République Algérienne Démocratique et populaire
Ministère de l'enseignement supérieur et de la recherche scientifique

Univérsité Abderrahmane Mira de Bejaia
Département Electronique

Mémoire de fin de cycle
En vue de l'obtention du diplôme MASTER
en Électronique
option : Automatique

Thème :

Développement d'applications pour le robot ED-7273

Réalisé par :

ABDOUS Nacer
AGUERCIF Thafsouth

Encadré et dirigé par :
Mr MENDIL Boubkeur

Devant le jury composé de :

Mr GUERMOUZ Loucif
Mr HADDAR Hocine

Béjaia, 2015

REMERCIEMENTS

Nous tenons à remercier Dieu, Le tout puissant par Sa création qui nous a permis d'arriver aussi loin.

Nous adressons nos plus sincères remerciements à notre encadreur Mr B.MENDIL pour sa disponibilité, ses remarques, ses conseils et toute la confiance qu'il a mise en nous.

Nos profonds remerciements à Mr L.GUERMOUZ et à Mr H.HADDAR pour l'honneur qu'ils nous ont fait d'avoir acceptés d'évaluer notre travail.

Nos remerciements vont également à l'ensemble de l'effectif du département de Génie électrique et à l'ensemble de nos enseignants qui ont contribués à notre apprentissage.

Nous tenons à exprimer toutes nos reconnaissances à tous ceux qui ont contribué de prêt ou de loin à la réalisation de ce modeste travail.

Dédicaces

Je dedie ce modeste travail :

A ma mére « Nouara » et à mon pére « Lounes », que Dieu les protège.

A mon frère, sa femme ainsi que leur petite ange « Héloïse »

A ma grand-mére « Megdouda », que Dieu nous la garde.

A tous mes cousins et cousines.

A tous mes oncles et tantes.

A tous mes amis et à toutes mes amies.

A tous ceux qui ont contribué de près ou de loin à l'accomplissement de ce modeste travail.

A mon cher ami et regretté « ZOUGAB Mourad », que Dieu l'accueille dans son vaste paradis.

Sans oublier mes camardes « MALEK Yasmine », « FREDJ Sihame », « AOUCI Katia », « ICHALLAL Azedine », « MOUZAIA Ryadh », « MESROUA Djamel », « YEDJEDD Redouane », « MOUHOUS Juba », « OURDANI Abdelghani », « OURDANI Djoudi » et « OURDANI Abdennour » qui m'ont toujours motivé, encouragé et incité à travailler plus.

ABDOUS Nacer

Dédicaces

Je dédie ce travail :

A la mémoire de mes chers grands parents « Eldjida » «Athmane » et « Md.Amezine », que Dieu les accueille dans son vaste paradis.

A mes très chers parents «Khaled» et « Naima » que j'aime beaucoup, qui ont contribué à ma réussite ainsi que pour leur soutien et leurs encouragements, que Dieu les protège.

A mes chers frères « Amazigh » et « Athmane » et ma petite princesse « Dida »

A mon oncle « Daoud », ainsi que toute sa famille

A mon oncle «Mouaouia »et sa famille

A ma tante « Nassima » et sa famille

A ma grand-mère « Khelidja », ainsi que tous mes tantes et oncles maternels,

A « Na Tounsia », et à toute la famille « AGUERCIF », cousins et cousines

A mon binôme Nacer, ainsi que tous ses amis

A mes amis Kafia, Anna, Sarah, Katia, Aldja, Yasmin, Siham, Lydia, Kamilia et Amina

Toufik, Massi, Azedine, Kiki

Et à tous ceux qui me sont chers.

Thafsouth.A

Table des matières

Liste des tableaux

Liste des abréviations

API :	Application Program Interface
AV :	Avancer
CCW :	Contre Clock-Wise
CMD :	Champ qui désigne la commande dans une trame de communication.
CW :	Clock-Wise
DC-Motor :	Moteur à courant continu
ID :	Identifiant
IR :	Capteur à infrarouge
IRES:	Intelligent Robot Education Studio
LED:	Light-Emitting Diode
MCC :	Microsoft Foundation Classes
MCU :	Micro Controller Unit
MLI :	Modulation Largeur d'Impulsion.
OP :	Amplificateur opérationnel
PC :	Personnel computer
Port COM :	Un port particulier destiné à la communication série
PSD :	Position-sensitive device, qui désigne le capteur à infrarouge.
TD_x :	Tourner à droite avec un angle de x degré.
TG_x :	Tourner à gauche avec un angle de x degré
TR :	Transistor
TTL-IC :	Transistor Transistor Logic- circuit intégré
US :	Capteur à Ultrason (Sensor Ultrasonic)
USB :	Bus Universel de communication Série (Universal Serial Bus)

Introduction générale

Introduction générale

De nos jours, la robotique mobile est impliquée dans plusieurs domaines, tels que : les tâches domestiques (robots aspirateurs), le domaine spatial (le curiosity de la NASA), la navigation dans des milieux hostiles et inconnus, sans oublier les robots utilisés pour la locomotion dans l'industrie.

Pour satisfaire les exigences des hommes, ces robots doivent réaliser leurs tâches (navigation, exploration, cartographie, locomotion) de manière autonome. Pour garantir cette autonomie et interagir avec leur environnement, les robots sont dotés de différents capteurs qui leurs permettent d'avoir une perception de leurs entourages et de leur propre état.

La robotique et le traitement d'information sont étroitement liés, par le fait qu'un robot a besoin de l'acquisition de l'information sensorielle, de la recognition et de la prise de décisions, pour accomplir ses tâches de manière optimale. Par conséquent, la programmation représente une partie cruciale. Généralement, le langage assembleur est plus adapté. Mais, des langages de haut niveau (C, C++, …) sont plus indiqués, vus leurs facilité d'utilisation. Les utilisateurs s'adaptent plus facilement avec une interface graphique qu'avec une console. C'est là où réside l'immense avantage de la programmation orientée objet.

Le but de ce projet est la réalisation d'applications pour le contrôle du robot (ED-7273), sous l'environnement de programmation IRES et avec les MFC (Microsoft Foundation Classes) de Visual C++. Cependant, par mesure de sécurité, plusieurs parties de notre travail ont été d'abord testées sur le kit ED-7271 (intelligent robot trainer kit).

En premier lieu, des applications sous formes de logigrammes IRES ont été développées pour les robots. Puis, le mode de commande (*Script*) a été utilisé ; vue sa commodité et sa flexibilité.

La dernière partie est consacrée au teste de quelques fonctions des bibliothèques fournies avec les robots. Dans ce contexte, la programmation à base des MFC de Visual C++ a été adoptée.

Pour une meilleure présentation de notre travail, le manuscrit a été organisé en quatre chapitres. Dans les deux premiers chapitres, on a introduit les notions de base de la robotique mobile et on a exposé les différents modules des robots ED-7271 et ED-7273, qu'on va utiliser. Au troisième chapitre, on a développé différents programmes sous l'environnement *IRES* pour le contrôle des capteurs et des moteurs. Enfin, dans le dernier chapitre, on a programmé des applications à base des MFC sous le Visual C++, en exploitant les bibliothèques fournies avec les robots.

Chapitre I : Généralités sur la robotique mobile

I.1. Introduction

La robotique est un domaine scientifique vaste et pluridisciplinaire, partant des aspects mécaniques, passant par l'automatique et allant jusqu'aux aspects de plus haut-niveau tels que la perception, la modélisation de l'environnement et la décision.

La relation entre l'homme et le robot a toujours été ambiguë, l'homme crée des robots de plus en plus évolués et autonomes pour se soulager des tâches fastidieuses, répétitives ou dangereuses, et ne cesse de fantasmer sur les facultés de sa création, à le surpasser, voire à se révolter contre lui. Un robot intelligent est celui qui a la capacité de perception des changements dans son environnement, de la cognition de la situation par lui-même, et d'exercer une mobilité autonome et une manipulation ou même une interaction avec les humains. Pour cela, il a besoin des structures mécaniques et électriques, une variété de capteurs, la reconnaissance vocale, la reconnaissance d'objets, la détection de niveau, et la détection d'accélération, pour reconnaître la situation de ses périphériques et de l'environnement externe, c'est-à-dire il a besoin d'imiter les cinq sens humains **[Man 71].**

I.2. Historique

Le terme *robot* est employé pour la première fois en 1920, par l'écrivain tchécoslovaque Karel Capek dans la pièce de théâtre RUR (Rossum's Universal Robots). Il vient du tchèque « ROBOTA », qui signifie travail pénible ou corvée, dont les robots sont des serviteurs dociles et efficaces pour réaliser les taches dangereuses. Capables de penser, les androïdes créés par le scientifique Rossum sont des travailleurs artificiels et véritables machines qui finissent par se révolter contre leurs créateurs.

- Le premier robot humanoïde est probablement dû à Léonard De Vinci, qui présenta en 1495 un chevalier en armure capable de s'asseoir, relever sa visière et bouger ses bras. Après avoir découvert ses notes et schémas en 1950, une réplique fonctionnelle fut construite et exposée au musée de Berlin.

- Le XVIIIe siècle fut très riche en innovations. Un automate très évolué fut présenté par Jacques de Vaucanson en 1738, présentant un canard capable d'ingurgiter (manger) de véritables aliments et la refouler après ingestion de cette dernière.

- Isaac Asimov, le célèbre auteur de science-fiction, est le premier qui a utilisé le mot *robotique* en 1941, dans des différents livres où apparaissent des robots, c'est lui qui avait fixé les fameuses trois lois de la robotique pour protéger l'être humain.

- Première loi : un robot ne peut porter atteinte à un être humain, ni rester passif et permettre qu'un être humain soit exposé au danger.

- Deuxième loi : un robot doit obéir aux ordres donnés par l'être humain, sauf si de tels ordres entrent en conflit avec la première loi.
- Troisième loi : un robot doit se protéger longtemps tant que cette protection n'entre pas en conflit avec la première ou la deuxième loi.

- Dans les années 1950 (en 1948), l'américain W. Grey Walter a construit les premiers robots mobiles autonomes. Pouvant interagir avec leur environnement. Il s'agissait de deux tortues mécaniques, *Elsie* et *Elmer*, dotées de capteurs de lumière et de contact qui leur permettaient de se guider et d'éviter les obstacles.

- En 1961, le premier robot industriel, *Unimate,* fut créée par George Devo.

- En 1966, à l'université de Caroline du Sud, est conçue *PhoneyPony,* une table à quatre pieds dont les mouvements sont contrôlés par ordinateur.

- En 1973, au Japon, à l'université de Waseda, le premier robot humanoïde à marche bipède avancée, *Wabot*, est réalisé. La machine est dotée de la parole, peut évaluer les distances et la direction d'objets en mouvement et les attraper.

Depuis, de nombreux robots plus sophistiqués ont vu le jour. D'ailleurs en 1999 le célèbre robot chien de compagnie *Aïbo* fit son apparition comme l'androïde *Asimov* capable de reconnaître des visages, de monter et descendre des escaliers, de comprendre la parole humaine et d'analyser son environnement. En 2000, Un robot humanoïde *Asimo* fut créé par Honda au Japon, en tant que robot marcheur bipède autonome, il attira l'attention mondiale. Et en 2003, c'est le tour des rovers *Spirit* et *Opportunity* d'explorer Mars pour la NASA.

Récemment, la robotique a évoluée vers des robots autonomes ayant des aptitudes de plus en plus proches de celles des humains, à telle point qu'ils envahissent notre quotidien : robot-jouet, robot-tondeuse, robot-aspirateur, robot explorateur, etc **[Web 1].**

Le nouveau type de la robotique est appelé la robotique avancée (Advanced Robotics). Généralement, c'est là où on introduit la notion de la mobilité du robot **[Bel 11].**

On peut dire qu'il y a deux catégories de robots selon le critère de mobilité:

- Les robots manipulateurs constitués par deux sous-ensembles distincts : une structure mécanique articulée et un organe terminal.

- Les robots mobiles sur quoi doit se reposer l'essentiel de notre travail.

La suite de ce chapitre est consacrée à la définition des composants principaux constituant un robot mobile en particuliers les capteurs et les actionneurs **[Bel 11].**

I.3. La locomotion

Aujourd'hui, l'utilisation des robots mobiles est très envisagée pour l'automatisation de différentes tâches. D'ailleurs, on les trouve dans plusieurs secteurs: robots-compagnons assistant les personnes à domicile ou en charge de la surveillance et des soins, robots industriels pour la réalisation de gestes pénibles et répétitifs, des tâches impliquant une grande précision, ou encore le transport dans les ateliers automatisés, l'agriculture, l'exploitation des mines, l'exploration planétaire et l'exploration de milieux hostiles. Un robot mobile a donc besoin de mécanismes de locomotion pour lui permettre de se déplacer dans des environnements différents, qu'ils soient terrestre, sous-marins ou aériens. La locomotion présente deux types de contraintes :

- Les contraintes cinématiques, qui portent sur la géométrie des déplacements possibles du robot dans l'environnement de travail.

- Les contraintes dynamiques, liées aux effets du mouvement (accélérations, vitesses bornées, présence de forces d'inertie ou de frottement). Tous ces facteurs influent sur le mouvement exécuté **[Che 14].**

On va se focaliser sur la locomotion terrestre, qui présente différents types parmi lesquels on cite :

I.3.1. Les mobiles à roues

Simple à concevoir, actuellement, c'est la structure mécanique la plus appliquée. Cette technique assure, selon la disposition et les dimensions des roues, un déplacement dans toutes les directions avec une accélération et une vitesse importante. Le franchissement d'obstacles ou l'escalade de marches d'escalier est possible dans une certaine mesure. Toutes les configurations (nombre, agencement, fonction) des roues sont appliquées.

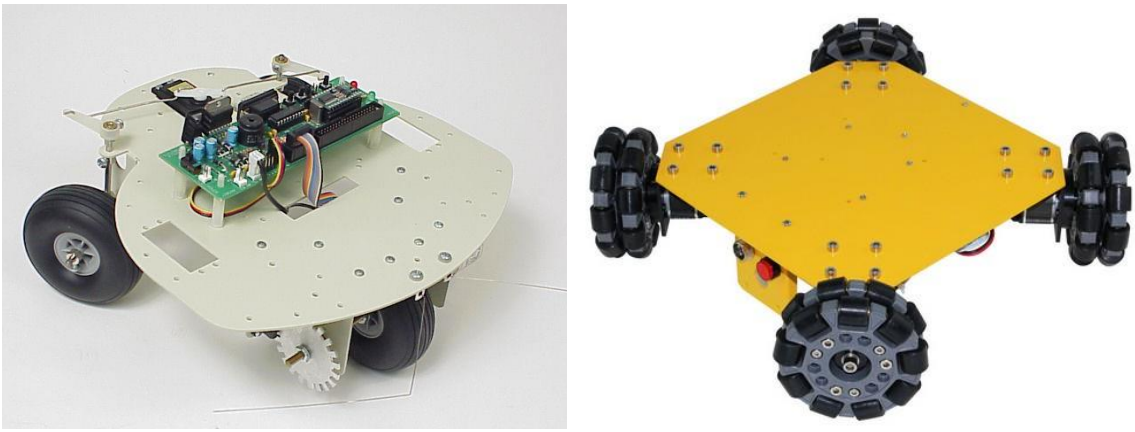


Figure I.1. *Robot à roues*

I.3.2. Les mobiles à chenilles

L'utilisation des chenilles présente l'avantage d'une bonne adhérence au sol et d'une faculté de franchissement d'obstacles. Avec une surface de contact entre le mobile et le sol, l'adhérence est plus importante, alors leur utilisation est sur terrain accidenté ou de mauvaise qualité concernant l'adhérence (présence de boue, herbe,…). Avec les robots à roues, ils sont les plus répandus. Ils sont faciles à commander grâce au nombre limité de degrés de liberté (le plus souvent deux) **[Sli 05].**

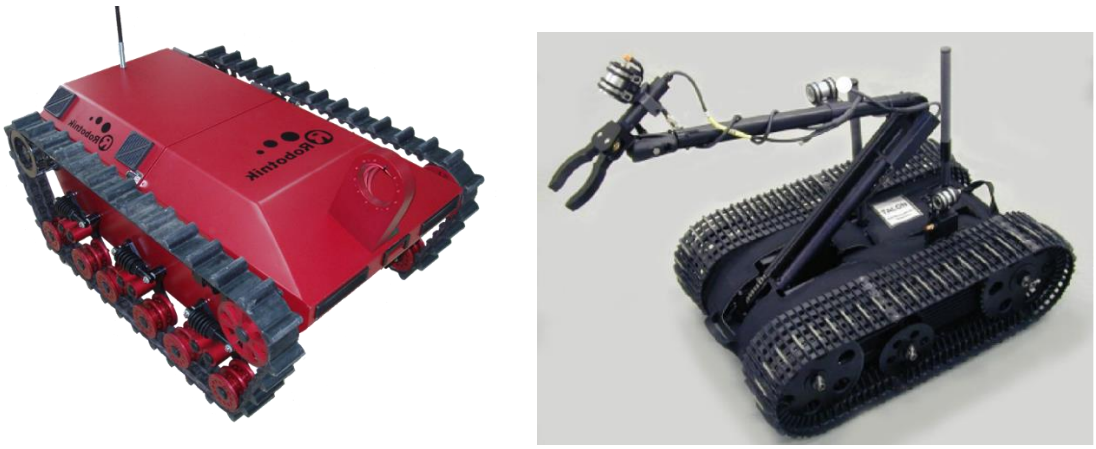


Figure I.2. *Robots à chenilles.*

I.3.3. Les mobiles marcheurs

S'inspirant des travaux du biologiste de Berkeley Robert J. Full sur la locomotion des animaux dotés de nombreuses pattes (crabes, fourmis, cafards, mille-pattes, geckos) **[Web 2]**, les chercheurs se sont orientés vers des systèmes de locomotion à pattes. Avec de tels systèmes, dont le contact avec le sol est discret, les robots mobiles marcheurs sont utilisés pour franchir les obstacles ou sur des sols dont l'utilisation des roues ne serait pas possible. Ces derniers sont même capables de monter des marches, traverser des lacunes (trous) et de marcher sur des terrains extrêmement rugueux.

Leur anatomie à nombreux degrés de liberté permet un rapprochement avec les robots manipulateurs. La locomotion est commandée en termes de coordonnées articulaires. Les méthodes de commande des articulations définissent le concept d'allure qui assure le déplacement stable de l'ensemble **[Sli 05].**

Figure I.3. *Robot marcheurs à pates*

I.3.4. Les robots rampants

Le robot rampant est capable de se déplacer sur le sol avec un seul et unique muscle pneumatique. Ce muscle ne travaille seulement que dans un axe. Donc, la forme du rampant devait réussir à transférer cette énergie mécanique de telle façon à ce que les pattes du robot puissent s'appuyer sur le sol efficacement. La forme la plus simple aurait été une banale articulation associée au vérin pneumatique. Pour des questions de stabilité et de simplicité, il est préférable que le robot se déplace sur au moins trois pattes. Le rampant possède donc une patte postérieure qui le pousse vers l'avant, et deux pattes supérieures qui le tractent. Ces deux dernières sont commandées par le vérin par l'intermédiaire d'un mécanisme qui transforme le mouvement du vérin en mouvement recherché.

La reptation est une solution de locomotion pour un environnement de type tunnel qui conduit à réaliser des structures filiformes. Le système est composé d'un ensemble de modules ayant chacun plusieurs mobilités. Les techniques utilisées découlent des méthodes de locomotion des reptiles.

