui établi entre l'émetteur et le récepteur. Il reprend les mêmes caractères de début et de fin et la même forme de message.

La seule différence entre la partie communication et celle-ci réside dans l'architecture. L'ordinateur et la carte Nucléo sont reliés par un câble USB - mini USB type B. La communication se fait en liaison série RS232.

5.3. Architecture logicielle

Le code complet de l'interface est disponible en annexe 9.

Les tests effectués indiquent un problème de communication entre la carte Nucléo et l'interface. En effet, l'interface envoie à priori les consignes mais la carte n'arrive pas à les lire. Cela a rendu impossible la réalisation de la réception des mesures par l'interface. Les graphiques ainsi paramétrés n'affichent donc pas de courbes.

Il reste donc à régler ce problème et réaliser la partie réception sur l'interface.

6. Performances énergétiques

6.1. Nécessité

Notre robot doit avoir une autonomie de 1 km avec tous les composants intégrés. Pour valider cet aspect du cahier des charges, nous devons connaître la puissance consommée par notre robot, et donc effectuer certains tests énergétiques.

6.2. Protocole

Nous voulons mesurer le courant et le voltage parcourue dans le circuit électrique lorsque le moteur est en marche et le robot sur le sol. Nous avons pour cela une alimentation variable et un ampèremètre pour nous indiquer l'intensité et la tension du courant.

Idéalement nous voulons mesurer la puissance lorsque le robot est à même le sol. Pour des raisons pratiques nous ne pouvons pas le faire; nous avons donc dû freiner les roues avec les mains; afin de simuler l'action du sol sur les roues. On doit aussi mesurer la tension fournie par la batterie (en pleine charge et lorsque la batterie est presque vide). La tension de la batterie a été dans notre cas directement fournie par M.Villemejane.

On doit aussi estimer la vitesse du robot lorsqu'elle est alimentée par la batterie.

6.3. Résultats

Pour une tension de 11.8V nous avons mesuré un courant de 1.4A. Les batteries livrent 7.8V lorsqu'elles sont chargées et peuvent descendre à 6.5V. On prendra une tension moyenne 7.2V. L'autonomie du véhicule sera donc de $\frac{4}{1.4} * \frac{7.2}{11.8}$=1h45.

Au vu des tests qu'on a fait on peut estimer la vitesse du robot à 20 cm/s soit 720m/h. On a donc une autonomie de 720m/h*1.75=1.260km

Le cahier des charges est rempli.

6.4. Axes d'amélioration

Les mesures que nous avons faites sont très approximatives; il est difficile de reproduire la consommation des roues sur un sol réel. Pour ce qui est de la vitesse du robot; là encore il est difficile de la mesurer; du moins on peut estimer que la vitesse du robot se situe aux alentours de 20 cm/s.

7. Conclusion

Afin d'illustrer l'avancement du projet, nous avons mis en place un classement de maturité des principales fonctions que doit opérer le robot Veronica. Le niveau de maturité associé à chaque fonction est résumé dans le tableau suivant :

Fonction	**Sous-fonction**	**Maturité 1 : Prise en main**	**Maturité 2 : Fonction en réalisation**	**Maturité 3 : Fonction finalisée**	**Maturité 4 : Test effectué**	**Maturité 5 : Fonction validée**
Télécommunication	*Émission*					
	Réception					
Pilotage	*Contrôle*					
	Écriture					
	Déplacement					
	Asservissement					
IHM						
Architecture du robot						
Capteurs						

Une case verte correspond à une validation, une case jaune correspond à une réalisation pas totalement effectuée. On constate ainsi qu'un certain nombre de tâches restent à accomplir pour que le robot soit parfaitement fonctionnel.