- Le type scolopendre constitue une structure inextensible articulée selon deux axes orthogonaux.
- Le type lombric comprend trois articulations, deux rotations orthogonales et une translation dans le sens du mouvement principal.
- Le type péristaltique consiste à réaliser un déplacement relatif d'un module par rapport aux voisins **[Sli 05].**

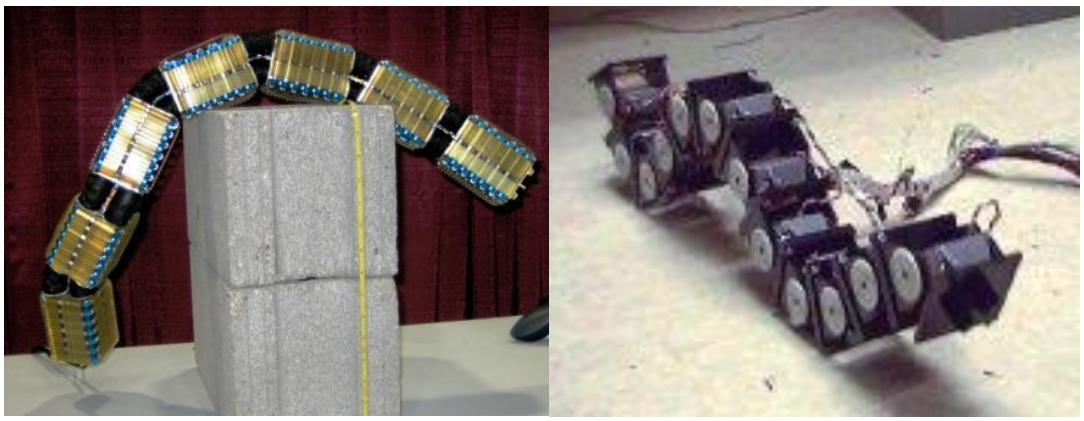


Figure I.4. *Robots rampants.*

Néanmoins, en plus des systèmes de locomotion précédemment cités, des systèmes hybrides, avec des roues et des pattes ou encore des chenilles et des pattes, sont utilisés combinant ainsi leurs avantages et malheureusement leurs complexités de mise en œuvre aussi.

Figure I.5. *Robots hybrides*.

I.4. Les robots à roues

La variété dans les types de locomotion, dans un système de robotique mobile, dans le cas de robot à roue, est due à l'un de ces éléments :

1 - Le système de propulsion : qui vient souvent d'une simple rotation des roues. La propulsion peut également provenir de l'actionnement d'articulations internes du châssis.

2 - Le système de direction : tels que :

- *Rotation par vitesses différentielles* qui consiste à imposer un différentiel de vitesse sur les roues.
- *Rotation à châssis articulé* qui contient une ou des parties orientables.

- *Rotation avec roues directrices indépendantes* qui offre de multiples possibilités de rotation et permettent des déplacements singuliers comme le déplacement en crabe.

Enfin Certaines plates-formes disposent de systèmes de direction hybrides, permettant ainsi de choisir le système le mieux adapté à la situation courante, voire d'en combiner plusieurs pour augmenter les possibilités de négocier une difficulté.

3- Le système de suspension : permet d'augmenter ses capacités de franchissement et sa stabilité en garantissant le maintien du contact des roues avec le sol. Ces suspensions peuvent être passives ou actives. Dans ce dernier cas, il est possible de développer diverses lois de commande tirant parti de ces suspensions actives pour améliorer les performances de locomotion du système (en optimisant des critères tels que la posture de la plate-forme, par exemple) **[Pey 06].**

I.4.1. Les types de robots à roues

Il y a plusieurs types de robots mobiles à roues parmi lesquels nous citons :

I.4.1.1. Robot Unicycle

C'est un robot qui fonctionne avec deux roues indépendantes et possédant un certain nombre de roues libres assurant sa stabilité, mais ces dernières n'interviennent pas dans la cinématique **[Zid 09]**. Ce type de robots est très répandu, en raison de sa simplicité de construction et de ses propriétés cinématiques intéressantes.

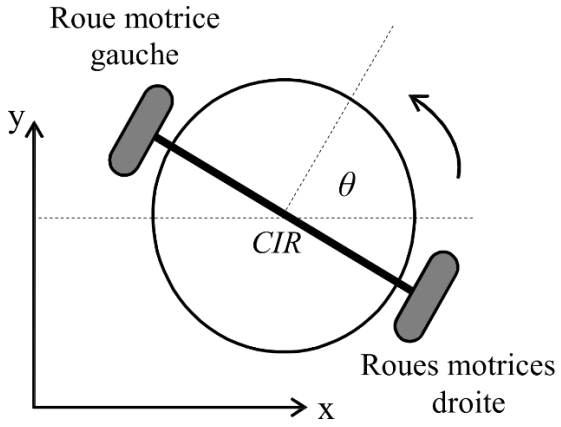


Figure I.6. *Robot Unicycle.*

I.4.1.2. Robot tricycle

Ce robot est constitué de deux roues fixes de même axe et d'une roue centrée orientable placée sur l'axe longitudinal du robot. Le mouvement est conféré au robot par deux actions : la vitesse

longitudinale et la direction de la roue orientable. De ce point de vue, il est donc très proche d'une voiture.

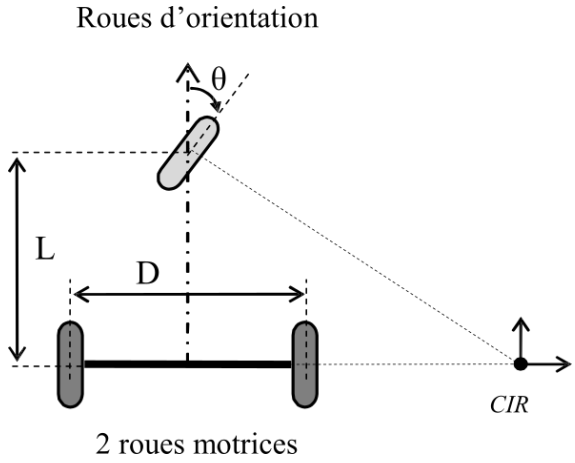


Figure I.7. *Robot Tricycle*

I.4.1.3. Le robot de type voiture

Composé d'un train moteur à l'arrière, du corps principal, et de roues de direction à l'avant. On introduit la notion de roue directrice centrale. Cette roue, virtuelle dans le cas d'une voiture, correspondrait à la roue directrice d'un tricycle équivalent. Son introduction permet de simplifier les équations en faisant abstraction du mécanisme de couplage des roues directrices servant à respecter les contraintes de roulement sans glissement et en ne considérant alors qu'un seul angle de direction.

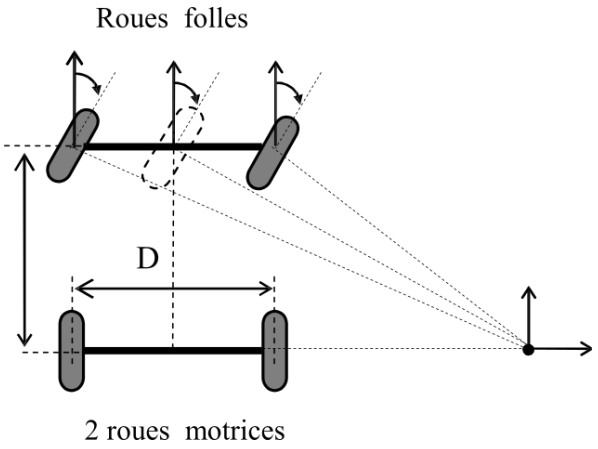


Figure I.8. *Robot type Voiture.*

I.4.1.4. Un robot mobile omnidirectionnel

On peut agir indépendamment sur les vitesses : vitesse de translation selon les axes x et y et vitesse de rotation autour de z. D'un point de vue cinématique, on montre que cela n'est pas possible avec des roues fixes ou des roues centrées orientables. On peut en revanche réaliser un robot omnidirectionnel en ayant recours à un ensemble de trois roues décentrées orientables ou de trois roues suédoises disposées aux sommets d'un triangle équilatéral **[Bea 01].**

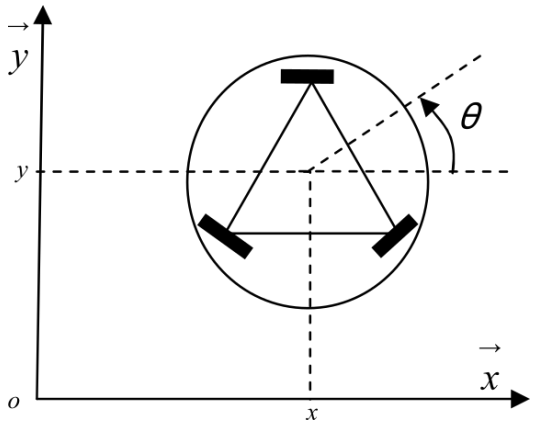


Figure I.9. *Robot Omnidirectionnel.*

I.5. La perception

Pour être capable d'améliorer ses propres performances par apprentissage ou d'agir dans un environnement ouvert ou confiné dynamique et imparfaitement modélisé ou même inconnu, un robot doit posséder des capacités de plus haut-niveau telles que : la perception, la modélisation de l'environnement et la prise de décision. La perception joue un rôle primordial en robotique mobile. Elle est relative à la capacité du système à recueillir, traiter et mettre en forme des informations utiles au robot pour agir et réagir dans son environnement. Les systèmes actuels privilégient des capteurs fournissant des informations de distances, comme des systèmes radars ou des détecteurs laser ou encore à ultrasons, du fait de la facilité à traiter les informations reçues.

I.5.1. Les capteurs

On ne peut parler de perception sans citer la matière première de tout système de perception que sont les capteurs. Dans cette partie on va les aborder sans trop de détails, puisque on en parlera encore dans le chapitre II « Présentation des robots ED7271/ED7273 et de l'environnement IRES ».

Dans la robotique mobile, les capteurs sont usuellement classés en deux catégories. Les capteurs proprioceptifs qui renseignent sur l'état propre du robot (capteurs de position ou de vitesse

des roues, capteurs de charge de la batterie). Les capteurs extéroceptifs qui informent le robot sur l'état de l'environnement où il évolue (capteurs de température ou du taux d'humidité environnant, capteurs de distance à infrarouge et à ultrason).

I.5.2. Les systèmes de perception

La fonction perception consiste globalement à saisir un certain nombre d'informations sensorielles dans le but d'acquérir une connaissance et une compréhension du milieu de l'évolution.

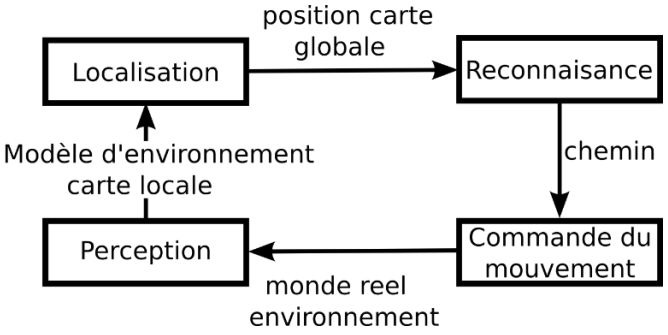


Figure I.10. *Schéma de perception*

Plusieurs constats peuvent être faits sur cet organe essentiel de la chaîne fonctionnelle de la navigation. Le premier concerne le choix d'un système de perception qui est souvent dépendant du milieu d'évolution du robot mobile ainsi que des fonctionnalités dont dispose ce dernier pour remplir sa mission.

Le second se révèle être générique et consiste à affirmer qu'un système de perception constitué d'un unique capteur sera rarement suffisant pour percevoir correctement l'environnement. Le système de perception d'un robot mobile intégrera le plus souvent plusieurs capteurs qui seront de types complémentaires pour un enrichissement des informations sensorielles, ou de types redondants pour répondre au problème de fonctionnement en mode dégradé. Dans ce cadre, des méthodes de fusion de données seront généralement employées pour conditionner ces informations sensorielles.

Enfin, le troisième constat c'est sur coût de l'intégration de capteurs sur le véhicule autonome. La précision désirée et une fréquence d'acquisition élevée seront autant de facteurs qui augmenteront le coût d'un capteur. Il s'agit donc là d'une contrainte qui pèsera inévitablement sur le choix d'un système de perception **[Sli 05].**

I.6. La cartographie

Un robot a besoin de connaitre sa position pour pouvoir cartographier son environnement. D'un autre côté, il doit absolument disposer d'une carte préétablie de son environnement pour pouvoir s'y localiser. Ces deux tâches sont liées. Lorsqu'on ne dispose pas d'une carte de l'environnement, ni

de données de localisation, le robot doit trouver ces informations simultanément, on parle alors de SLAM **[Ham 12].**

SLAM (Simultaneous Localization And Mapping) est l'un des problèmes les plus complexes et fondamentaux dans la robotique mobile. Sa complexité vient du fait que pour construire une bonne carte, le robot doit connaitre précisément sa position, mais pour connaitre sa position il a besoin d'une carte précise. C'est cette corrélation entre la position globale et la carte qui rend le problème difficile et qui exige de chercher une solution dans une dimension encore plus grande **[Ana 12].**

I.7. La localisation

Il existe trois méthodes de localisation, relative, absolue et mixte.

Les méthodes de localisation relative sont centrées sur un aspect local de positionnement. Elles consistent à prendre en considération l'ensemble des déplacements effectués par le mobile relativement à un point de départ ou à sa position initiale. Aucune référence à l'environnement extérieur n'est utilisée dans ces méthodes.

Celles de localisation absolue utilisent l'information provenant de balises ou de points de repères ayant des coordonnées connues et fixes. Il est donc possible de calculer la position à partir des distances mesurées entre le robot et les balises. La notion d'absolue est d'avoir accès aux paramètres qui définissent la position du robot dans l'univers, d'un point de vue pratique, on doit se limiter à la position sur terre.

Les autres méthodes de localisation dites mixtes consistent à combiner la localisation relative et absolue. Par défaut, la localisation est faite de façon relative jusqu'à ce qu'une balise ou un point de repère, ayant une position connue, soit identifiée. La position du robot peut alors être réinitialisée par rapport à des points absolus de l'environnement, diminuant les erreurs liées à la localisation relative dans la mesure où les repères sont uniques et précis **[Clo 07].**

I.7.1. Localisation relative par odométrie

La technique de l'odométrie est la plus utilisée pour la localisation des robots mobiles à roues. Elle permet de déterminer la position et l'orientation d'un véhicule par intégration de ses déplacements élémentaires, et ce, par rapport à un repère lié à sa configuration initiale **[Sli 05].**

La mesure de rotation est en général effectuée par un codeur optique disposé sur l'axe de la roue, ou sur le système de transmission. Le problème majeur de cette mesure est que l'estimation du déplacement fournie dépend très fortement de la qualité du contact entre la roue et le sol. Elle peut être relativement correcte dans le cas de roulement sans glissement. Mais, elle est en général quasiment inutilisable seule pour un robot à chenille par exemple. Pour limiter ce problème, il peut

être intéressant de positionner le codeur optique sur une roue non motrice qui glissera moins. Notons cependant que l'erreur de ces méthodes se retrouve en général principalement sur l'estimation de la direction du robot, tandis que la mesure de la distance parcourue est souvent de meilleure qualité **[Dav13].**

I.7.2. La Télémétrie

Quelle que soit la technologie utilisée pour effectuer la mesure télémétrique, le capteur renvoie généralement deux informations. La première donne l'angle de gisement, c'est-à-dire la direction dans laquelle a été faite la mesure. La seconde donne la distance au corps ayant réfléchi l'onde émise. Cette technique de mesure permet donc de positionner les objets présents dans la scène par rapport au robot. Elle se prête très bien aux environnements d'intérieur, structurés, comportant des formes régulières et statiques comme des murs, qui par ailleurs, possèdent généralement de bonnes propriétés de réflexion **[Bay 10].**

I.7.3. Localisation absolue par GPS (Localisation par satellites)

Il existe très peu de systèmes donnant la position absolue d'un point dans un repère fixe. Le *GPS* (Global Positioning System) est initialement développé pour des applications militaires américaines. Il est actuellement à la disposition du public pour des applications civiles. Le GPS fonctionne avec un ensemble de satellites qui effectuent des émissions synchronisées dans le temps. Par recoupement des instants d'arrivée des signaux et de la position des satellites émetteurs, les récepteurs peuvent calculer leur position. Le principe de calcul de la position est basé sur la triangulation **[Bay 10]**. Les temps mis en jeux sont mesurés en nanosecondes. Chaque satellite possède, à bord, une horloge atomique. Mais, le récepteur n'est équipé que d'un simple quartz. Ainsi, un quatrième satellite est souvent utilisé pour corriger le temps **[Men 14].**

Ce système fonctionne en effet difficilement dans des environnements urbains, et n'est pas utilisable à l'intérieur des bâtiments. Sa précision est de plus souvent trop faible pour qu'un robot terrestre puisse utiliser ces informations seules. En pratique, il est souvent couplé à un système inertiel qui permet de palier aux pertes du signal GPS **[Fil 13].**

I.7.4. Localisations absolues par SIG

Le SIG (Système d'Informations Géographique) est un système de localisation absolue basé sur une base de données d'amers géo-référencés. En observant ces amers à travers un capteur extéroceptif (comme une caméra par exemple) et en utilisant des méthodes de triangulation, nous pouvons remonter à une localisation absolue de notre robot. Si le référencement absolu de la localisation d'un véhicule a un intérêt indéniable dans la gestion des objectifs de sa mission (points de passage, couloir de navigation, etc.), il comporte des limites dues à l'absence d'information locale susceptible de changements incompatibles avec une gestion globale de la navigation **[Mal 11].**

I.8. La navigation

Pour un robot mobile, la navigation est une tâche qui consiste à réaliser un mouvement libre dans l'espace des configurations (environnement de travail) sans collisions avec les obstacles. La navigation est usuellement définie comme le guidage des véhicules d'un point à un autre. En robotique mobile, le sens du terme est élargi à l'ensemble des processus qui permettent au robot mobile de déterminer et d'exécuter les déplacements nécessaires pour atteindre une cible donnée **[Web 3].**

I.8.1. Catégories de navigation

On définit deux catégories de navigation, la navigation autonome et la navigation visuelle.

- **Navigation autonome**

La robotique mobile autonome vise à concevoir des systèmes capables de se déplacer de façon autonome. Les applications directes se situent notamment dans les domaines de l'automobile, de l'exploration planétaire ou de la robotique de service par exemple. De nombreuses applications restent à découvrir, qui ne découlent pas directement des avancées de la robotique mais qui utilisent ses méthodes et ses développements **[Lef 06**].

- **Navigation visuelle**

Sous le vocable navigation visuelle, on désigne toutes les fonctions qu'un système autonome doit exécuter pour planifier et exécuter ses déplacements dans un environnement quelconque, inconnu à priori, perçu par une ou plusieurs caméras embarquées. Parmi les fonctions que le robot mobile doit exécuter, citons:

- l'acquisition d'une image et l'extraction d'une description 2D de la scène courante. Il s'agit d'une fonction d'interprétation qui identifie dans l'image, les entités de l'environnement utiles pour la navigation d'un robot terrestre: zones de terrain navigable, obstacles, chemins, amer, etc.
- la construction d'un modèle de l'environnement par fusion des descriptions acquises depuis les positions successives du robot mobile.
- la planification d'une trajectoire requise par une tâche que doit exécuter le robot mobile, typiquement dans le contexte *«aller vers la cible tout en évitant les obstacles»* par exemple, cette trajectoire est générée à partir du modèle de l'environnement, acquis au préalable par le robot mobile, ou introduit par un opérateur dans la base de connaissance du système.
- enfin, l'exécution de trajectoires par séquencement de mouvements élémentaires contrôlés à partir des images **[Avi 05].**

I.8.2. Les stratégies de navigation

Il existe diverses stratégies de navigation qui permettent à un robot mobile de se déplacer pour rejoindre un but, de même que les classifications qui peuvent en être faites. Nous reprenons ici une classification établie par Trullier et Meyer, qui présente l'avantage de distinguer les stratégies sans modèles internes et les stratégies avec modèle interne, selon la hiérarchie illustrée par la ***Figure I.11***.