Annexes

Annexe 1 : Structure principale de la fonction de déplacement

```
/* Auteur(s) : Veronica 2(Achraf Ayeb - Antoine Beignet  - Samuel Aychet Claisse  - Loïck Ripoteau
 *  Device : Nucleo-L476RG
 *
 *  Objectif fonction : piloter deux moteurs à courant continu, à l'aide d'un
 *                        point en H en suivant des consignes envoyées à distance
 *
 *  Entrée(s) :   -   digitale(s) : ports D4,D5 pour le moteur gauche
 *                        ports D8,D9 pour le moteur droit
 *            -   Série :   ports PC10, PC11  pour la réception de consigne
 *
 *
 *  Sortie(s) :   -   digitale(s) : ports D2 pour le moteur gauche
 *                        ports D6 pour le moteur droit
 *            -   Pwm :                  ports D3 pour le moteur gauche
 *                        ports D7 pour le moteur droit
 *
 *  Création   :   16 mars 2021
 *  Mise(s) à jour   :   vide
 */

#include "mbed.h"/*importation librairie mbed*/
#include "math.h"/*importation librairie math*/
#include "string.h"/*importation librairie string*/
#define pi 3.14159265358979323846
#define r 6// en cm

//  Initialisation et déclaration des entrée(s), sortie(s)
DigitalOut dir_G(D2);// Sortie PWM pour moteur gauche
PwmOut en_G(D3);// Sortie PWM pour moteur gauche

DigitalOut dir_D(D6);// Sortie PWM pour moteur droit
PwmOut en_D(D7);// Sortie PWM pour moteur droit

DigitalIn Sa_G(D4);// Position de l'axe moteur gauche
DigitalIn Sb_G(D5);
DigitalIn Sa_D(D8);// Position de l'axe moteur droit
DigitalIn Sb_D(D9);

Serial Consigne(PC_10,PC_11);// Consigne de déplacement à suivre

// Déclaration des variables globales
double rc_D = 0.5;// commande du moteur
double rc_G = 0.5;
int sens_G;
int sens_D;
int SensTab[1];
double ConsDis[9];// Variable liée à la distance du déplacement
char ConsDep[9];// Variable liée au type de déplacement
double old_Cons;
char Message[50];

double OldErreur = 0;// Variables utilisées pour l'asservissement
double SommeErreur = 0;
int Kp = 0;// Valeurs à fixer expérimentalement
int Ki = 0;
int Kd = 0;

int old_Sa_G;// Variables de stockage de la position de l'axe moteur gauche
```

```
int old_Sb_G;
int old_Sa_D;// Variables de stockage de la position de l'axe moteur droit
int old_Sb_D;

double Angle_Sa_G;// Variables pour mise en mémoire du déplacement angulaire
double Angle_Sb_G;
double Angle_Sa_D;
double Angle_Sb_D;
double Angle_Final;
double Pos_Sa_G;// Variables pour mise en mémoire du déplacement
double Pos_Sb_G;
double Pos_Sa_D;
double Pos_Sb_D;
double Pos_Final;

int Position;// marqueur de position

bool Flag;// Variable permettant la vérification des consignes
int Stop;// variable pour allouer l'index du tableau relatif à la fin des consignes


// Déclaration des sous-fonctions utiles
void Reception(void);
void Verification(void);
void Ecriture(void);
void Deplacement(void);
void AsservissementRC(void);

//  Ecriture de la fonction
int main()
{
        SensTab[0] = 0;
        SensTab[1] = 1;

        Angle_Sa_G = 0;// Assignation des valeurs liées au déplacement angulaire
        Angle_Sb_G = 0;
        Angle_Sa_D = 0;
        Angle_Sb_D = 0;
        Angle_Final = 0;
        Pos_Sa_G = 0;// Assignation des valeurs liées au déplacement
        Pos_Sb_G = 0;
        Pos_Sa_D = 0;
        Pos_Sb_D = 0;
        Pos_Final = 0;

        en_D.period_ms(10);
        en_G.period_ms(10);


        Flag = false; // Vérificateur de validité
        Position = 0; // Position dans le message reçu
        Consigne.attach(&Reception);

        while(1) {

                if(Flag){
                        // ACTIONS
                        Verification();
                        Ecriture();
                        Deplacement();
                }
        }
}
```

Annexe 2 : Structure de la sous-fonction Reception()

```
void Reception(){
	// Variables locales
	char StckChr;// variable de stockage temporaire d'un caractère de la consigne

	// Initialisation des variables
	Position = 0;

	// Parcours du message reçu et assignation dans un tableau
	while(Stop == 0){
		StckChr = Consigne.getc();

		if (StckChr == '\0'){
			Message[Position] = StckChr;
			Stop = Position;
			Flag = true;
		}
		else {
			Message[Position] = StckChr;
			Position++;
		}
	}
}
```