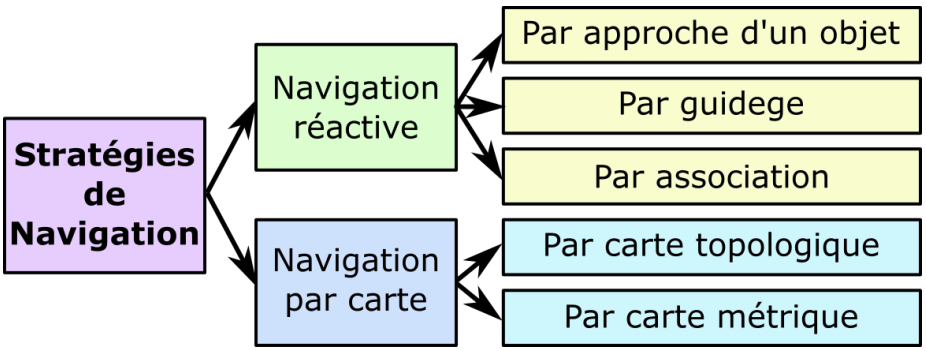


Figure I.11. *Stratégies de navigation*

I.8.2.1. La navigation réactive

Les algorithmes de modélisation de l'environnement sont très coûteux en temps de calcul. En plus, le robot n'a souvent qu'une vue très partielle de son environnement. Ceci implique que le robot peut remettre en cause son plan, en raison de présence imprévue d'obstacles par exemple, entraînant une nouvelle modélisation coûteuse de l'environnement et une régénération de chemin. De plus, le délai entre l'apparition d'un nouvel obstacle, sa prise en compte dans le modèle et la régénération d'un nouveau plan pouvant être important, il existe un risque de collision pour le robot si une réponse immédiate est nécessaire. Pour cela les roboticiens ont cherché à remplacer ou compléter la navigation à base de carte par des processus rapides qui se regroupent sous le terme de navigation réactive. La navigation réactive utilise des actions réflexes pour guider le robot et se différencient essentiellement par le type de perceptions utilisées pour déclencher ces actions. Les comportements de ce type restent essentiels dans les robots modernes. Car, du fait de leur simplicité, ils sont généralement exécutés très rapidement et permettent de réaliser des tâches de bas niveau, comme l'évitement des obstacles imprévus, essentielles à la sécurité d'un robot. On peut citer trois sous classe :

a. Navigation par approche d'un objet

Cette capacité de base permet de se diriger vers un objet visible depuis la position courante du robot. Elle est en général réalisée par une remontée sur la perception de l'objet, comme dans l'exemple célèbre des véhicules de Valentino Braitenberg qui utilisent deux capteurs de lumière pour atteindre ou fuir une source lumineuse. Cette stratégie utilise des actions réflexes, dans lesquelles

chaque perception est directement associée à une action. C'est une stratégie locale, c'est-à-dire fonctionnelle uniquement dans la zone de l'environnement pour laquelle le but est visible **[Dav 13]**.

b. Navigation par Guidage.

Elle permet d'atteindre un but qui n'est pas un objet matériel directement visible, mais un point de l'espace caractérisé par la configuration spatiale d'un ensemble d'objets remarquables, ou amers, qui l'entourent ou qui en sont voisins. La stratégie de navigation consiste alors à se diriger dans la direction qui permet de reproduire cette configuration. Le robot tente donc de mémoriser une "photo" de l'ensemble de ses perceptions, lorsqu'il se trouve dans une situation but et cherche à faire correspondre ses perceptions courantes avec la situation mémorisée. Cette stratégie de navigation requiert néanmoins la perception directe des amers caractéristiques de la situation but et dès lorsqu'ils ne sont plus perçus, la stratégie échoue et ne peut donc être utilisée que dans un contexte local. Cette capacité semble être utilisée par certains insectes, comme les abeilles. Elle a été utilisée sur divers robots. Cette stratégie utilise également des actions réflexes et réalise une navigation locale qui requiert que les amers caractérisant le but soient visibles. Enfin, il est à noter que ce type de navigation ne requiert aucune modélisation spatiale. En effet, la mise en correspondance d'une mémoire perceptive avec les perceptions courantes ne requiert en aucun cas un traitement spatial spécifique **[Dav 13]**.

c. Navigation par association

C'est une stratégie permettant de rejoindre un but ou des amers, caractérisant son emplacement, à partir des positions pour lesquels ces derniers sont invisibles. On doit associer une action à effectuer à chacun des lieux pour lesquels les perceptions restent similaires. L'enchaînement de ses actions associées définit une route vers la cible. Même si ce type de navigation est plus robuste que celui par guidage, il ne permet pas l'établissement de stratégie quand les informations mémorisées dépendent directement du but à rallier. Pour atteindre un autre but, le robot doit posséder une autre mémorisation le ralliant vers ce dernier. Cette méthode permet donc une autonomie plus importante mais est limité à un but fixé **[Dav 13]**.

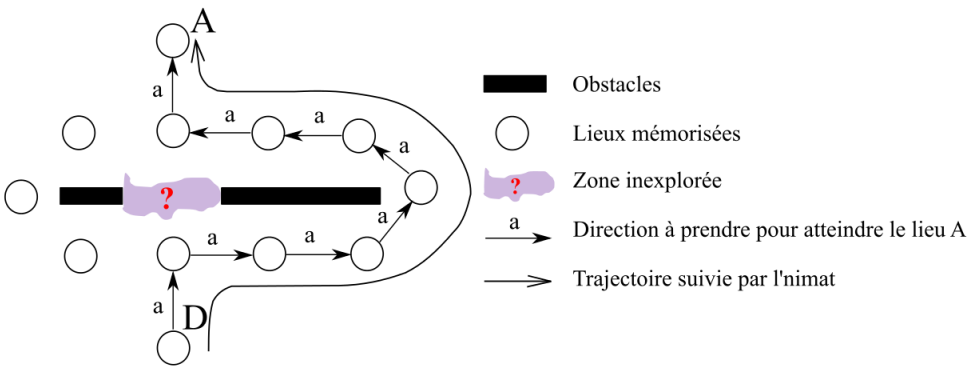


Figure I.12. *Navigation par association*

En chaque lieu, représenté par un cercle, l'action à accomplir pour rejoindre le but A est représentée par une flèche indiquant la direction à suivre à partir de ce lieu. Ce qui permet de rejoindre le but fixé en suivant des chemins prédéfinis. Dans cet exemple, le chemin joignant le lieu D au lieu A et passant par la droite de l'obstacle a été appris. Rejoindre le lieu A depuis le lieu D ne pourra alors être réalisé que par ce chemin. Le passage par la gauche est inconnu ce qui le rend inutilisable **[Dav 13]**.

I.8.2.2. La navigation par carte

Elle s'appuie sur un modèle interne du monde (une carte) qui supporte une planification. Ce modèle interne mémorise la structure spatiale de l'environnement, indépendamment d'un but précis. Chacune des positions mémorisées dans ce modèle interne peut alors être utilisée comme but par le processus de planification dont le rôle est de calculer une route vers ce but. Le principe de la navigation par carte est basé sur la cartographie, la localisation et la planification. On peut citer deux sous classe **[Dav 13]**.

a. Navigation par carte topologique

Une carte topologique est un graphe, dans lequel chaque nœud correspond à un endroit caractéristique (carrefour entre couloirs, entrées dans les espaces ouverts ...), appelé meeting point dans plusieurs travaux, ou encore lieu. Chaque lieu devra être décrit par un ensemble de caractéristiques propres qui permettront au robot de le reconnaître. Une arête liant deux nœuds signifie qu'il existe une commande référencée capteur que le robot peut exécuter afin de se déplacer entre les deux lieux correspondants. Cette représentation est particulièrement adaptée pour décrire des réseaux de couloirs : les lieux correspondent aux carrefours, les arêtes aux couloirs qui les lient. Pour la génération d'une trajectoire entre deux lieux définis dans une carte topologique, il suffit de rechercher un chemin dans le graphe. Dans cette stratégie, le robot n'a nul besoin de connaître sa position précise par rapport à un repère du monde. Il doit seulement déterminer une localisation qualitative « *Je suis dans le lieu A* » et, au mieux, une estimation de sa position relative par rapport à un repère lié au lieu dans lequel il se trouve **[Dav 13]**.

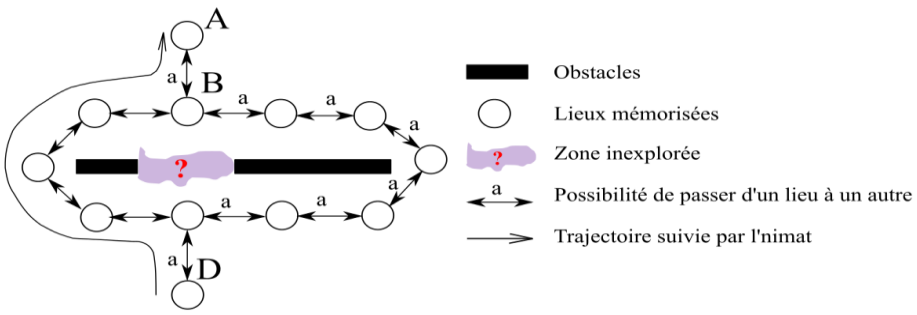


Figure I.13. *Navigation topologique*

Cette technique permet de mémoriser un ensemble de nœuds et les possibilités de passer de l'un à l'autre. Uniquement parmi les lieux et les chemins déjà connus, une étape de planification permet de rechercher le chemin optimum vers la cible. Dans notre exemple, le chemin le plus court entre D et A peut alors être calculé et l'obstacle contourné par la gauche, mais le chemin traversant en ligne droite de D à A n'est pas considéré **[Dav 13]**.

b. Navigation par carte métrique

La navigation métrique exploite une représentation géométrique du monde, qui peut être donné par un utilisateur, ou construite par le robot lui-même. Elle représente une extension de la précédente. Car, elle permet au robot de planifier des chemins au sein de zones inexplorées de son environnement. Elle mémorise pour cela les positions métriques relatives des différents lieux, en plus de la possibilité de passer de l'un à l'autre. Ces positions relatives permettent, par simple composition de vecteurs, de calculer une trajectoire allant d'un lieu à un autre, même si la possibilité de ce déplacement n'a pas été mémorisée sous forme d'un lien.

L'équivalent mathématique de la carte métrique, est un espace vectoriel muni d'une distance, où chaque vecteur AB exprime le chemin entre un lieu A et un lieu B en tenant compte de la distance et de l'orientation. Les détours et raccourcis métriques peuvent alors être manipulés via les opérations vectorielles telles que la norme (estimation des distances), l'addition ou la soustraction vectorielle (construction de nouveaux chemins). Il n'est plus requis d'avoir emprunté un chemin pour pouvoir l'utiliser, et c'est bien là la différence fondamentale avec la carte topologique **[Dav 13]**.

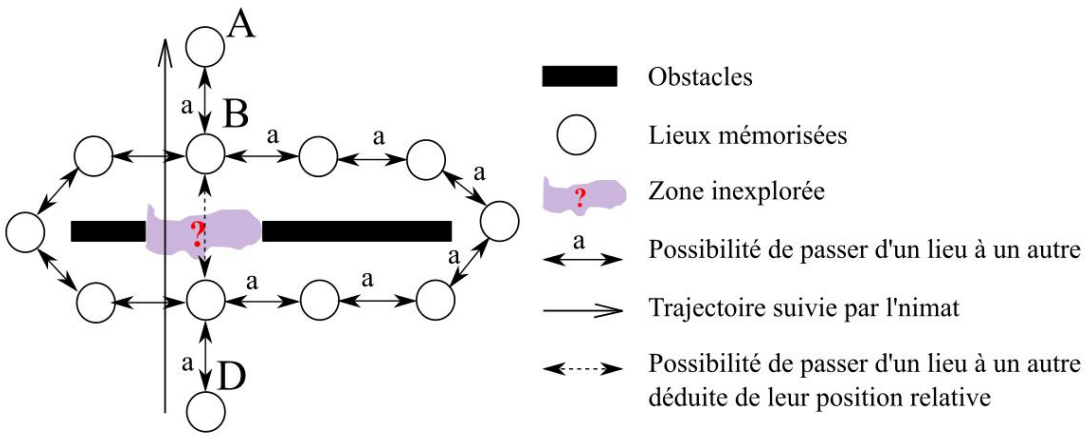


Figure I.14. *Navigation métrique*

En plus de calculer le chemin le plus court entre deux lieux mémorisés, cette technique permet de planifier des raccourcis au sein de zones inexplorées. Pour cela, la carte mémorise la position métrique relative de chacun des nœuds visités par le robot. Ainsi il est possible de prévoir un nouveau

déplacement, à travers une zone inexplorée, entre deux lieux. Dans l'exemple ci-dessus, le chemin liant en ligne droite le lieu D à A peut-être emprunté **[Dav 13].**

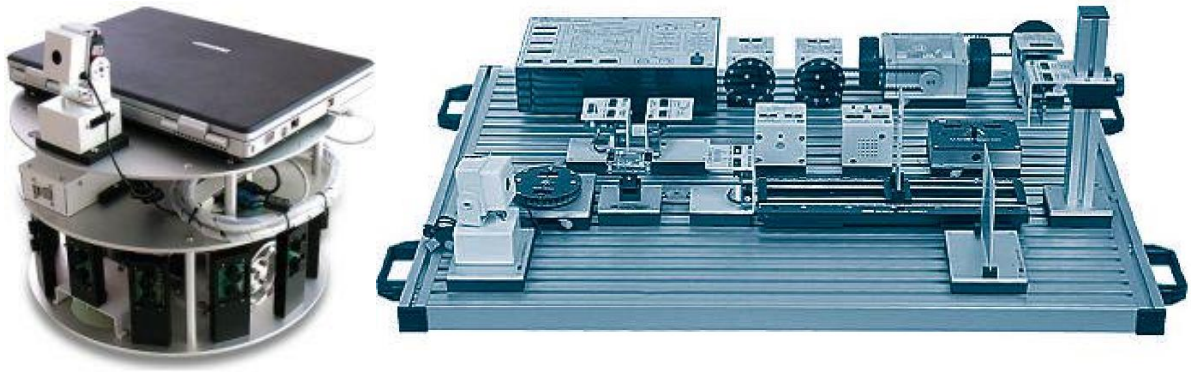
I.9. Conclusion

Dans ce premier chapitre, tout d'abord, nous avons défini la robotique mobile, comme étant parmi les domaines les plus innovants. Actuellement, elle occupe une place majeure dans notre quotidien grâce à la diversité de ces services et leurs exploitations dans différents milieux (terrestre, sous-marins ou aériens). Ensuite, on a fait un bref aperçu historique relatant le développement de la robotique. Enfin, on a essayé d'introduire brièvement ses notions de base, telles que la locomotion et ces différents types, la perception, la localisation et la navigation. Ces notions sont importantes pour la conception d'applications pour la commande des robots mobiles. Des études de développements sont toujours en cours pour avoir des robots autonomes et plus intelligent.

Chapitre II : Présentation des robots ED-7271/ED-7273 et de l'environnement IRES

II.1. Introduction

Dans cette section, en premier lieu, on va jeter un coup d'œil à la description des deux robots *ED-7273* et *ED-7271* et leurs configurations. Sachant que ces deux robots sont fournis avec le logiciel IRES (***I**ntelligent **R**obot **E**ducation **S**tudio*) qui permet de contrôler et de commander leurs différents modules, de ce fait, en deuxième lieu, on va donner les étapes d'installation de ce logiciel et enfin on va décrire brièvement quelques-unes de ces fonctionnalités les plus importantes.



(a) **(b)**

Figure II.1. *(a) Robot ED-7273 et (b) ED-7271*

II.2. Présentation du robot ED-7273

Le Kit de développement de robot intelligent *ED-7273* est un robot mobile de type uni-cycle. Il est entrainé par deux roues motrices, contrôlées individuellement avec deux servomoteurs à courant continu, et deux roues décentrées orientables pour l'équilibre. Il mesure *380 mm Ø × 260 mm (600 mm)* de hauteur et pèse *10 kg*. Il permet l'évitement d'obstacles et le déplacement autonome, en utilisant huit capteurs à ultrasons et quatre capteurs à infrarouges. En plus, le robot peut être reconfiguré en lui remplaçant certains capteurs tels que les capteurs de bataille (*battle sensors*) ou bien en lui rajoutant d'autres organes de perception tel que la *camera* pour la vision artificielle et le *stargaze*r pour la localisation du robot dans des environnements structurés intérieurs. Il est muni d'une batterie (*15 V - 8 A*) et une télécommande pour alimentation autonome embarquée et contrôle à distance.

II.2.1. L'architecture du robot

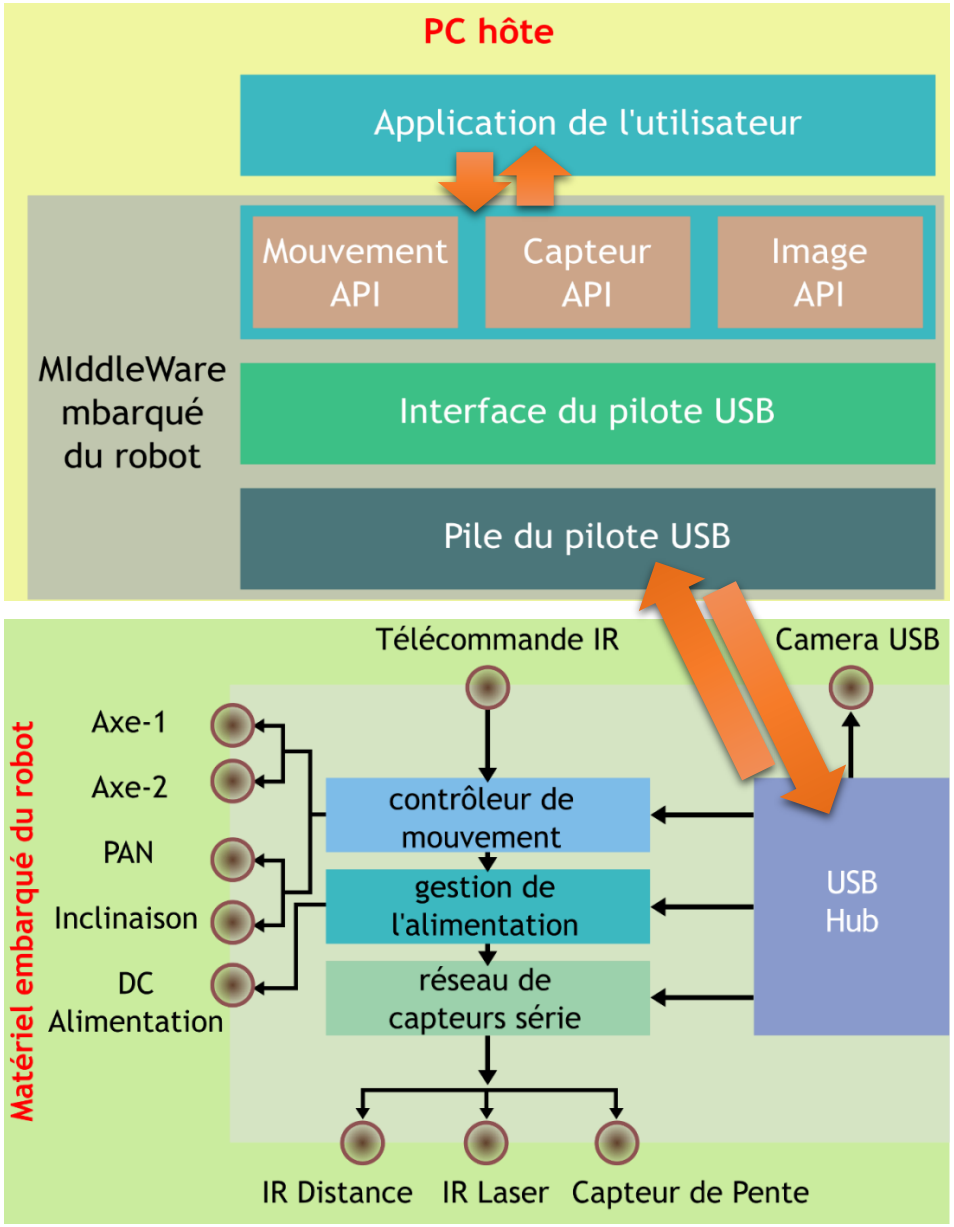


Figure II.2. *Architecture de communication ED-7273*

La partie *PC* représente ici le logiciel *IRES*, les pilotes et les applications créées par l'utilisateur. Une communication série entre le robot et le PC s'établit par le port *USB*. Les différents modules embarqués du robot sont connectés à l'*USB* et les capteurs forment un réseau série de communication.

II.2.2. Robot ED-7273

Le robot est devisé en quatre parties: Plaque de fond, plaque centrale, plaque supérieure, et ordinateur portable.

La plaque de fond

Elle contient tous les actionneurs et tous les capteurs, mise à part le dispositif de la camera qui est sur la plaque supérieur. Le robot est entrainé par deux moteurs à courant continu, chacun connecté à une roue, de deux roulettes qui le font déplacer en équilibre, et des capteurs qui permettent de réaliser des fonctions spéciales.

La plaque du milieu

Prend en charge la connexion et la communication entre le robot et l'ordinateur portable. Elle envoie la valeur du capteur de la plaque inferieur à l'ordinateur dans un sens et envoie le programme de commande vers la plaque dans l'autre sens. Elle est composée de l'USB_FIX qui prend en charge cette communication, du POWER_FIX qui fournit de l'énergie, d'une batterie et son guide de fixation, de l'ED-4657 reliant cette batterie avec le robot, et de PCB qui reçoit la valeur des capteurs et l'envoie à l'ordinateur, puis ce dernier transfère la commande à la plaque inférieure pour déplacer le robot.

La plaque supérieure

Se compose d'une caméra contrôlée par le moteur RC pour le traitement d'image, du capteur pour reconnaitre position (module optionnel stargazer), du récepteur pour la télécommande et d'un panneau pour fixer l'ordinateur portable.

L'ordinateur portable

Constitue la quatrième partie non embarquée du robot.

II.3. Présentation du robot ED-7271

Le robot intelligent *ED-7271* est équipé de tous les éléments nécessaires, y compris le contrôleur, le mécanisme et les capteurs. Ce qui permet d'améliorer son fonctionnement, sa configuration et sa conception par l'intermédiaire d'applications.

Dans l'ensemble, le *PC* prend en charge toutes les activités de contrôle. Quand il envoie une commande au module principal concerné, ce dernier la transmet aux moteurs et aux capteurs, puis récupère des données pour enfin les soumettre au PC à nouveau.

Le module de commande principal peut être considéré comme un messager entre le *PC*, le moteur et les capteurs. En contrôlant le capteur et commandant les actionneurs (moteurs), on peut faire déplacer le robot d'une manière autonome, et ce grâce à la programmation sur *PC*.

Le système d'application est équipé de divers capteurs et actionneurs, couramment utilisés dans les plates-formes intelligentes ou mobiles. En utilisant ce kit de pratique, on peut comprendre les principes et les caractéristiques de base de chaque module et acquérir le concept et le fonctionnement de base d'un robot à travers les tests d'application sur les combinaisons des modules.

II.3.1. Architecture du robot *ED-7271*

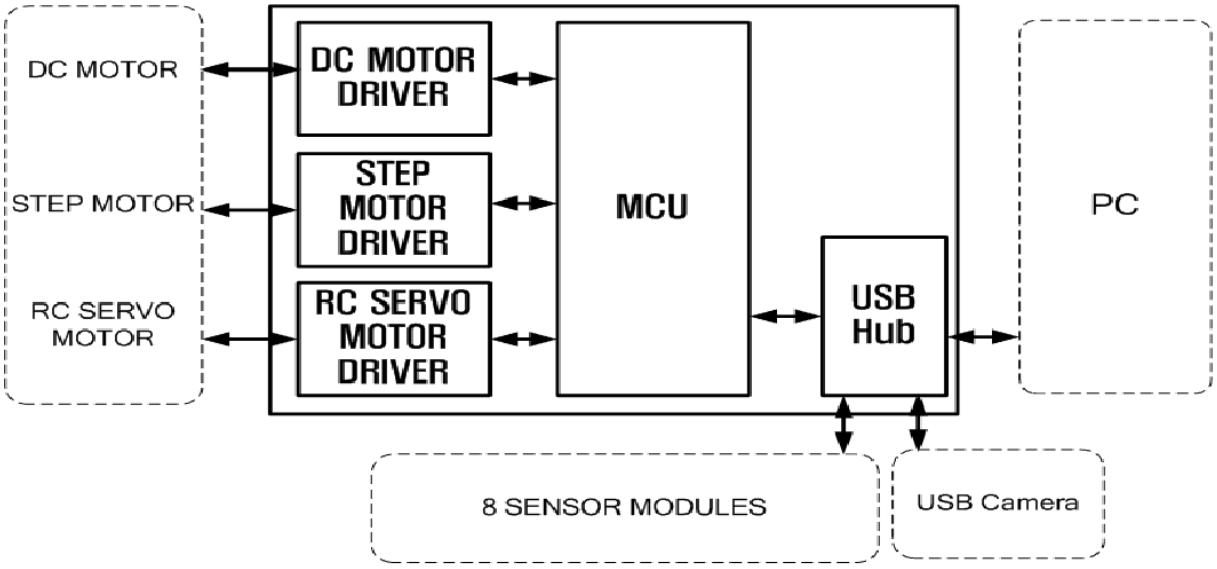


Figure II.3. *Architecture de communication ED-7271*

Le transfert de la commande entre le module principal et le *PC* s'effectue par *USB 2.0* de haut débit. La commande est transférée, via *USB*, à la *MCU* depuis le *PC*, pour faire tourner le moteur ou transférer la valeur du capteur au *PC*.

Pour qu'un signal du capteur soit lu depuis le *PC*, ce dernier est transféré à ce *PC* via un concentrateur *USB* Hub du module principal (**Figure II.3**). Le *PC* enregistre les informations sur l'existence d'obstacles et la détermination des distances. Ce qui permet au robot de se déplacer librement (sans difficulté).

Le système d'application du robot *ED-7271* est composé de sept types de capteurs, de trois types de pilote et de quatre types de module d'application, qui sont nécessaires pour les robots intelligents et mobiles.

II.4. Les capteurs

Les capteurs utilisés dans les applications du robot intelligent jouent un rôle très important dans sa configuration. Ceux cités ci-dessous sont utilisés dans les robots *ED-7271* et *ED-7273*.

II.4.1. Capteur à ultrasons (ED-7271 et ED-7273)

Le terme ultrason, indique un son non audible, par l'oreille humaine, ayant une fréquence au-delà de *14 kHz*, pour cet effet elle est souvent exploitée dans des zones industrielles.

La gamme de fréquence est définie selon le besoin d'utilisation des capteurs, mais celle normalement utilisée est comprise entre *9 kHz* à *50 kHz*, vu qu'elle est facile à générer. Son principe est de mesurer le temps *t* que met une onde générée par l'émetteur à être reçue par le récepteur après être réfléchie par un obstacle à une distance L.

$$L = \frac{t}{2} * u \quad avec\ u\ la\ vitesse\ du\ son$$

Ceux fournis avec les robots utilisent une onde ultrasonore de *40kHz*. Les capteurs d'ondes ultrasonores transfèrent les distances calculées vers l'ordinateur par le protocole spécifique.

Tension: DC *5V*
Plage de mesure: *1.0 cm ~ 400 cm*
Ultrasons fréquence active: *40 kHz*

Principe de l'application du capteur

Ce capteur utilise l'émetteur *Murata « MA40B8S »* pour l'envoi d'onde ultrasonore générée et le récepteur *«MA40B8R»* pour la réception de ces ondes, après qu'elles soient réfléchies par l'obstacle. Une fois l'impulsion de *40 kHz* transmise à l'émetteur, commence alors le comptage du temps pour se terminer à la réception de cette onde réfléchit. Ainsi, on peut déduire la distance jusqu'à cet obstacle.

Quand *ID* du capteur, préalablement ajusté, et le *CMD* sont envoyés par le *PC* pour le capteur ultrasons, ce dernier lui communique son *ID*, le *CMD* et la donnée distance au *PC*. Enfin, ces données sont enregistrées dans la mémoire.

Figure II.4. *Emetteur Murata MA40B8S*

II.4.2. Capteur de Distance Infrarouge ED-7271 et ED-7273

Ce capteur de lumière calcule la distance par la quantité de lumière réfléchie, capteur de distance infrarouge calcule indépendamment en amont la lumière réfléchie en utilisant la méthode *PSD* (position-sensitive device). Ce dernier est un capteur à base de semi-conducteurs. Il réagit à la lumière et possède une structure similaire à photodiode, sauf que sa résistance électrique au niveau des jonctions PN est plus élevée. Son principe de fonctionnement repose sur des calculs géométriques. Un signal lumineux (laser ou *LED* infrarouge focalisée) est envoyé vers la cible. La lumière réfléchie est collectée par une lentille puis projetée sur une matrice de photodiodes ou un *PSD*. La distance est donnée par $D = \frac{f.L}{X}$ avec f fréquence de l'onde émise (*5MHz*).

Les capteurs infrarouges sont constitués d'un ensemble émetteur/récepteur fonctionnant avec des ondes infrarouges, selon le principe de la triangulation optique. Ils possèdent l'avantage d'avoir un cône de détection beaucoup plus restreint. Il est possible de mesurer simplement le retour ou le non-retour d'une impulsion codée. Ce qui permet de détecter la présence ou l'absence d'un obstacle dans une certaines portion de l'espace.

Principes d'application du capteur

Le capteur *GP2D12* de Sharp permet de détecter la présence d'objet sans être considérablement influencé par leurs couleurs, par des surfaces réfléchissantes ou par des conditions de lumières ambiante. Il est utilisé pour le module *IR*. La forme de sortie de ce capteur infrarouge est une tension analogique proportionnelle à l'inverse de la distance de réflexion de l'obstacle c'est-à-dire La tension analogique de sortie diminue à mesure que la distance entre le capteur et l'obstacle est plus grande.

II.4.3. Capteur d'éclairement *ED7271-1*

La luminosité mesurée de la lumière peut être exprimée par niveaux de 0 à 10, et le capteur d'éclairement transfère ses données au *PC* par le protocole spécifié.

Gamme de mesure : 0~100 Lux
LED 0~10 Niveau affichage le long de la luminosité

II.4.4. Source d'éclairage du capteur ED7271-2

Comme un dispositif d'assistant pour l'expérience du capteur d'éclairement *ED7271-1*, ce système émet une lumière intense à la demande de l'utilisation spécifiée.

II.4.5. Capteur Pyroélectricité ED7271-7

Ce capteur est utilisé pour la détection d'une présence humaine. Il détermine l'humain à partir

de la variation de la température entrant dans l'objectif installé en face du capteur. Lorsque l'homme est perçu, la *LED* s'allume pour montrer sa perception et alors peut envoyer des données vers un *PC*.

Plage de mesure: Max 5 m

II.4.6. Boussole électronique ED7271-5

Système électronique indiquant la direction du nord et qui utilise 32 *LED*.

Plage de mesure: 0.0 ~ 359,9°

II.4.7. La reconnaissance vocale et de synthèse ED7271-21

Ce module comprend les fonctions de reconnaissance vocale, la reconnaissance de direction d'une source sonore, et la synthèse vocale, peut reconnaître la voix d'un utilisateur en utilisant l'approche indépendante du locuteur.

Distance de reconnaissance: *40 ~ 80 cm*

II.4.8. Capteur d'inclinaison ED7271-10

Ce module permet la mesure de l'inclinaison dans les axes X et Y, et envoi au *PC*, et il est communément utilisé dans les robots et les véhicules.

Plage de mesure: X, l'axe Y-90 ° ~ 90°.

II.5. Moteurs présents sur ED-7271 et ED-7273

II.5.1. Moteurs à courant continu (MCC)

Généralement, le moteur à courant continu nécessite pour sa mise en marche, une absorption d'une dizaine de *mA* à quelques *A* de courant électrique et une tension jusqu'à quelques centaines de volts. Pour commander ce moteur à courant continu, avec un circuit intégré *TTL-IC* ou un amplificateur opérationnel *OP* qui utilise relativement une faible quantité d'électricité, un transistor (*TR*) ou un *MOSFET* électrique est nécessaire.

Les méthodes de contrôle à transistor électrique *TR* incluent la méthode de fonctionnement linéaire et la *MLI* (Modulation Largeur d'Impulsion).

Pour faire varier la vitesse d'un moteur, il nous vient à l'esprit de faire varier la tension aux bornes du moteur. Malheureusement, pour de faibles valeurs, le moteur ne tourne pas. Le moteur

demande une tension minimale pour démarrer. Si cette dernière est trop basse, les forces électromagnétiques ne sont pas suffisantes pour vaincre les frottements.

Une solution astucieuse à ce problème est de fournir au moteur une tension qui est toujours la même, soit la tension maximale. Par contre, cette tension ne sera appliquée que par très courtes périodes de temps. En ajustant la longueur de ces périodes de temps, on arrive à faire tourner plus ou moins vite les moteurs. Plus encore, la vitesse des moteurs devient proportionnelle à la longueur des périodes de temps. Contrôler la longueur des périodes passées à la tension maximale par rapport au temps passé sans application de tension (tension nulle) est donc le cœur de la solution nommée MLI. Alors la tension moyenne absorbée par le moteur est exprimée par la relation suivante :

$$V_{moy} = \alpha . V_{cc} \qquad avec \quad \alpha = \frac{t}{T}$$

On ne doit pas oublier de deviser V_{moy} par *100*, lorsque α est exprimé en pourcentage.

Il est à noter que le contrôle d'un *MCC* ne se fait pas directement à travers le microcontrôleur. Car, la tension délivrée par ce dernier est limitée, en général *5V*, inférieure à la tension nominale du moteur. Pour cela, un circuit d'électronique de puissance commandé par le microcontrôleur est nécessaire. Le principal intérêt de la technique *MLI* est de limiter la chauffe des composants électroniques. Le *MLI* est un signal numérique, donc la tension peut prendre deux valeurs seulement. Dans certains cas très spécifiques (onduleurs à *MLI* par exemple), on fabrique un troisième niveau en inversant la tension du niveau haut, pour permettre de contrôler les deux sens de rotation.

Un encodeur optique, à *14* impulsions, est placé sur l'axe du moteur à courant continu (*DC-Motor*).

Encodeurs optiques

Selon le besoins, les encodeurs montés sur un système d'entrainement ont des taches différentes. Une des tâches principales d'un codeur est de permettre le positionnement d'un système d'entrainement. Ce positionnement peut se faire soit en mode incrémental soit en mode absolu. Le codeur peut être monté sur le moteur ou sur la ligne. Les codeurs peuvent également être utilisés pour la régulation de vitesse. Grace au retour moteur direct, ils améliorent significativement la précision de vitesse et la qualité de régulation, même en cas de fortes variations de charge.

Un codeur absolu, mémorise l'information de déplacement actuel, dans ce type de capteur on fait coder chaque position en utilisant un nombre suffisant d'émetteurs et de récepteurs optiques. On peut alors savoir la position à chaque instant.

Un codeur incrémental, en convertissant chaque incrémentation de position en impulsion à l'aide d'un disque codé avec encoches et un compteur d'impulsions.

II.5.2. Moteur pas à pas

Un moteur pas à pas est un moteur qui est alimenté en courant continu. Son rotor est constitué de N pôles magnétiques (Nord et Sud). Son Stator est constitué de bobines qui sont alimentées par un circuit électronique les unes à la suite des autres pour créer des pôles nord et sud en fonction du sens d'alimentation. L'axe du moteur tourne par pas. En fonction du moteur, le pas peut être compris entre 0.5° et 90°. Ce type de moteur se trouve dans un grand nombre de périphériques informatiques. On le trouve, par exemple, dans les imprimantes à jet d'encre pour déplacer les têtes d'impression ou entraîner le papier et dans les lecteurs de disquettes ou dans des disques durs qui demandent une grande précision de positionnement.

Le circuit d'attaque de moteur pour la commande pas à pas est inclus dans le module *ED-7271-14*.

II.5.3. Moteur RC

Le moteur RC à la base est appelé sous-moteur RC. Sa gamme de mouvement est autour de 0~180°. Il est utilisé pour le dispositif d'orientation des roues de voitures, et il est fréquemment utilisé pour les robots articulés en raison de sa commodité dans le contrôle de l'angle, avec alimentation et entrée de signal MLI, ce contrôle d'angle spécifique est disponible. Il est aussi utilisé pour le contrôle de l'angle de caméra, avec *2* servomoteurs RC, la commande de position, de sorte que la génération du mouvement pan et le basculement de la caméra est possible.

II.6. Réglage de l'environnement et l'installation IRES

On peut contrôler le matériel de robot formateur intelligent dans le mode de l'interface *USB 2.0*. Ainsi, il faut qu'on installe les pilotes des périphériques dans chaque matériel comme suit. L'unité robot formateur intelligent est expédiée de l'usine et livrés aux clients avec le pilote de périphérique déjà installé. Lorsqu'on veut installer un pilote en cours d'utilisation, on peut l'installer comme suit :

- Installation du contrôleur de mouvement *ED*
- Connecter au *PC* avec le câble USB après avoir allumé l'interrupteur d'alimentation, le PC détecte un matériel, "Périphérique EDMotion Control".
- Choisir son emplacement.

II.6.1. L'installation du programme IRES

Pour commander et contrôler les robots intelligents par ordinateur, il faut installer les applications interactives du programme *IRES* comme suit :

On fait un double clic sur *« Setup.exe »* sur la première fenètre on clique directement sur le bouton *« next »*, sur la deuxième on selectionne d'abord (I accept the terms of the license agreement) et on clique *« next »*, sur la suivante on ecrit le nom d'utilisateur puis appuyé sur *« next »*, après on selectione le nom du robot.

Apres avoir sélectionné le nom de robot, on sélectionne son type *« Intelligent Robot Kit »* selon l'option de produit et on appuyer sur le bouton *"Next"* pour entamer l'installation.

Lorsque on installe sur le système d'exploitation windows7 ou une version antérieure, le message de la figure suivante apparait, pas de panique faut juste appuyer sur *« installer »*.