Annexe 3 : Structure de la sous-fonction Verification()

```
void Verification(){
        // Déclaration des variables locales
        int check_1 = 0;
        int check_21[9];
        int check_22 = 0;
        int check_31[9];
        int check_32 = 0;
        int check_Cons = 0;
        int check_41[9];
        int check_42 = 0;
        int i;

        // Mise en défaut du vérificateur
        Flag = false;

        // Remplissage des tableaux de check
        for (i=0;i<9;i++){
                check_21[i] = 0;
                check_31[i] = 0;
                check_41[i] = 0;
        }

        // Check de la taille du message
        if (Stop%5 == 0){
                check_1 = 1;
        }

        // Check des premières données de chaque consigne
        for (i=0;i<(Stop/5)-1;i++){
                if ((Message[i*5] == 'u') || (Message[i*5] == 'd') || (Message[i*5] == 'l') || (Message[i*5] == 'r')){
                        check_21[i] = 1;
                }
        }
        for (i=0;i<(Stop/5)-1;i++){
                check_22 += check_21[i];
        }

        // Check des deuxièmes données de chaque consigne
        for (i=0;i<(Stop/5)-1;i++){
                if (Message[(i*5)+1] == ' '){
                        check_31[i] = 1;
                }
        }
        for (i=0;i<(Stop/5)-1;i++){
                check_32 += check_31[i];
        }

        // Check des données de déplacement de chaque consigne
        for (i=0;i<(Stop/5)-1;i++){
                check_Cons = (100*(Message[(i*5)+2]-'0')) + (10*(Message[(i*5)+3]-'0')) + (Message[(i*5)+4]-'0');
                if ((Message[i*5] == 'u') || (Message[i*5] == 'd')){
                        if (check_Cons <=999 && check_Cons >=0){
                                check_41[i] = 1;
                        }
                }
                else {
                        if (check_Cons <=180 && check_Cons >=0){
                                check_41[i] = 1;
                        }
                }
        }
```

```
    for (i=0;i<(Stop/5)-1;i++){
        check_42 += check_41[i];
    }

    // Mise à jour du flag
    if ((check_1 == 1) && (check_22 == (Stop/5)-1) && (check_22 == check_32) && (check_22 == check_42)){
        Flag = true;
    }

}
```

Annexe 4 : Structure de la sous-fonction Ecriture()

```
void Ecriture(){
        // Déclaration des variables locales
        int i;// variable d'incrément

        // Remplissage du tableau lié à la direction de déplacement
        for (i=0;i<(Stop/5)-1;i++){
                ConsDep[i] = Message[i*5];
        }
        // Remplissage du tableau lié à la distance/angle de déplacement
        for (i=0;i<(Stop/5)-1;i++){
                ConsDis[i] = (100*(Message[(i*5)+2]-'0')) + (10*(Message[(i*5)+3]-'0')) + (Message[(i*5)+4]-'0');
        }
}
```

Annexe 5 : Structure de la sous-fonction Deplacement()

```
void Deplacement(){

	// Variable locale
	int i;// variable d'incrément

	// Boucle et condition de déplacement
	for (i=0;i<(Stop/5);i++){
	switch(ConsDep[i])
	{

	//-------------------------------------------------------------------
	case 'u' :
		// Condition pour avancer en ligne droite
		while (Pos_Final < ConsDis[i]){
		en_G.write( rc_G);
		dir_G = SensTab[0];
		en_D.write( rc_D);
		dir_D = SensTab[1];

		// Conditions pour assigner les valeurs de déplacement du moteur gauche
			if ((Sa_G.read() != old_Sa_G) && (Sa_G.read() == 1)){
				Angle_Sa_G += (360/19.225)+90;
				Pos_Sa_G += (Angle_Sa_G/180)*pi*r;
			}
			old_Sa_G = Sa_G.read();
			if ((Sb_G.read() != old_Sb_G) && (Sb_G.read() == 1)){
				Angle_Sb_G += 360/19.225;
				Pos_Sb_G += (Angle_Sb_G/180)*pi*r;
			}
			old_Sb_G = Sb_G.read();

		// Conditions pour assigner les valeurs de déplacement du moteur droit
			if ((Sa_D.read() != old_Sa_D) && (Sa_D.read() == 1)){
				Angle_Sa_D += 360/19.225;
				Pos_Sa_D += (Angle_Sa_D/180)*pi*r;
			}
			old_Sa_D = Sa_D.read();
			if ((Sb_D.read() != old_Sb_D) && (Sb_D.read() == 1)){
				Angle_Sb_D += (360/19.225)+90;
				Pos_Sb_D += (Angle_Sb_D/180)*pi*r;
			}
			old_Sb_D = Sb_D.read();

		// Test pour voir si la distance souhaitée est atteinte et sortir de la boucle
			Angle_Final += (((Angle_Sa_G+Angle_Sb_G)/2)+((Angle_Sa_D+Angle_Sb_D)/2))/2;
			Pos_Final += (((Pos_Sa_G+Pos_Sb_G)/2)+((Pos_Sa_D+Pos_Sb_D)/2))/2;

		}
	break;

	//---------------------------------------------------------------
	case 'd' :
		// Condition pour reculer en ligne droite
		while (Pos_Final < ConsDis[i]){
		en_G.write( rc_G);
		dir_G = SensTab[1];
		en_D.write( rc_D);
		dir_D = SensTab[0];
```