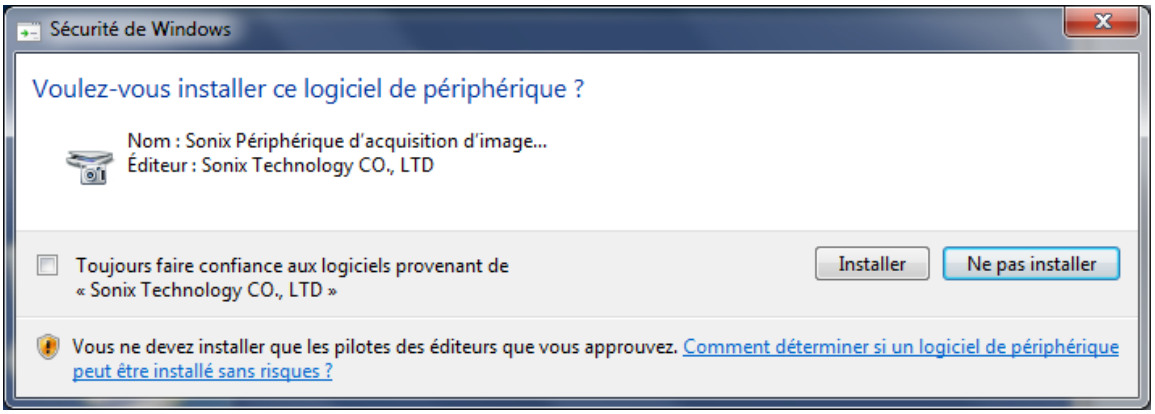


Figure II.5. *Message de Windows*

Lorsque l'installation est terminée, son raccourci et le programme sont enregistrés sur le bureau de l'ordinateur.

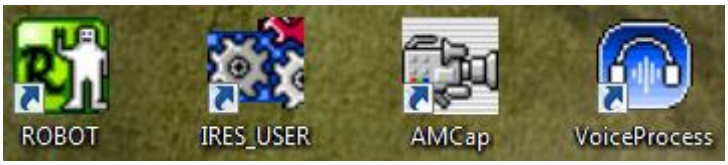


Figure II.6. *Icones IRES sur bureau*

II.6.2. Présentation de l'environnement IRES

IRES (Intelligent Robot Education Studio) est un logiciel de commande des robots, fabriqués par Ed Corp. Il est utilisé dans les systèmes d'applications basé sur des organigrammes. Il est applicable aux robots *ED-7273, ED-7270, ED-7271 et 7272*. Il fournit une bibliothèque de commande des robots. Une bibliothèque pour *C ++* est aussi disponible, mais qui requiert une certaine habilité.

IRES se compose de deux programmes, *IRES_USER* et *ROBOT*. Le premier est utilisé pour construire, compiler et charger le programme aux robots. Le second sert à afficher l'état des capteurs et les programmes en cours d'exécution, en temps réel.

II.6.3. Construire un programme sur IRES

Après le lancement du logiciel IRES, plusieurs fenêtres s'affichent :

1- **Espace de travail (*workspace*) :** c'est l'espace dans lequel on peut construire l'organigramme.
2- **Une cellule :** on choisit la cellule à mettre dans le *workspace*.
3- **Programme :** l'onglet qui contient les cellules de programmation comme début, fin, déclaration de variable globale, et autres.
4- **Control :** l'onglet des cellules pour le control des moteurs, bataille et voie.
5- **Branchement conditionnel :** c'est l'onglet des branchements conditionnels si *« if »*, selon la valeur lue par un capteur.
6- **Information :** l'espace ou s'affiche les informations sur les paramètres de la cellule sélectionnée.

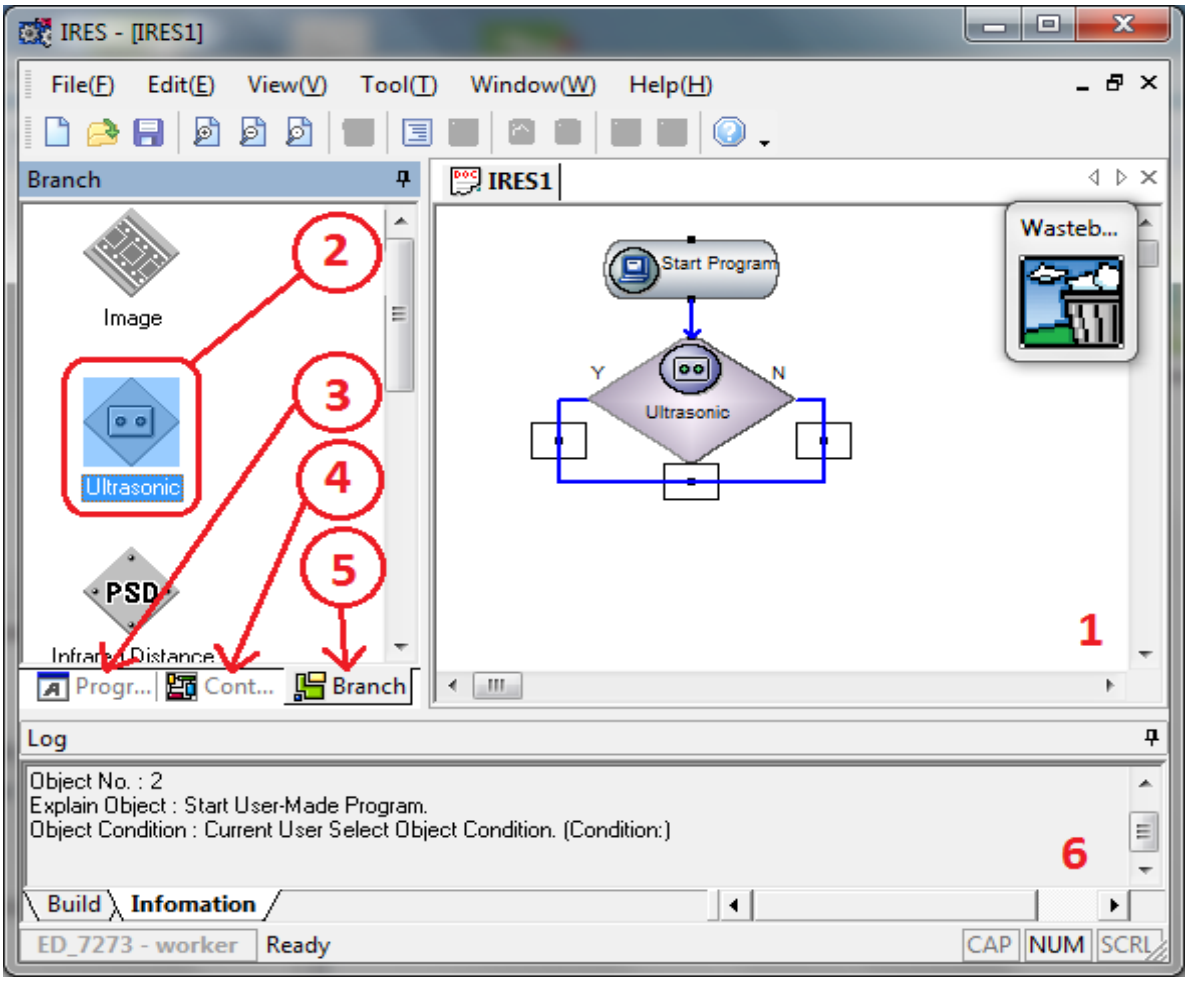


Figure II.7. *Fenêtre d'accueil IRES*

Pour faire un organigramme, on commence toujours par la cellule « *Start Program* », ne pas oublier de la nommer en double-cliquant dessus. Ensuite, selon les besoins de programmation, on choisit les cellules à partir des onglets concernés. Enfin, on termine toujours par la cellule *« End Program »*.

II.6.4. Description des cellules

Dans l'onglet programme on trouve par exemple

- **SubFunction :** c'est la cellule sous-programme, on peut la nommer en choisissant « *Set Environnement »* dans le menu contextuel. Dans cette cellule qui est, en fait, un environnement de travail, dans lequel on peut mettre un ensemble de cellules qui commence par « *Start Sub* » et se termine par « *End Sub* ».

- **Delay** : c'est la cellule dans laquelle on définit le temps, en millisecondes, d'attente du programme avant de passé à l'exécution de la cellule suivante. Attention à son utilisation, durant le temps choisit, le programme ne fait rien qu'attendre !

- **Move Step :** cette cellule permet le branchement sans condition vers la cellule désignée.

On trouve aussi dans l'onglet control

- **DC-Motor :** cette cellule permet de choisir le moteur (moteur gauche « LeftMotor » ou moteur à droite *«Right Motor »*), de choisir le mode de contrôle (contrôle de vitesse *« Speed Control Mode »* ou contrôle de position *«Position Control Mode»*), de définir le sens de rotation (le sens des aiguilles d'une montre *« CW »* ou sens trigonométrique *« CCW »*), et enfin de choisir la valeur à affecter. En cas de besoin, un environnement par ligne de commande *« script »* est proposé.

- **RcMotor :** permet d'agir sur les deux moteurs de la camera *«Tilt»* ou *« Pan»*. Là encore même chose que le DC-Motor en ce qui concerne le script.

Dans l'onglet branchement on trouve

- **Ultasonic** : cette cellule permet de sélectionner le numéro de capteur à tester, et de définir la condition sur la distance lue par ce capteur. On peut aussi utiliser le script.

- **Infrared Distance** : pour utiliser cette cellule on doit d'abord sélectionner le numéro de capteur et de définir la condition de test et la valeur à laquelle on compare la distance lue par le capteur choisi. On peut aussi utiliser le script pour le faire.

II.6.5. IRES Script

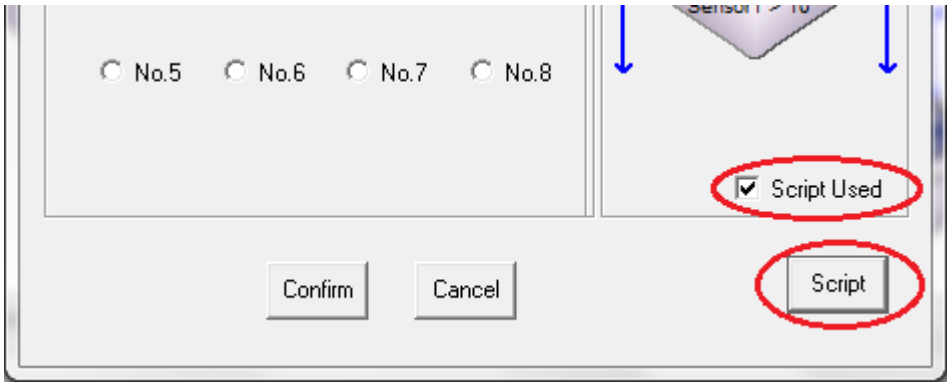


Figure II.8. *Exécution de script*

Pour que le scripte soit exécuté, il faudra cocher la case « *Script Used* ». Lorsqu'on clique sur le bouton « *Script* », une fenêtre pour écrire le scripte s'ouvrira. Il faudra alors écrire entre « *SubScript()* » et « *End Sub* » comme le montre ***Figure II.9.***

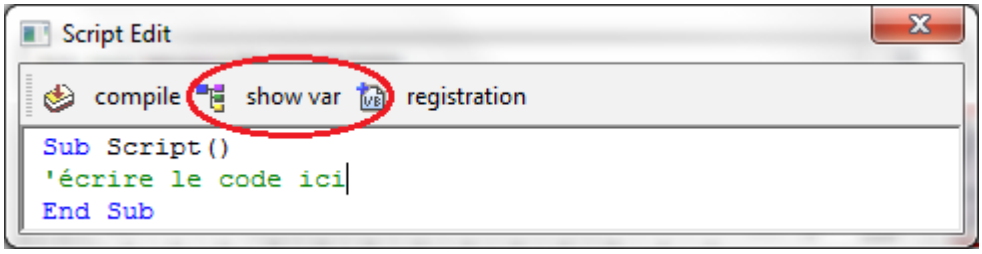


Figure II.9. *Ecriture d'un script*

Ensuite, si on clique sur « *show var* » une fenêtre contenant une liste de variables (***Figure II.10***).

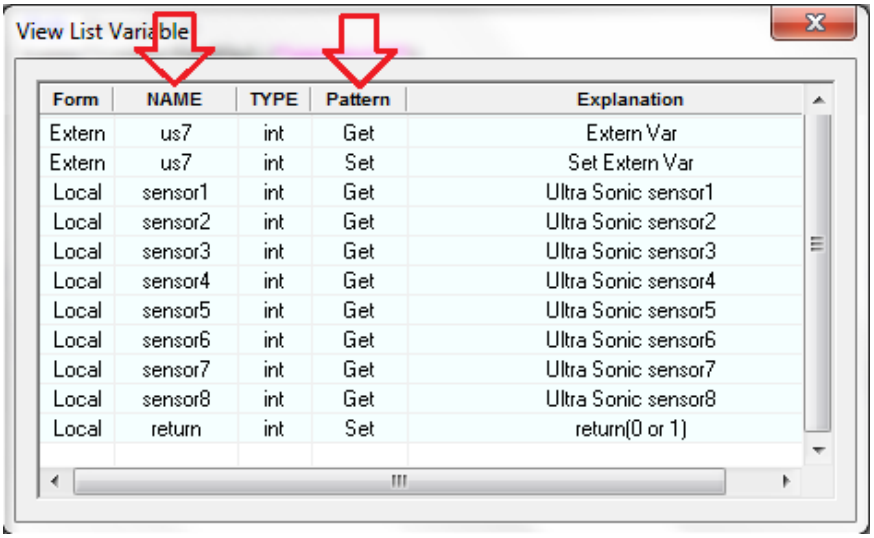


Figure II.10. *Affichage des variables d'un script*

Cette liste nous montre toutes les variables auxquelles on peut accéder par ce script. Par exemple, prenons la liste de ***Figure II.10***, on peut accéder en lecture seule aux mesures des capteurs ultrasoniques. On a un accès en lecture et en écriture à toutes les variables globales créées auparavant. Mais, on n'a aucun accès aux paramètres des moteurs ou aux autres capteurs. Pour ce faire, il faudra un autre bloc.

La première colonne indique la forme de la variable c'est-à-dire locale ou globale (externe). On peut voir dans cette liste, des variables locales qu'on n'a pas forcement crées, ce sont les variable via lesquelles on peut accéder aux paramètres des moteurs ou des capteurs. La deuxième colonne indique le nom de la variable. La troisième son type et la quatrième son mode d'accès écriture (*set*) ou lecture (*get*).

Si on place le curseur sur un endroit désiré dans la fenêtre scripte et on fait un double clic sur un nom de variable de la liste dans la fenêtre des variables, alors une ligne de commande comme celle-ci (**SetTagVal**"NomVariable") sera notée à l'endroit du curseur dans le script. Bien sûr, si cette ligne correspond à un mode d'accès « set » sinon ça sera (**GetTagVal (**"NomVariable"**)**.

II.7. Conclusion

Dans ce chapitre, on a présenté les deux robots sur quoi va se porter notre travail, leurs architectures et le mode de communication entre les différents modules de chaque robot et l'ordinateur. Comme nous avons exposé les différents capteurs et actionneurs présents dans les deux kits. Le fonctionnement des capteurs (ultrason et infrarouge par exemple) dépend d'un nombre important de paramètres comme la lumière, la vitesse, la taille, la forme et la matière des obstacles.

En ce qui concerne le logiciel *IRES*, nous avons procédé aux étapes d'installation, vu les différents paramètres de chaque module auxquels nous avons accès par l'ordinateur soit pour juste connaitre son état, soit carrément le régler (le cas des actionneurs).

Chapitre III : Programmation des Robots avec Logigramme et Script dans l'environnement IRES

III.1. Introduction

Dans ce chapitre, on s'intéressera à l'environnement de simulation (***IRES***) pour les deux robots. On va se focaliser sur les capteurs (à infrarouge et à ultrason) et le moteur à courant continu « *DC-Motor»,* qui sont les parties les plus essentielles dans la navigation en robotique mobile.

Plus exactement, on va développer des programmes pour tester le fonctionnement de ces capteurs et pour comprendre le contrôle des moteurs (sens, position et vitesse). Enfin, on va développer des programmes permettant le déplacement autonome du robot *ED-7273*.

III.2. Exemple 1 (va et vient pour ED-7271)

Dans cet exemple, on va essayer de simuler le comportement d'un véhicule ayant deux capteurs (un capteur à ultrason disposé à l'avant et un autre à infrarouge à l'arrière du mobile), qui fait un va et vient entre deux point dans un environnement structuré. Pour ce faire, on va élaborer un programme pour le faire déplacer, tout en surveillant la distance entre le mobile et l'obstacle.

III.2.1. Principe

Faire avancer le mobile avec une certaine vitesse, jusqu'à ce qu'il détecte, par le capteur frontale, un obstacle suffisamment proche. Il réduit sa vitesse, pour carrément changer de direction si cette distance est inférieure à une valeur minimale. Ce qui lui permet de ralentir avant le changement de direction. Les mêmes tests sont effectués, en le faisant reculer.

Avant d'entamer le programme, on va d'abord faire des tests séparément sur ces deux capteurs. On va tester le capteur à ultrason puis le capteur à infrarouge, sous l'environnement de programmation IRES. Le robot fixe (*ED-7271*) permet de faire des tests libres (sans crainte) sur les différents capteurs. On a établi des organigrammes pour exploiter la fonction ultrason en faisant des branchements conditionnels sur les distances lues par ces capteurs.

III.2.2. Première partie

En première partie, on a vérifié le déroulement du programme, pour le capteur à ultrason et à infrarouge, en effectuant les tests et les branchements suivants.

- Si la distance, *d*, entre le capteur et l'obstacle est supérieure à *20 cm* (*d > 20*), alors les deux moteurs tournent vers l'avant avec une vitesse de *20 tour/mn (rpm).*

- Si la distance est entre *20* et *13* centimètre (*13 < d ≤ 20*), alors aller vers l'avant à la vitesse*10 tour/mn.*
- Sinon, changer de direction et aller avec une vitesse de *10 tour/mn.*

On a fixé deux conditions sur les capteurs *« sensor1 ≤20 »* et *« sensor1 ≤ 13»* et affecté les valeurs nécessaires au paramètres des moteurs (mode de contrôle, vitesse en *tours/mn* et sens de rotation). Faire varier ces distances en déplaçant l'obstacle fixé sur la règle « *Slide module ED7271-16* », permet de vérifier le déroulement.

- En premiers lieu, on a positionné l'obstacle à une distance *d = 21.2 cm*, supérieure à celle de la condition. Les figures suivantes montrent le déroulement de ces programmes, le branchement après le test est défini par le sens de parcours, et l'étape exécutée (en cours) est colorée en noir.