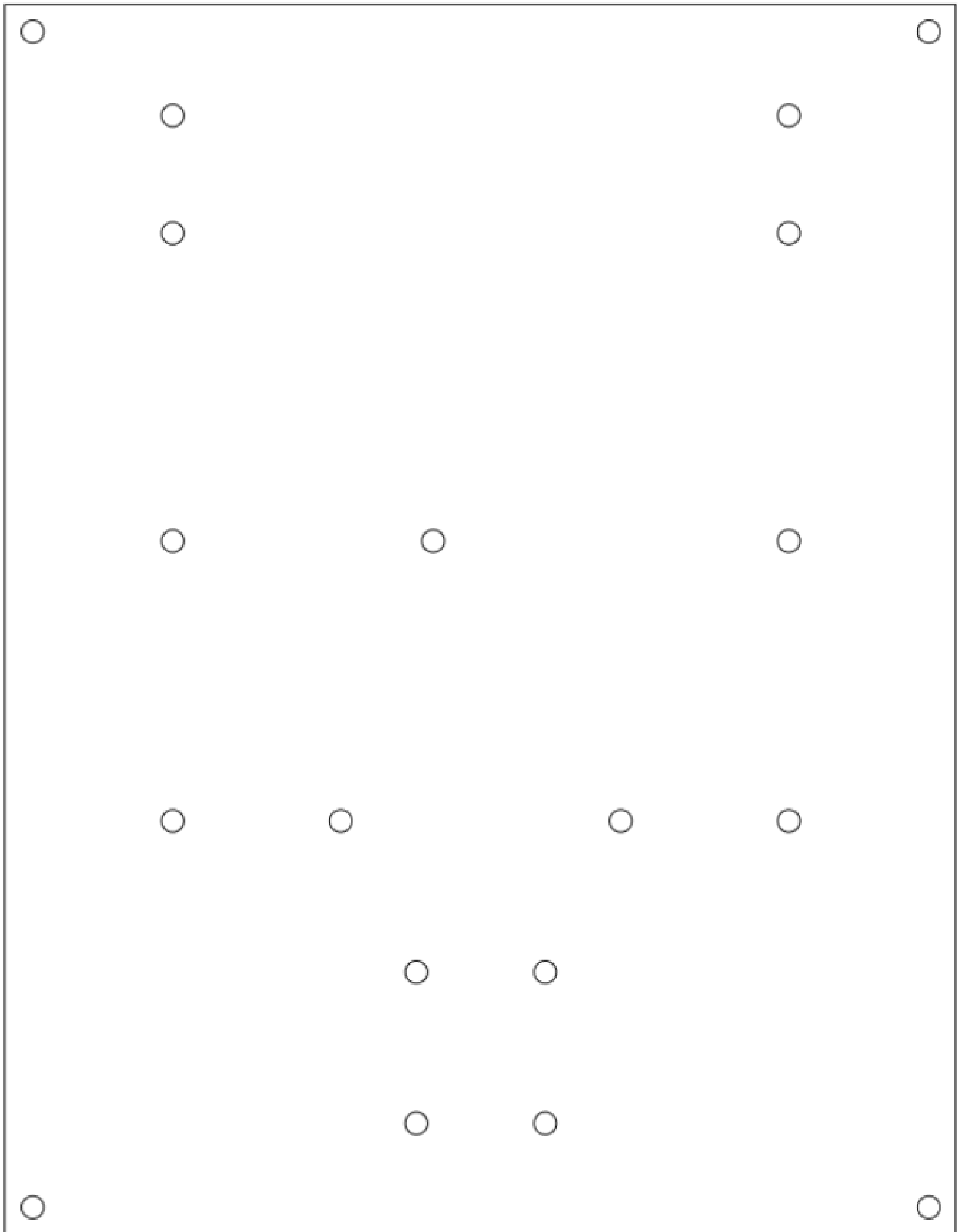
```
		// Conditions pour assigner les valeurs de déplacement du moteur gauche
			if ((Sa_G.read() != old_Sa_G) && (Sa_G.read() == 1)){
				Angle_Sa_G += 360/19.225;
				Pos_Sa_G += (Angle_Sa_G/180)*pi*r;
			}
			old_Sa_G = Sa_G.read();
			if ((Sb_G.read() != old_Sb_G) && (Sb_G.read() == 1)){
				Angle_Sb_G += (360/19.225)+90;
				Pos_Sb_G += (Angle_Sb_G/180)*pi*r;
			}
			old_Sb_G = Sb_G.read();

		// Conditions pour assigner les valeurs de déplacement du moteur droit
			if ((Sa_D.read() != old_Sa_D) && (Sa_D.read() == 1)){
				Angle_Sa_D += (360/19.225)+90;
				Pos_Sa_D += (Angle_Sa_D/180)*pi*r;
			}
			old_Sa_D = Sa_D.read();
			if ((Sb_D.read() != old_Sb_D) && (Sb_D.read() == 1)){
				Angle_Sb_D += 360/19.225;
				Pos_Sb_D += (Angle_Sb_D/180)*pi*r;
			}
			old_Sb_D = Sb_D.read();

		// Test pour voir si la distance souhaitée est atteinte et sortir de la boucle
			Angle_Final += (((Angle_Sb_G+Angle_Sa_G)/2)+((Angle_Sb_D+Angle_Sa_D)/2))/2;
			Pos_Final += (((Pos_Sa_G+Pos_Sb_G)/2)+((Pos_Sa_D+Pos_Sb_D)/2))/2;

		}
	break;

	//-----------------------------------------------------------------
	case 'l':
		// Condition pour tourner dans le sens trigonométrique
		while (Angle_Final < ConsDis[i]){
		en_G.write( rc_G);
		dir_G = SensTab[1];
		en_D.write( rc_D);
		dir_D = SensTab[0];

		// Conditions pour assigner les valeurs de déplacement du moteur gauche
			if ((Sa_G.read() != old_Sa_G) && (Sa_G.read() == 1)){
				Angle_Sa_G += (360/19.225)+90;
			}
			old_Sa_G = Sa_G.read();
			if ((Sb_G.read() != old_Sb_G) && (Sb_G.read() == 1)){
				Angle_Sb_G += 360/19.225;
			}
			old_Sb_G = Sb_G.read();

		// Conditions pour assigner les valeurs de déplacement du moteur droit
			if ((Sa_D.read() != old_Sa_D) && (Sa_D.read() == 1)){
				Angle_Sa_D += (360/19.225)+90;
			}
			old_Sa_D = Sa_D.read();
			if ((Sb_D.read() != old_Sb_D) && (Sb_D.read() == 1)){
				Angle_Sb_D += 360/19.225;