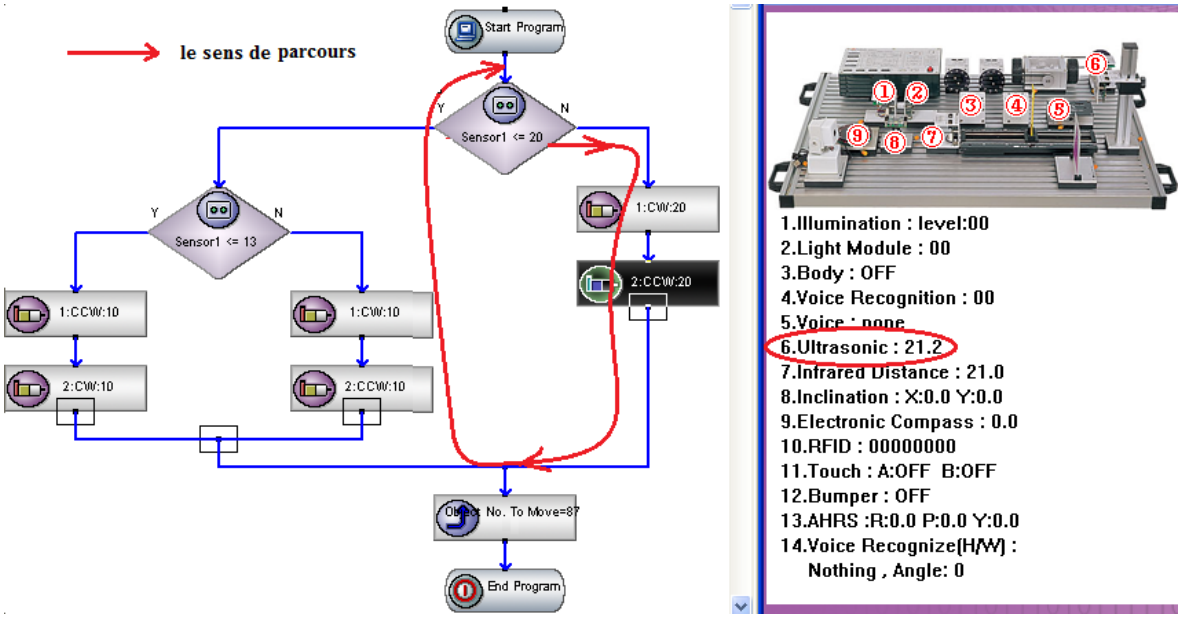


Figure III.1. *Test numéro 1 avec capteur ultrason dans l'environnement IRES « d = 21.2 cm ».*

On voit bien que, pour cette valeur de *d*, la première condition n'est pas satisfaite les moteurs tournent vers l'avant avec une vitesse *20 tours/mn.*

- En deuxième lieu, la valeur lue par ce capteur est *d=19.9 cm.*

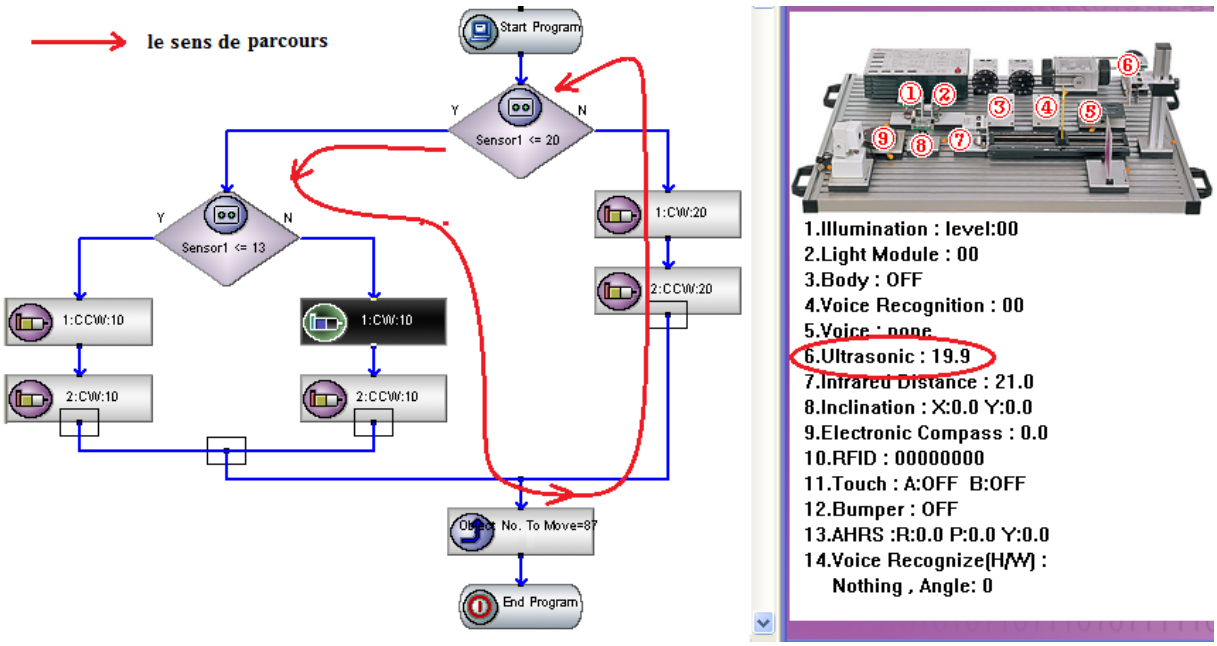


Figure III.2. *Test numéro 1 avec capteur ultrason, d=19.9, dans l'environnement IRES*

Dans ce cas, la valeur de *d* détectée par le capteur est *19.9 cm (13 < d ≤20)*. On remarque que la première condition est satisfaite contrairement à la deuxième condition. La trajectoire a donc pris un autre sens de parcours, faisant ainsi diminuer la vitesse des moteurs comme prévu.

- En troisième lieu, la valeur lue par ce capteur est de *13.6 cm*.

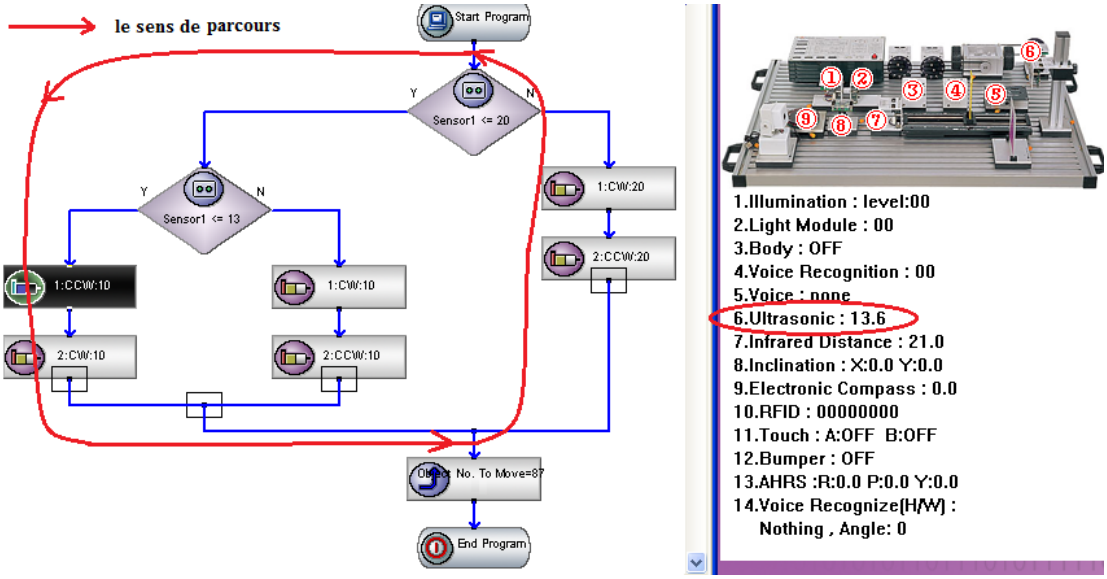


Figure III.3. *Test numéro 1 avec capteur ultrason dans l'environnement IRES « d=13.6 cm ».*

La deuxième condition *« sensor1≤ 13 »* n'est pourtant pas satisfaite mais le déroulement de ce programme n'a pas pris le parcours prévu, (*13.6 > 13*). Comme on peut le remarquer les valeurs admises, dans la cellule condition sur l'ultrason, sont toutes entières et comprises entre *10* et *100 cm*, alors que la valeur affichée est réelle. La seule justification dans ce cas est due au fait que cette

cellule est créée (construite) par une classe qui utilise la conversion (le *cast*) réel vers entier, qui fait carrément enlevé la partie après la virgule. Alors *13.6* est vu comme *13* qui satisfait la deuxième condition.

- En quatrième lieu, la valeur lue par ce capteur est de *11.0cm*.

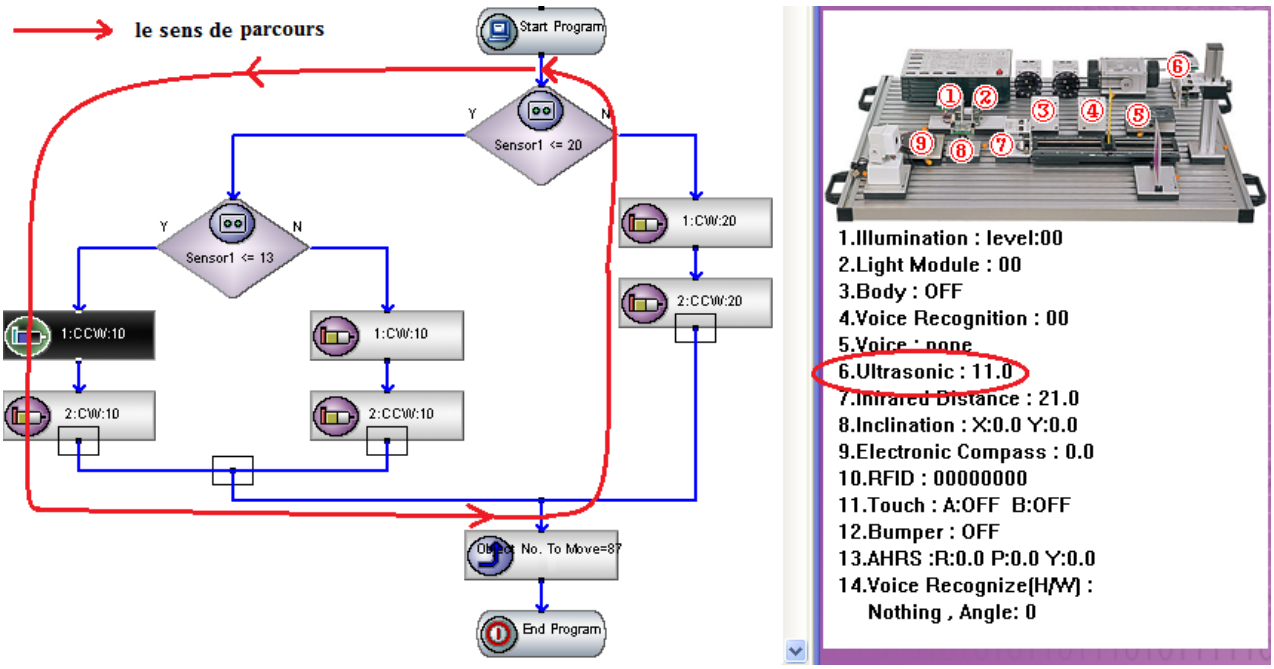


Figure III.4. *Test numéro 1 avec capteur ultrason dans l'environnement IRES « d=11.0 cm ».*

Dans ce cas, la valeur détectée par le capteur est $d = 11cm$, les moteurs ont changé de direction. Le déroulement a donc pris le sens prévu et s'est parfaitement exécuté.

Enfin, on s'intéresse au capteur infrarouge et en refait presque les mêmes étapes que celle précédentes. Et on obtient les résultats illustrés dans les figures suivantes.

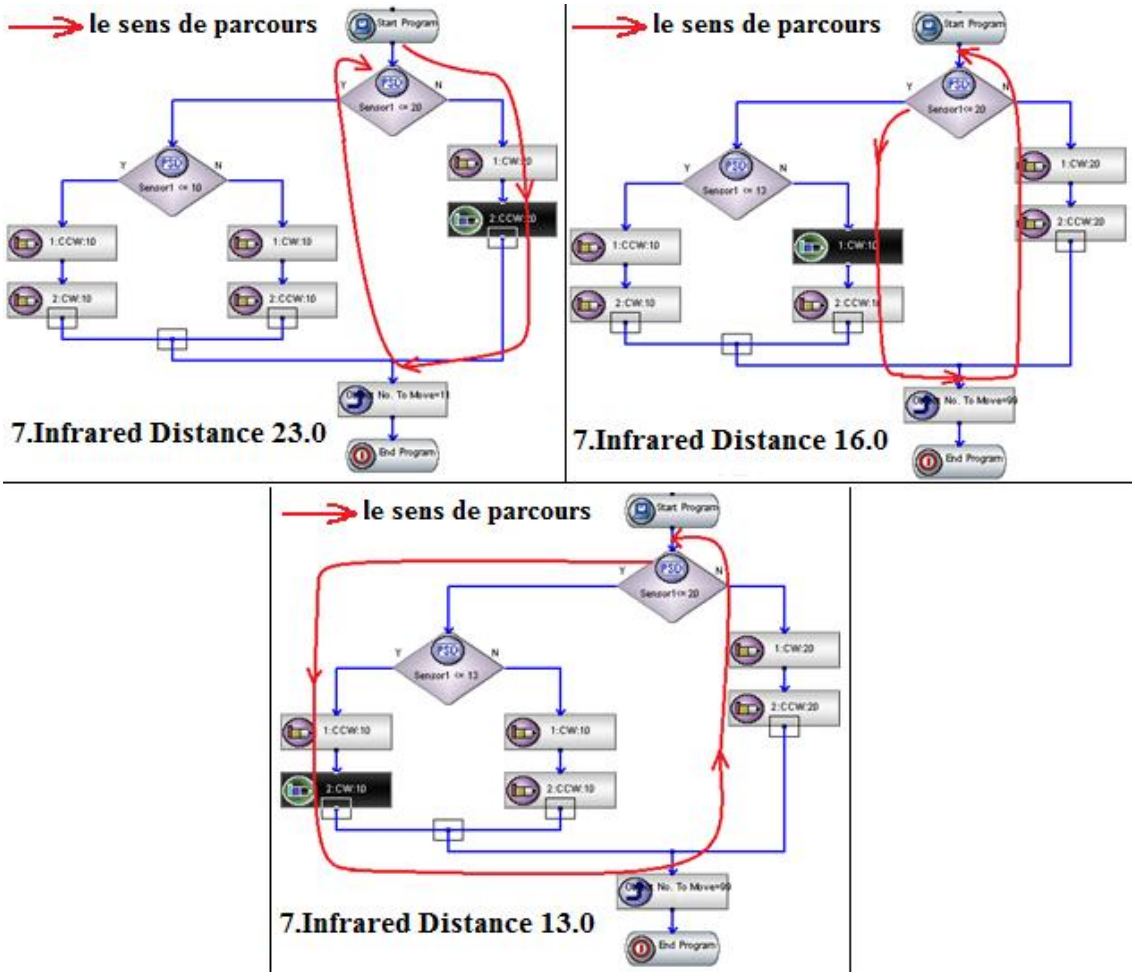


Figure III.5. *Test numéro 1 avec capteur infrarouge dans l'environnement IRES*

On remarque un comportement similaire à celui avec les capteurs à ultrasons. La seule différence réside dans les distances mesurées par ces capteurs, qui sont lues et affichées en tant que valeurs entières.

III.2.3. Deuxième partie

Dans cette partie, on va s'intéresser aux distances lues par les capteurs et affichés par IRES, et aux positions réelles de l'obstacle (mesuré par la règle du *slide module*).

Tableau III.1. *Les mesures du capteur ultrason*

Mesurée (*cm*)	3	4	5	6	7	8	9	10	11
Affichée (*cm*)	1,7	2,3	3,2	4,6	5,2	6,5	7,1	8	9,1
Mesurée (*cm*)	12	13	14	15	16	17	18	19	20
Affichée (*cm*)	9,9	11,2	12	13,3	13,7	14,8	15,9	17,1	17,6
Mesurée (*cm*)	21	22	23	24	25	26	27		
Affichée (*cm*)	18,7	19,1	20,4	22,1	22,3	24	25,4		
err_rel_moy	0.169								

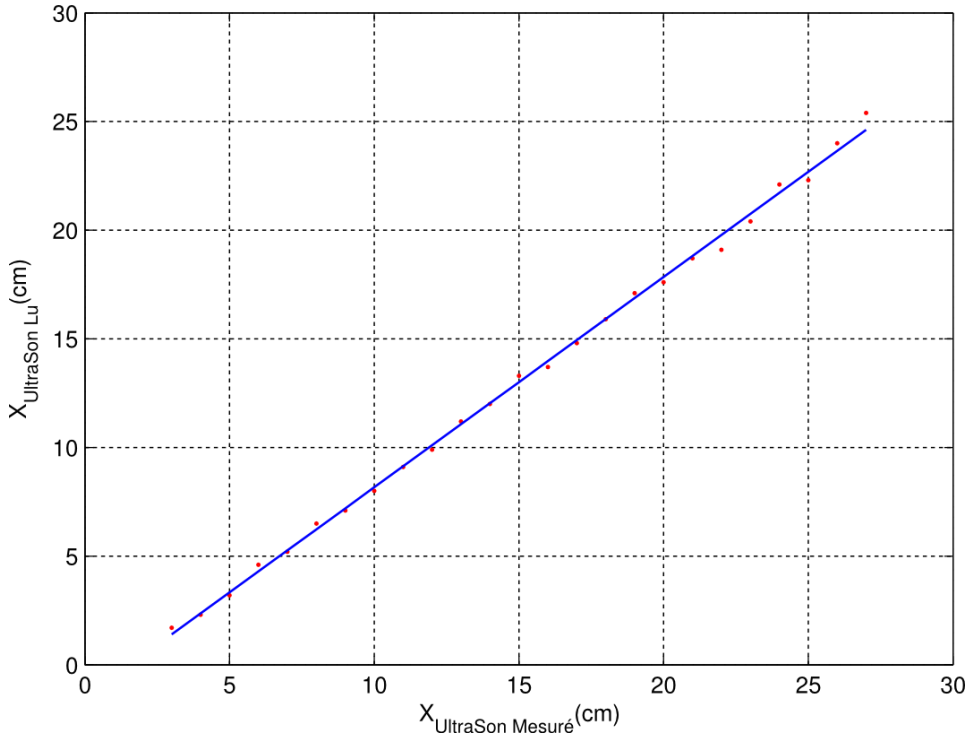


Figure III.6. *Les mesures lues par le capteur à ultrason*

D'après cette figure, on remarque que les points sont disposés d'une manière linéaire, mais légèrement décalé par rapport à la première bissectrice avec une erreur relative moyenne de *0.169* (*~ 17%*). En utilisant la méthode des moindres carrés, on a calculé l'indice de corrélation *Ic= 0.9991*, qui indique une forte linéarité, et tracé la droite de régression (*a = 0.9676*, *b = -1.5062*) représentée en bleu.

En ajoutant *1.8 cm* à chaque valeur lue par le capteur, l'erreur relative moyenne devient *0.025* (*2.5%*). D'après le manuel *ED-7273*, la plage de mesure est entre *1.0cm ~ 400 cm.*

Tableau III.2. *Les mesures du capteur à infrarouge*

Mesurée (*cm*)	3	4	5	6	7	8	9	10	11
Affichée (*cm*)	8	6	6	6	7	8	9	10	11
Mesurée (*cm*)	12	13	14	15	16	17	18	19	20
Affichée (*cm*)	12	13	14	15	16	17	18	19	20
Mesurée (*cm*)	21	22	23	24	25	26	27		
Affichée (*cm*)	21	22	23	24	25	26	27		
err_rel_moy	0,095								

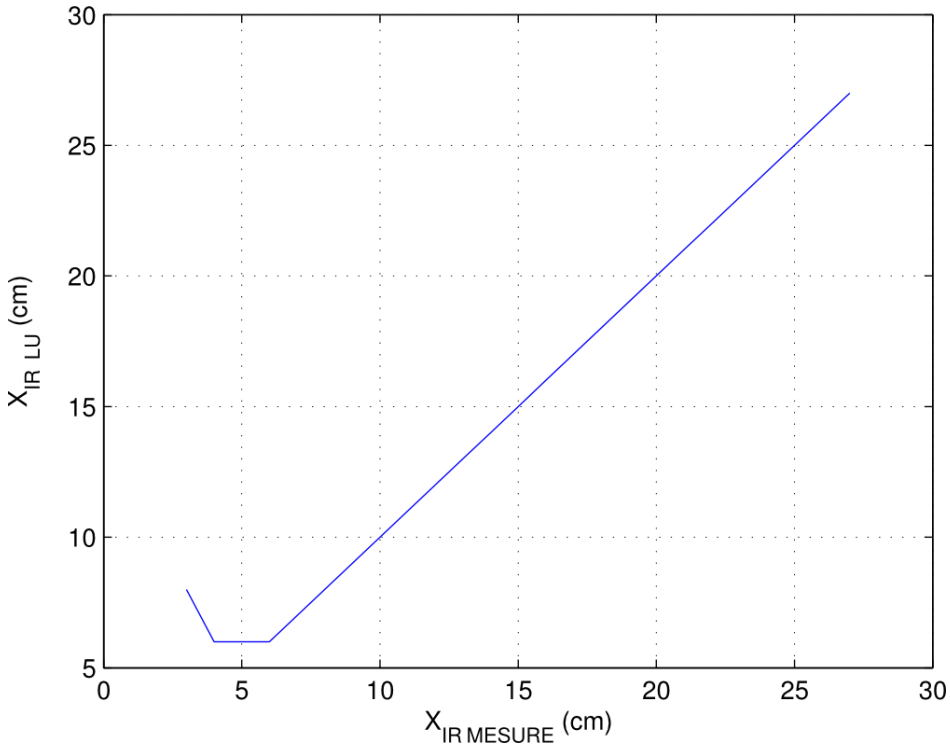


Figure III.7. *Les mesures lues par le capteur infrarouge*

On remarque que les valeurs mesurées inferieures à *6 cm*, ne sont pas bien détectées par ce capteur. Par contre, celles lues au-delà de *6 cm* sont exactement identiques à celles mesurées. Tous les points entre *6 et 27 cm* présentent une linéarité parfaite.