			}
			old_Sb_D = Sb_D.read();

		// Test pour voir si la distance souhaitée est atteinte et sortir de la boucle
			Angle_Final += (((Angle_Sa_G+Angle_Sb_G)/2)+((Angle_Sa_D+Angle_Sb_D)/2))/2;
		}
	break;
```

```
//------------------------------------------------------------------
case 'r' :
        // Condition pour tourner dans le sens horaire
        while (Angle_Final < ConsDis[i]){
        en_G.write rc_G);
        dir_G = SensTab[1];
        en_D.write( rc_D);
        dir_D = SensTab[0];

        // Conditions pour assigner les valeurs de déplacement du moteur gauche
                if ((Sa_G.read() != old_Sa_G) && (Sa_G.read() == 1)){
                        Angle_Sa_G += 360/19.225;
                }
                old_Sa_G = Sa_G.read();
                if ((Sb_G.read() != old_Sb_G) && (Sb_G.read() == 1)){
                        Angle_Sb_G += (360/19.225)+90;
                }
                old_Sb_G = Sb_G.read();

        // Conditions pour assigner les valeurs de déplacement du moteur droit
                if ((Sa_D.read() != old_Sa_D) && (Sa_D.read() == 1)){
                        Angle_Sa_D += 360/19.225;
                }
                old_Sa_D = Sa_D.read();
                if ((Sb_D.read() != old_Sb_D) && (Sb_D.read() == 1)){
                        Angle_Sb_D += (360/19.225)+90;
                }
                old_Sb_D = Sb_D.read();

        // Test pour voir si la distance souhaitée est atteinte et sortir de la boucle
        Angle_Final += (((Angle_Sa_G+Angle_Sb_G)/2)+((Angle_Sa_D+Angle_Sb_D)/2))/2;
        }
break;
}
}

// Remise en défaut du Flag
Flag = false;

// Arrêt des moteurs
en_G.write( rc_G);
en_D.write( rc_D);

//Remise à zéro des données
Angle_Sa_G = 0;// Déplacement angulaire
Angle_Sb_G = 0;
Angle_Sa_D = 0;
Angle_Sb_D = 0;
Angle_Final = 0;
Pos_Sa_G = 0;// Déplacement en translation
Pos_Sb_G = 0;
Pos_Sa_D = 0;
Pos_Sb_D = 0;
Pos_Final = 0;

}
```