D'après le manuel *ED-7273*, la plage de mesure est entre $10 \sim 80cm$.

III.2.4. Troisième partie

Enfin, en combinant les deux programmes et en assurant les branchements nécessaires, on obtient le logigramme, illustré par la **Figure III.8**, qui peut garantir le fonctionnement prévu, au début de cet exemple.

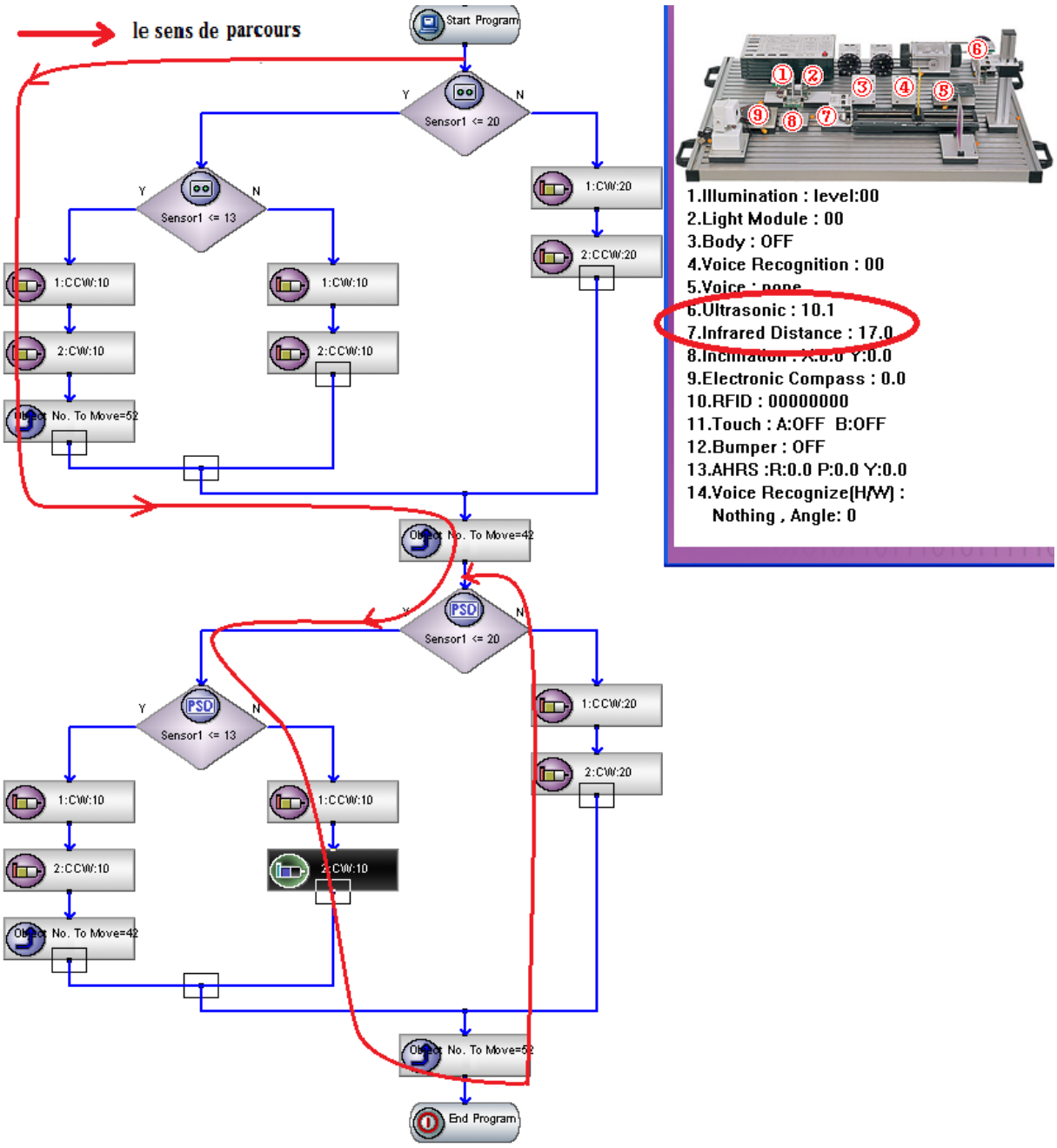


Figure III.8. *Programme du véhicule sous l'environnement IRES*

Après avoir exécuté ce programme, pour différentes positions des obstacles, on a pu vérifier son bon fonctionnement.

III.3. Exemple 2 (Evitement d'obstacles pour ED-7273)

Dans ce deuxième exemple, on va utiliser le robot *ED-7273* et lui charger un programme d'évitement d'obstacles, en se basant sur la mesure de distance par télémétrie et l'orientation par l'odométrie. Pour des raisons de fiabilité et de portée, on ne va exploiter que les capteurs frontaux à ultrason (de *1* à *5*).

III.3.1. Principe

On fait avancer le robot jusqu'à l'apparition d'un risque de collision imminente avec un obstacle, pour l'arrêter et l'orienter vers une autre direction sans risque. Le robot ne fera pas de marche arrière mais pourra faire un demi-tour en deux temps. On fait tourner le robot autour de son centre, qui est le point milieu entre ses deux roues motrices, afin d'éviter qu'il rentre en collision avec les obstacles environnant qui l'entoure.

III.3.1.1. Evaluation des risques de collision

Pour que le robot puisse avancer sans rentrer en collision avec un obstacle, on doit calculer les distances minimales en connaissant la forme géométrique du robot et la disposition de ses capteurs (**Figure III.9**).

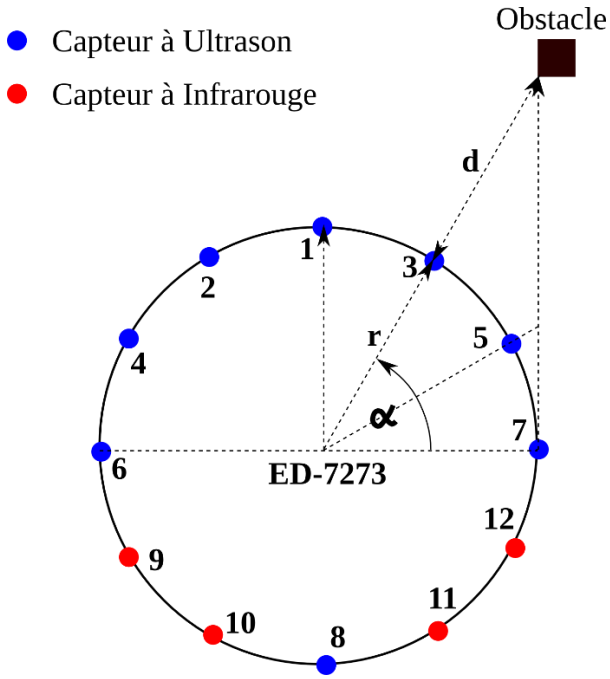


Figure III.9. *Distance minimale d'évitement d'obstacle*

Comme l'illustre la figure, α représente l'angle d'orientation du capteur et d la distance minimale entre le robot et l'obstacle qui peut être calculée comme suit

$$(d + r) = \frac{r}{cos(\alpha)} \Rightarrow d = r * \left(\frac{1}{\cos(\alpha)} - 1\right)$$

Pour $\alpha = 30°$ on a $d = 2.939\ cm$

Pour $\alpha = 60°$ on a $d = 19\ cm$

Les capteurs dans ED-7273 sont disposés de manière symétrique. Alors, pour éviter la collision en avançant (ou en reculant) en ligne droite, les distances lues par les capteurs 2 et 3 (respectivement par les capteurs 10 et 11) doivent être supérieurs à *19 cm*. De même pour les distances lues par les capteurs 4, 5, 9 et 12 doivent être supérieurs à *2.939 cm*, sans oublier de surveiller à chaque fois les capteurs 1 ou 8. En revanche, les capteurs 6 et 7 ne sont pas directement concernés par la collision. C'est-à-dire, quand le mobile avance, ces capteurs ne perçoivent pas les objets susceptibles de rentrer en collision avec lui. C'est pourquoi on n'a utilisé que les capteurs de 1 à 5 pour établir la table de décision suivante.

Tableau III.3. *Table de décision*

US 1	US 3	US 2	US 5	US 4	
1	1	1	1	1	**AV**
1	1	1	1	0	**TD_30**
1	1	1	0	1	**TG_30**
1	1	1	0	0	**TD_90**
1	1	0	1	1	**TD_45**
1	1	0	1	0	**TD_45**
1	1	0	0	1	**TG_60**
1	1	0	0	0	**TD_90**
1	0	1	1	1	**TG_45**
1	0	1	1	0	**TD_90**
1	0	1	0	1	**TG_45**
1	0	1	0	0	**TD_90**
1	0	0	1	1	**TD_60**
1	0	0	1	0	**TD_60**
1	0	0	0	1	**TG_60**
1	0	0	0	0	**TD_90**
0	1	1	1	1	**TD_30**
0	1	1	1	0	**TD_30**
0	1	1	0	1	**TG_30**
0	1	1	0	0	**TD_90**
0	1	0	1	1	**TD_30**
0	1	0	1	0	**TD_30**
0	1	0	0	1	**TG_30**
0	1	0	0	0	**TD_90**

US 1	US 3	US 2	US 5	US 4	/
0	0	1	1	1	**TG_30**
0	0	1	1	0	**TD_60**
0	0	1	0	1	**TG_30**
0	0	1	0	0	**TD_90**
0	0	0	1	1	**TD_60**
0	0	0	1	0	**TD_60**
0	0	0	0	1	**TG_60**
0	0	0	0	0	**TD_90**

Une présence d'obstacle pouvant empêcher l'avancement du véhicule est exprimée dans la table par zéro *« **0** »*, et une absence d'obstacle par un *« **1** »*. Nous avons aussi utilisés les abréviations suivantes pour exprimer les actions à prendre pour chaque cas.

AV : avancer.

TD_x : tourner à droite avec un angle de x degré.

TG_x : tourner à gauche avec un angle de x degré.

On a coloré les cases de la table en gris, pour indiquer que faire un test sur ce capteur est inutile, puisque on aboutit sur la même action dans les deux cas.

III.3.1.2. Orientation du robot

Pouvoir utiliser cette table, exige un contrôle sur l'orientation du robot. En se basant sur les calculs d'odométries et en supposant un roulement sans glissement (sachant que la vitesse de rotation du mobile est *V (en tours/mn))*, on peut facilement orienter le robot, c'est-à-dire le faire tourner d'un angle *α en degrés*. Alors, au moins deux possibilités s'offrent à nous, soit faire un contrôle de position sur les deux roues (moteurs) ou bien utiliser des tempos (*Timers*).

a. En utilisant un contrôle de position

Soit θ l'angle que la roue doit faire, *2R* le diamètre de la roue et *2L* la distance entre les deux roues, on a alors

$$\theta . R = \alpha . L \implies \theta = \frac{\alpha . L}{R}$$

On a *R=75 mm* et *L=120 mm,* alors θ est donnée par la relation suivante :

$$\theta = \frac{\alpha . 120}{75} = 1.6\ \alpha$$

Pour $\alpha = 45° \implies \theta = 72°$

Pour $\alpha = 90° \implies \theta = 144°$

Pour vérifier les calculs théoriques obtenus, on a élaboré le logigramme ci-dessous, dans lequel on a mis une cellule de branchement conditionnel avec un *goto*, pour faire une boucle infini et donner ainsi le temps, pour la commande en position, de se terminer.

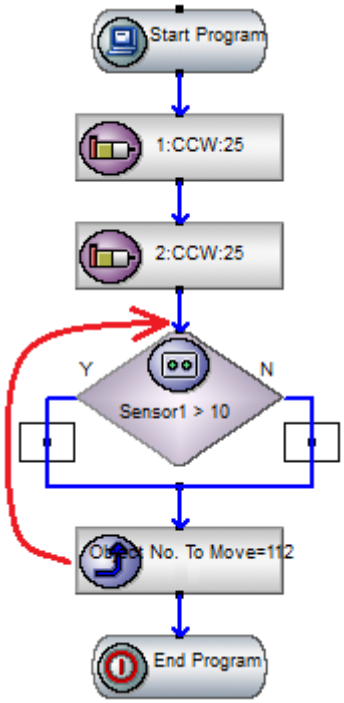


Figure III.10. *Logigramme pour le test de la commande en position*

Apres exécution de ce logigramme, on a trouvé les résultats suivants :

- Pour avoir $\alpha = 45°$ il faut un angle $\theta = 100°$

- Pour avoir $\alpha = 90°$ il faut un angle $\theta = 200°$

Les résultats obtenus ne correspondent pas à ceux calculer théoriquement et c'est dû à la faible résolution des encodeurs optiques, qui est de 14 impulsions (soit une erreur de presque 26°).

b. En utilisant les tempos (*Timer*)

Etant donné le diamètre de la roue *2R* et la distance entre roues *2L* alors on a :

$$L.\alpha = V.\frac{360}{60}.t.R \implies t = \frac{L.\alpha}{6.V.R}$$

On a *R=75 mm* et *L=120 mm*, alors *t* en second est donnée par la relation suivante :

$$t = \frac{120.\alpha}{6.75.V} = \frac{20}{75.V}\alpha$$

$$t = \frac{4}{15} * \frac{\alpha}{V}$$

On a fixé *V=20 tours/mn*, alors

$$t = \frac{4.\alpha}{15.20} = \frac{1}{75}\alpha$$

Pour $\alpha = 45° \Longrightarrow t = 0.6\ s$

Pour $\alpha = 90° \Longrightarrow t = 1.2\ s$

Après avoir choisi, pour chacun des deux moteurs, le mode « *contrôle de vitesse* » avec une valeur *V=20 tours/mn*, choisi le sens de rotation et fixé la valeur du temps au Delay à $t = 600\ ms$, on a remarqué que le robot ne tourne pas avec l'angle désiré ($\alpha = 45°$), et surtout qu'il n'y-a aucune linéarité entre t et α. C'est pourquoi on a décidé de déterminer expérimentalement les durées des tempos utilisées. Apres différents tests on a abouti aux résultats suivants :

Pour tourner de $\alpha = 45°$ et $\mathrm{V} = 20\ \mathrm{tours/mn} \Longrightarrow t = 100\ ms$

Pour tourner de $\alpha = 90°$ et $\mathrm{V} = 20\ \mathrm{tours/mn} \Longrightarrow t = 1s = 1000\ ms$

Pour tourner de $\alpha = 30°$ et $\mathrm{V} = 15\ \mathrm{tours/mn} \Longrightarrow t = 100\ ms$

Pour tourner de $\alpha = 60°$ et $\mathrm{V} = 25\ \mathrm{tours/mn} \Longrightarrow t = 100\ ms$

III.3.2. Conception du programme d'évitement d'obstacles

En se basant sur la table de décision et les résultats obtenus dans l'orientation en utilisant les tempos, on a finalisé le logigramme ci-dessous

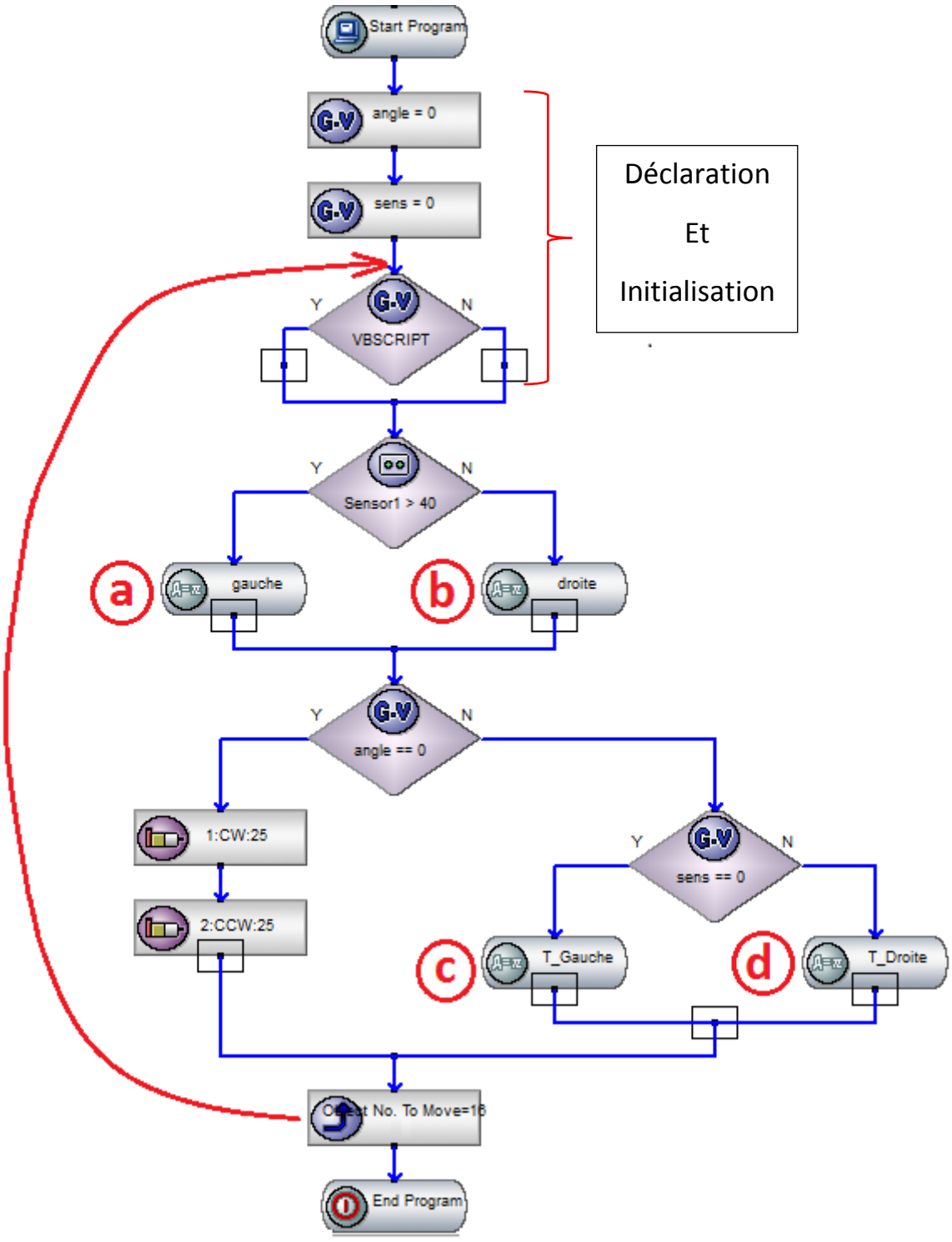


Figure III.11. *Programme complet d'évitement d'obstacles*

Pour des raisons de lisibilité, on a découpé en deux parties et mis la ***Figure III.12.*** *Sous-programme « Gauche »* dans **Annexe A.1** et **Annexe A.2**. On a fait la même chose pour la ***Figure III.13.*** *Sous-programme « Droite »* pour la mettre dans **Annexe A.3** et **Annexe A.4**