Annexe 6 : Structure de la sous-fonction Asservissement()

```
void AsservissementRC(){// On choisit rc_D comme valeur cible

    // Déclaration des variables locales
    double rc_G_Reel;
    double rc_D_Reel;
    double Erreur = 0;
    double Delta = 0;
    double Correction = 0;

    // Assignation de leurs valeurs aux variables
    rc_G_Reel = 0;// Ce calcul reste à déterminer

    Erreur = rc_D_Reel - rc_G_Reel;// Calcul lié au propotionnel

    SommeErreur += Erreur;// Calcul lié à l'intégrateur

    Delta = Erreur - OldErreur;// Calcul lié au dérivateur

    Correction = Kp*Erreur + Ki*SommeErreur + Kd*Delta;

    // Mise en place d'un test pour borner les corrections
    if (Correction > 1){// Valeurs  à définir
    rc_G = 1;
    }
    if (Correction < 0){// Valeurs  à définir
    rc_G = 0;
    }

    rc_G = Correction;
}
```

Annexe 7 : Schéma de la structure en plexiglas



Annexe 8 : Programmes d'émission et réception

```
#include "mbed.h"

/*
//Emetteur (pour tester)
Serial carte2(PC_10, PC_11);


char chaine[] = "au 100l 150d 255r 125z"; //format des données à transmettre (exemple)
int main()
{
    carte2.baud(19200);

    while(1)
    {
        carte2.printf("%s",chaine);  //transmission
        wait(0.5);                   //pause entre 2 transmission
    }
}
*/

//Receveur

Serial carte1(PC_10, PC_11);
Serial pc(USBTX, USBRX);

    char caractere;             //caractère reçu
    char tab[64];               //message reçu (tableau de caractères)
    int i = 0;                  //compteur de position dans tab
    int debut = 0;              //signale la réception du caractère de début
    int fin = 0;                //signale la réception du caractère de fin

void IT_reception(void);

int main()
{
    pc.baud(115200);
    carte1.baud(19200);
    carte1.attach(&IT_reception, Serial::RxIrq);  //interruption pour executer le programme de réception

    while(1)
    {
    }
}

void IT_reception(void)
{
    caractere = carte1.getc();    //réception d'un caractère

    if(caractere == 'z')          //si c'est le caractère de fin
    {
        fin = 1;
        tab[i] = '\0';
    }
    if( (debut == 1) && (fin == 0) )   //si on a reçu le caractère de début mais pas celui de fin
    {
        tab[i] = caractere;          //remplissage de tab (message)
        i++;
    }
    if(caractere == 'a')          //si c'est le caractère de début
    {
        debut = 1;
    }
    if(fin == 1)          //si la fin est atteinte
    {
        fin = 0;          //réinitialisation des variables
        debut = 0;
        i = 0;
        pc.printf("S = %s\r\n",tab);   //traitement du message (affichage pour test ici)
    }
}
```

Annexe 9 : Code de l'Interface Homme Machine

```
classdef InterfaceVeronica < matlab.apps.AppBase

    % Properties that correspond to app components
    properties (Access = public)
        UIFigure                       matlab.ui.Figure
        GridLayout                     matlab.ui.container.GridLayout
        EXECUTERButton                 matlab.ui.control.Button
        JournalLabel                   matlab.ui.control.Label
        Label                          matlab.ui.control.Label
        ModeDropDown                   matlab.ui.control.DropDown
        ModeDropDownLabel              matlab.ui.control.Label
        AngledegrsTolrance3EditField   matlab.ui.control.NumericEditField
        AngledegrsTolrance3EditFieldLabel  matlab.ui.control.Label
        DistancecmTolrance2cmEditField  matlab.ui.control.NumericEditField
        DistancecmTolrance2cmEditFieldLabel  matlab.ui.control.Label
        PortCOMEditField               matlab.ui.control.NumericEditField
        PortCOMEditFieldLabel          matlab.ui.control.Label
        RobotVeronicaInterfacedeCommunicationLabel  matlab.ui.control.Label
        UIAxes                         matlab.ui.control.UIAxes
        UIAxes2                        matlab.ui.control.UIAxes
    end


    properties (Access = private)
        s % Serial port
        portList % List of the different available com ports
    end

    methods (Access = private)

        function results = rs232Interrupt(app, src, ~)
            rLine = readline(src);
            rLineChar = char(rLine);
            app.PortListLabel.Text = rLineChar;
            drawnow;
            flush(app.s)
        end

    end


    % Callbacks that handle component events
    methods (Access = private)

        % Code that executes after component creation
        function startupFcn(app)
            app.EXECUTERButton.Enable = true;
            app.DistancecmTolrance2cmEditField.Enable = false;
            app.AngledegrsTolrance3EditField.Enable = false;
            app.ModeDropDown.Items = {'Aide','Liste Ports COM','Connecter'};
            app.ModeDropDown.Value = 'Aide';
        end

        % Value changed function: ModeDropDown
        function ModeDropDownValueChanged(app, event)
            value = app.ModeDropDown.Value;
            switch value
                case {'Aide','Liste Ports COM','Connecter'}
                    app.DistancecmTolrance2cmEditField.Enable = false;
                    app.AngledegrsTolrance3EditField.Enable = false;
                case {'Avancer','Reculer'}
                    app.DistancecmTolrance2cmEditField.Enable = true;
                    app.AngledegrsTolrance3EditField.Enable = false;
                case {'Tourner à gauche','Tourner à droite'}
                    app.DistancecmTolrance2cmEditField.Enable = false;
                    app.AngledegrsTolrance3EditField.Enable = true;
            end
            drawnow;
        end

        % Button pushed function: EXECUTERButton
```

```
    function EXECUTERButtonPushed(app, event)
        mode = app.ModeDropDown.Value;
        switch mode
            case 'Aide'
                open InterfaceVeronica.mlx;
            case 'Avancer'
                distance = app.DistancecmTolrance2cmEditField.Value;
                writeline(app.s, strcat('u',string(distance),'s'));
                app.Label.Text = 'Envoi effectué';
                data = read(app.s,1,"uint8");
                app.Label.Text = strcat('Envoi effectué\n',data);
            case 'Reculer'
                distance = app.DistancecmTolrance2cmEditField.Value;
                writeline(app.s, strcat('d',string(distance),'s'));
                app.Label.Text = 'Envoi effectué';
            case 'Tourner à gauche'
                angle = app.AngledegrsTolrance3EditField.Value;
                writeline(app.s, strcat('l',string(angle),'s'));
                app.Label.Text = 'Envoi effectué';
            case 'Tourner à droite'
                angle = app.AngledegrsTolrance3EditField.Value;
                writeline(app.s, strcat('r',string(angle),'s'));
                app.Label.Text = 'Envoi effectué';
            case 'Liste Ports COM'
                app.portList = serialportlist();
                app.Label.Text = app.portList;
            case 'Connecter'
                try
                    app.s = serialport("COM"+app.PortCOMEditField.Value, 115200);
                    configureTerminator(app.s, "CR/LF");
                    % configureCallback(app.s, "byte", 1, @app.rs232Interrupt)
                    configureCallback(app.s, "terminator", @app.rs232Interrupt);
                    app.ModeDropDown.Items = {'Aide','Avancer','Reculer','Tourner à gauche','Tourner à droite'};
                    app.Label.Text = 'Connexion réussie';
                catch
                    warning('Problem using serial.');
                    app.Label.Text = 'Connexion impossible';
                end
        end
        drawnow;
    end
end

% Component initialization
methods (Access = private)

    % Create UIFigure and components
    function createComponents(app)

        % Create UIFigure and hide until all components are created
        app.UIFigure = uifigure('Visible', 'off');
        app.UIFigure.Color = [0.94 0.94 0.94];
        app.UIFigure.Position = [100 100 640 480];
        app.UIFigure.Name = 'MATLAB App';

        % Create GridLayout
        app.GridLayout = uigridlayout(app.UIFigure);
        app.GridLayout.ColumnWidth = {'1x', '1x', '2x'};
        app.GridLayout.RowHeight = {'1x', '1x', '0.5x', '1.5x', '1x', '1x', '1x', '1x', '1x'};

        % Create UIAxes2
        app.UIAxes2 = uiaxes(app.GridLayout);
        title(app.UIAxes2, 'Pression')
        xlabel(app.UIAxes2, 'Distance parcourue (m)')
        ylabel(app.UIAxes2, 'Pression (hPa)')
        app.UIAxes2.Layout.Row = [6 9];
        app.UIAxes2.Layout.Column = 3;

        % Create UIAxes
        app.UIAxes = uiaxes(app.GridLayout);
        title(app.UIAxes, 'Température')
        xlabel(app.UIAxes, 'Distance parcourue (m)')
        ylabel(app.UIAxes, 'T (°C)')
        app.UIAxes.Layout.Row = [2 5];
        app.UIAxes.Layout.Column = 3;
```

```
% Create RobotVeronicaInterfacedeCommunicationLabel
app.RobotVeronicaInterfacedeCommunicationLabel = uilabel(app.GridLayout);
app.RobotVeronicaInterfacedeCommunicationLabel.HorizontalAlignment = 'center';
app.RobotVeronicaInterfacedeCommunicationLabel.FontSize = 22;
app.RobotVeronicaInterfacedeCommunicationLabel.FontColor = [1 0 0];
app.RobotVeronicaInterfacedeCommunicationLabel.Layout.Row = 1;
app.RobotVeronicaInterfacedeCommunicationLabel.Layout.Column = [1 3];
app.RobotVeronicaInterfacedeCommunicationLabel.Text = 'Robot Veronica - Interface de Communication';

% Create PortCOMEditFieldLabel
app.PortCOMEditFieldLabel = uilabel(app.GridLayout);
app.PortCOMEditFieldLabel.HorizontalAlignment = 'right';
app.PortCOMEditFieldLabel.Layout.Row = 2;
app.PortCOMEditFieldLabel.Layout.Column = 1;
app.PortCOMEditFieldLabel.Text = 'Port COM';

% Create PortCOMEditField
app.PortCOMEditField = uieditfield(app.GridLayout, 'numeric');
app.PortCOMEditField.Layout.Row = 2;
app.PortCOMEditField.Layout.Column = 2;

% Create DistancecmTolrance2cmEditFieldLabel
app.DistancecmTolrance2cmEditFieldLabel = uilabel(app.GridLayout);
app.DistancecmTolrance2cmEditFieldLabel.HorizontalAlignment = 'right';
app.DistancecmTolrance2cmEditFieldLabel.Layout.Row = 7;
app.DistancecmTolrance2cmEditFieldLabel.Layout.Column = 1;
app.DistancecmTolrance2cmEditFieldLabel.Text = {'Distance (cm)'; 'Tolérance : 2 cm'};

% Create DistancecmTolrance2cmEditField
app.DistancecmTolrance2cmEditField = uieditfield(app.GridLayout, 'numeric');
app.DistancecmTolrance2cmEditField.Limits = [0 Inf];
app.DistancecmTolrance2cmEditField.Layout.Row = 7;
app.DistancecmTolrance2cmEditField.Layout.Column = 2;

% Create AngledegrsTolrance3EditFieldLabel
app.AngledegrsTolrance3EditFieldLabel = uilabel(app.GridLayout);
app.AngledegrsTolrance3EditFieldLabel.HorizontalAlignment = 'right';
app.AngledegrsTolrance3EditFieldLabel.Layout.Row = 8;
app.AngledegrsTolrance3EditFieldLabel.Layout.Column = 1;
app.AngledegrsTolrance3EditFieldLabel.Text = {'Angle (degrés)'; 'Tolérance : 3°'};

% Create AngledegrsTolrance3EditField
app.AngledegrsTolrance3EditField = uieditfield(app.GridLayout, 'numeric');
app.AngledegrsTolrance3EditField.Limits = [0 180];
app.AngledegrsTolrance3EditField.Layout.Row = 8;
app.AngledegrsTolrance3EditField.Layout.Column = 2;

% Create ModeDropDownLabel
app.ModeDropDownLabel = uilabel(app.GridLayout);
app.ModeDropDownLabel.HorizontalAlignment = 'right';
app.ModeDropDownLabel.Layout.Row = 6;
app.ModeDropDownLabel.Layout.Column = 1;
app.ModeDropDownLabel.Text = 'Mode';

% Create ModeDropDown
app.ModeDropDown = uidropdown(app.GridLayout);
app.ModeDropDown.Items = {'Aide', 'Avancer', 'Reculer', 'Tourner à gauche', 'Tourner à droite', 'Liste Ports COM', 'Connecter'};
app.ModeDropDown.ValueChangedFcn = createCallbackFcn(app, @ModeDropDownValueChanged, true);
app.ModeDropDown.Layout.Row = 6;
app.ModeDropDown.Layout.Column = 2;
app.ModeDropDown.Value = 'Aide';

% Create Label
app.Label = uilabel(app.GridLayout);
app.Label.VerticalAlignment = 'top';
app.Label.Layout.Row = [4 5];
app.Label.Layout.Column = [1 2];
app.Label.Text = '';

% Create JournalLabel
app.JournalLabel = uilabel(app.GridLayout);
app.JournalLabel.HorizontalAlignment = 'center';
app.JournalLabel.FontSize = 16;
app.JournalLabel.Layout.Row = 3;
```

```
            app.JournalLabel.Layout.Column = [1 2];
            app.JournalLabel.Text = 'Journal';

            % Create EXECUTERButton
            app.EXECUTERButton = uibutton(app.GridLayout, 'push');
            app.EXECUTERButton.ButtonPushedFcn = createCallbackFcn(app, @EXECUTERButtonPushed, true);
            app.EXECUTERButton.BackgroundColor = [0.851 0.3255 0.098];
            app.EXECUTERButton.FontSize = 16;
            app.EXECUTERButton.FontWeight = 'bold';
            app.EXECUTERButton.Layout.Row = 9;
            app.EXECUTERButton.Layout.Column = [1 2];
            app.EXECUTERButton.Text = 'EXECUTER';

            % Show the figure after all components are created
            app.UIFigure.Visible = 'on';
        end
    end

    % App creation and deletion
    methods (Access = public)

        % Construct app
        function app = InterfaceVeronica

            % Create UIFigure and components
            createComponents(app)

            % Register the app with App Designer
            registerApp(app, app.UIFigure)

            % Execute the startup function
            runStartupFcn(app, @startupFcn)

            if nargout == 0
                clear app
            end
        end

        % Code that executes before app deletion
        function delete(app)

            % Delete UIFigure when app is deleted
            delete(app.UIFigure)
        end
    end
end
```