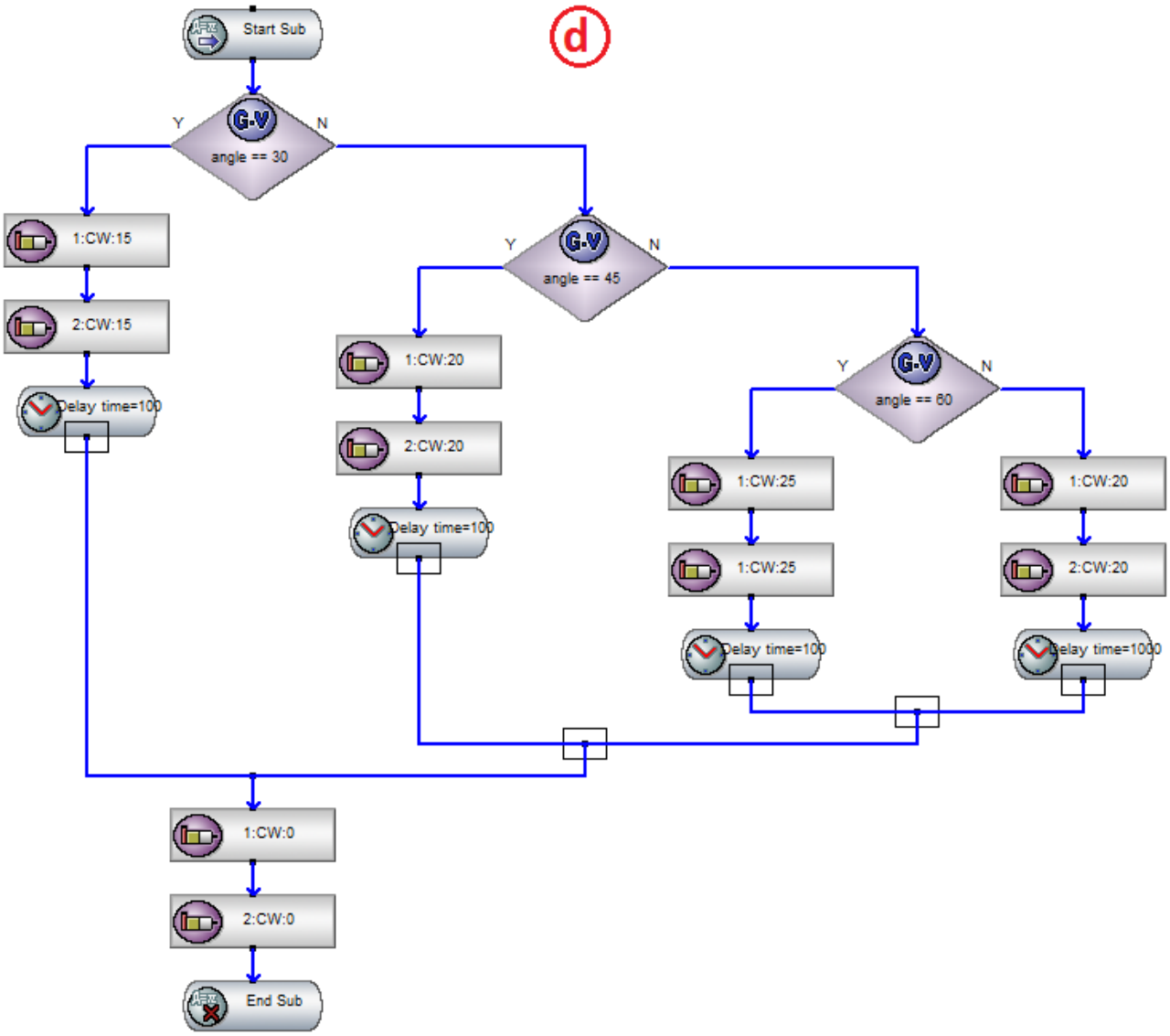


Figure III.14. *Fonction pour tourner à gauche« T_Gauche »*

Pour ne pas encombrer le logigramme principal, on a utilisé des fonctions « *subfuncntion* ». Dans cet exemple, on a créé deux fonctions « *gauche* » et « *droite* ». ***Figure III.14*** illustre la Fonction pour tourner à gauche « *T_Gauche* ». En ce qui concerne la fonction pour tourner à droite « *T_Droite* », on l'a représentée par le même logigramme, à l'exception dans les cellules *DC-Motor*, on a inversé le sens de rotation en *CCW*.

Pour définir une action après vérification des capteurs, on a utilisé deux variables globales « *sens* » qui prend deux valeurs (*1* pour tourner à droite et *0* pour tourner à gauche) et « *angle* » qui prend la valeur de l'angle d'orientation. Pour spécifier le sens et l'angle d'orientation voici un exemple de scripte utilisé.

Script

```
SubScript()

SetTagVal"angle",30' tourner avec un angle de 30°
SetTagVal"sens",1' 1 pour tourner à droite
SetTagVal"return",1

EndSub
```

Apres exécution du programme précédent, on a pris ces quelques photos

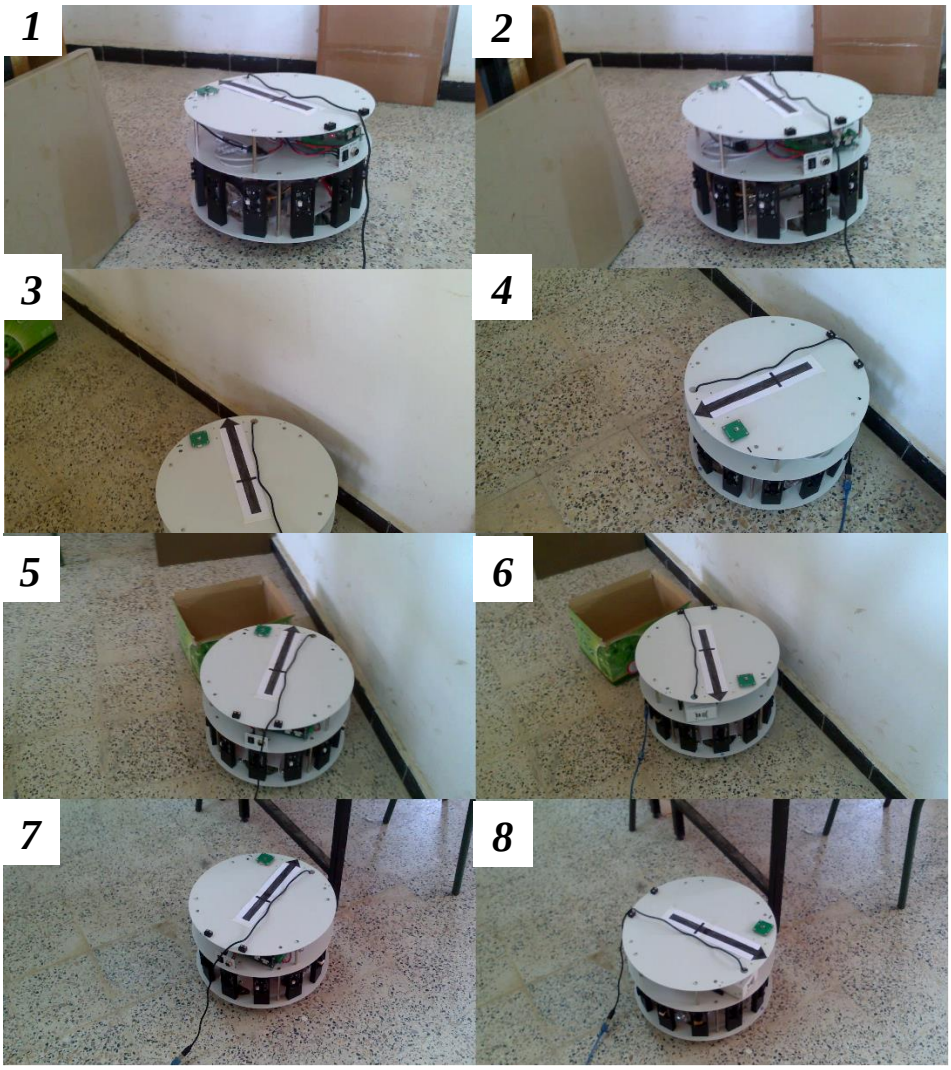


Figure III.15. *Détection et évitement d'obstacles*

Dans les photos (*1 et 3*) le robot s'est arrêté en détectant un obstacle. Puis, il s'oriente pour l'éviter comme le montre les photos (*2 et 4*). Concernant les photos (*5 et 6*), on voit bien que le robot a fait un demi-tour, et les photos (*7* et *8*) montrent la détection d'une forme étroite (le pied d'une table).

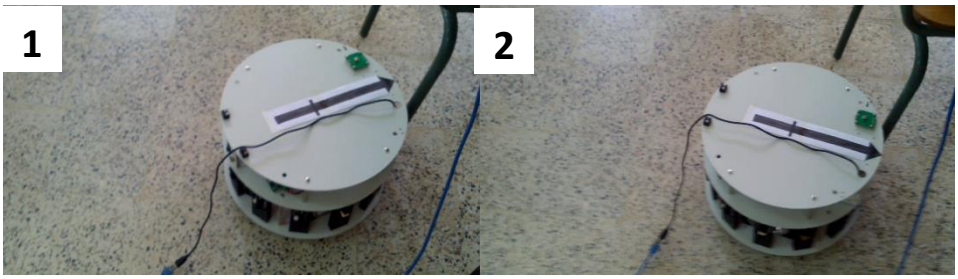


Figure III.16. *Réaction tardive du robot.*

Les deux photos de Figure III.16 montrent le robot s'orienter pour éviter le pied de la chaise, après l'avoir frôlé (légèrement touché). Ceci est dû, d'une part, à la forme fine de l'obstacle et, d'une autre part, à cause du retard de réaction du robot, qui est dû au mode de communication (en série par trame *« frame »*) entre les différents modules du robot et le *PC* et, enfin, à cause de la vitesse ; plus exactement à cause des forces de Coriolis qui apparaissent au freinage et le font déplacer de quelques centimètres.

III.3.3. Des solutions au problème de réaction tardive

Deux solutions à ce problème sont possibles : augmenter les distances de sécurité et ralentir avant de freiner. La première solution est d'augmenter les distances de sécurité, mais le robot risque de s'arrêter très loin des obstacles et de ne pas trouver de passage là où il peut passer. C'est pourquoi, la deuxième solution est de ralentir le robot assez loin de l'obstacle et de freiner près de ce dernier pour s'orienter. Seul inconvénient est de trop surcharger le programme.

III.4. Exemples 3 (suivi d'un objet en mouvement pour ED-7273)

Dans ce dernier exemple, le robot doit suivre le mouvement d'un objet mobile perceptible par l'un de ces capteurs frontaux de *1* à *7*.

III.4.1. Principe

Le robot reste immobile jusqu'à ce qu'il détecte le mouvement d'un objet par l'un de ses capteurs frontaux. Puis, il s'oriente et se dirige vers ce dernier, tout en évitant les chocs, pour s'arrêter à une certaine distance de l'objet en question. On va supposer qu'un seul objet en mouvement. Dans le cas contraire, le suivi se fait sur le premier détecté.

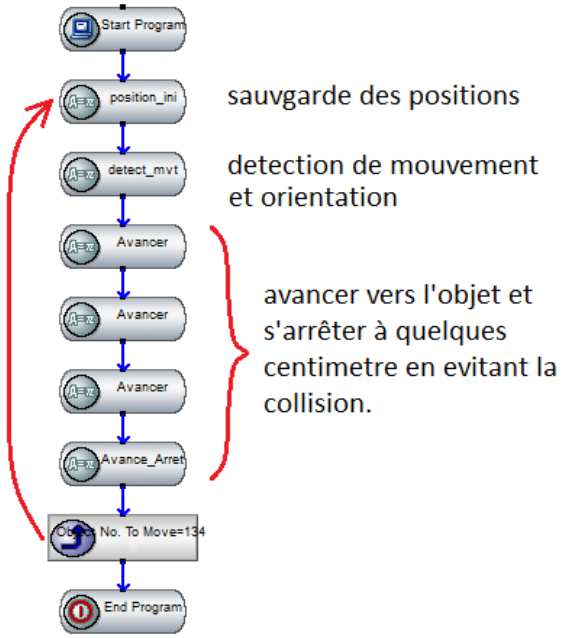


Figure III.17. *Logigramme suivit d'un objet en mouvement.*

III.4.2. Déclaration et sauvegarde des positions dans la fonction*« position_ini »*

Au début on n'avait pas posé de restriction sur la portée des capteurs, mais comme ces derniers affichaient, pour des grandes distances, des variations parfois supérieures à *100 cm*, on a décidé de se limiter à la détection de mouvement distant au-dessous de *120 cm.* Voici le programme *Script* utilisé dans cette fonction

```
SubScript()
SetTagVal"us1",GetTagVal("sensor1")
IfGetTagVal("us1")>120Then
SetTagVal"us1",120
EndIf
SetTagVal"us2",GetTagVal("sensor2")
IfGetTagVal("us2")>120Then
SetTagVal"us2",120
EndIf
SetTagVal"us3",GetTagVal("sensor3")
IfGetTagVal("us3")>120Then
```

```
SetTagVal"us3",120
EndIf
SetTagVal"us4",GetTagVal("sensor4")
IfGetTagVal("us4")>120Then
SetTagVal"us4",120
EndIf
SetTagVal"us5",GetTagVal("sensor5")
IfGetTagVal("us5")>120Then
SetTagVal"us5",120
EndIf
SetTagVal"us6",GetTagVal("sensor6")
IfGetTagVal("us6")>120Then
SetTagVal"us6",120
EndIf
SetTagVal"us7",GetTagVal("sensor7")
IfGetTagVal("us7")>120Then
SetTagVal"us7",120
EndIf
SetTagVal"return",1
EndSub
```

III.4.3. Détection de mouvement et orientation dans la fonction *« detect_mvt »*

Si l'un des capteurs détecte un mouvement, le robot s'oriente vers la direction de ce dernier et sort de la fonction. Sinon il scrute les capteurs en boucle.

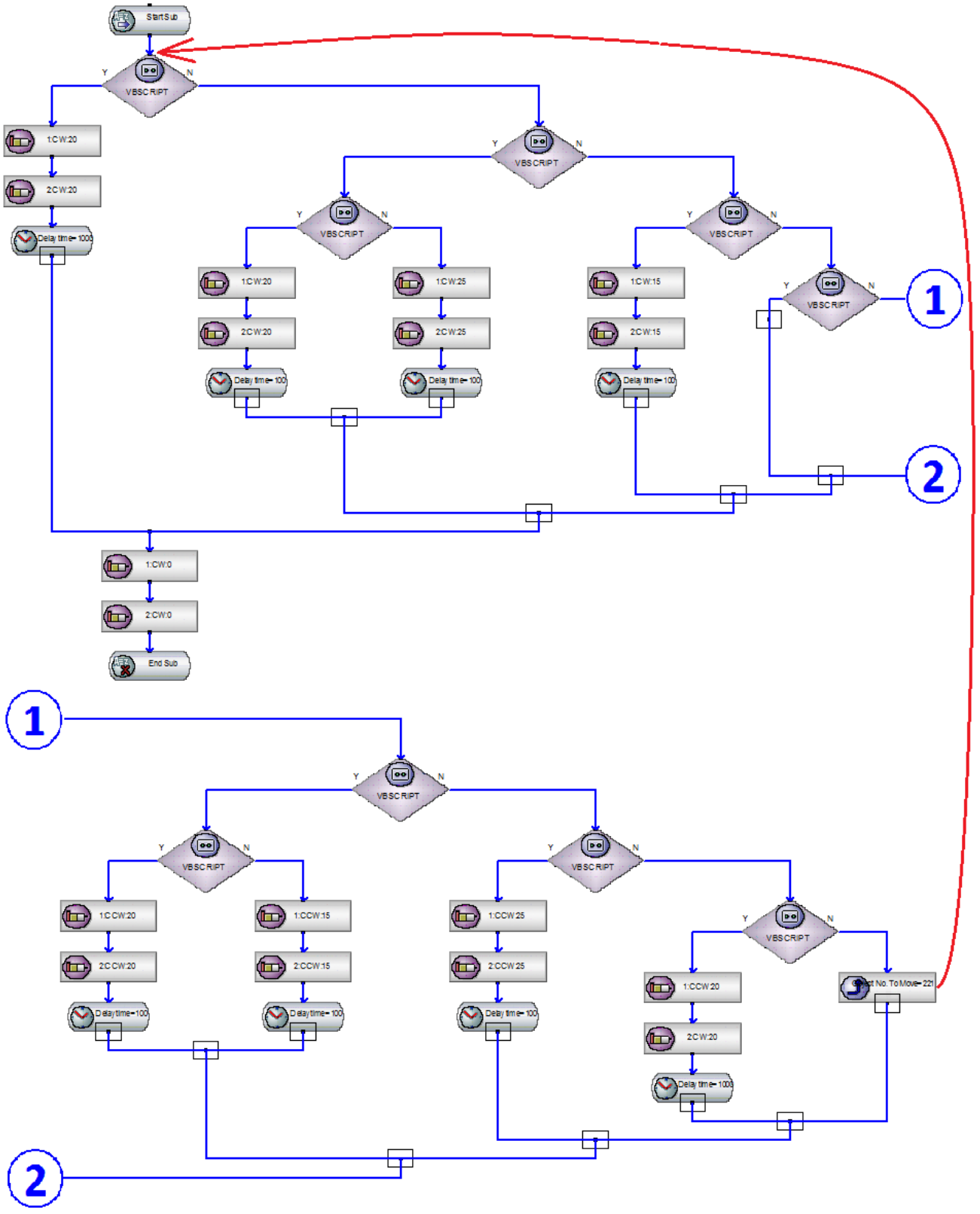


Figure III.18. *Fonction détection mouvement et orientation « detect_mvt ».*

A cause de restrictions imposées sur les grandes distances et des variations permanentes (± *2 cm*) dans la mesure, l'utilisation des *Script* est indispensable pour la surveillance des capteurs. Voici un exemple qu'on a appliqué pour le capteur *7*.

```
SubScript()
temp71=GetTagVal("sensor7")
If temp71>120Then
temp71 = 120
EndIf

temp72=GetTagVal("us7")
temp=temp72-temp71
temp0=0
If temp<=-3Then
temp0=1
EndIf
If temp>=3Then
temp0=1
EndIf

SetTagVal"return",temp0
EndSub
```

On lit la valeur mesurée par le capteur et on lui applique la restriction. On la compare à l'ancienne valeur et si on détecte une variation supérieure à *3 cm*, alors un objet s'est déplacé devant le capteur concerné (dans cet exemple le capteur 7).

III.4.4. Avancer en évitant la collision dans la fonction *« Avancer »*

Dans ce logigramme, on fait avancer le robot d'une distance de 25 *cm* vers l'objet et on l'arrête à *30 cm* de ce derniers ou dès qu'il y a risque de collision. Au début, on voulait l'intégrer dans une fonction « Timer » mais malheureusement cette dernière ne fait pas le comptage de temps, alors on a décidé d'utiliser des *« delay »* et de le déplacer en plusieurs fois.

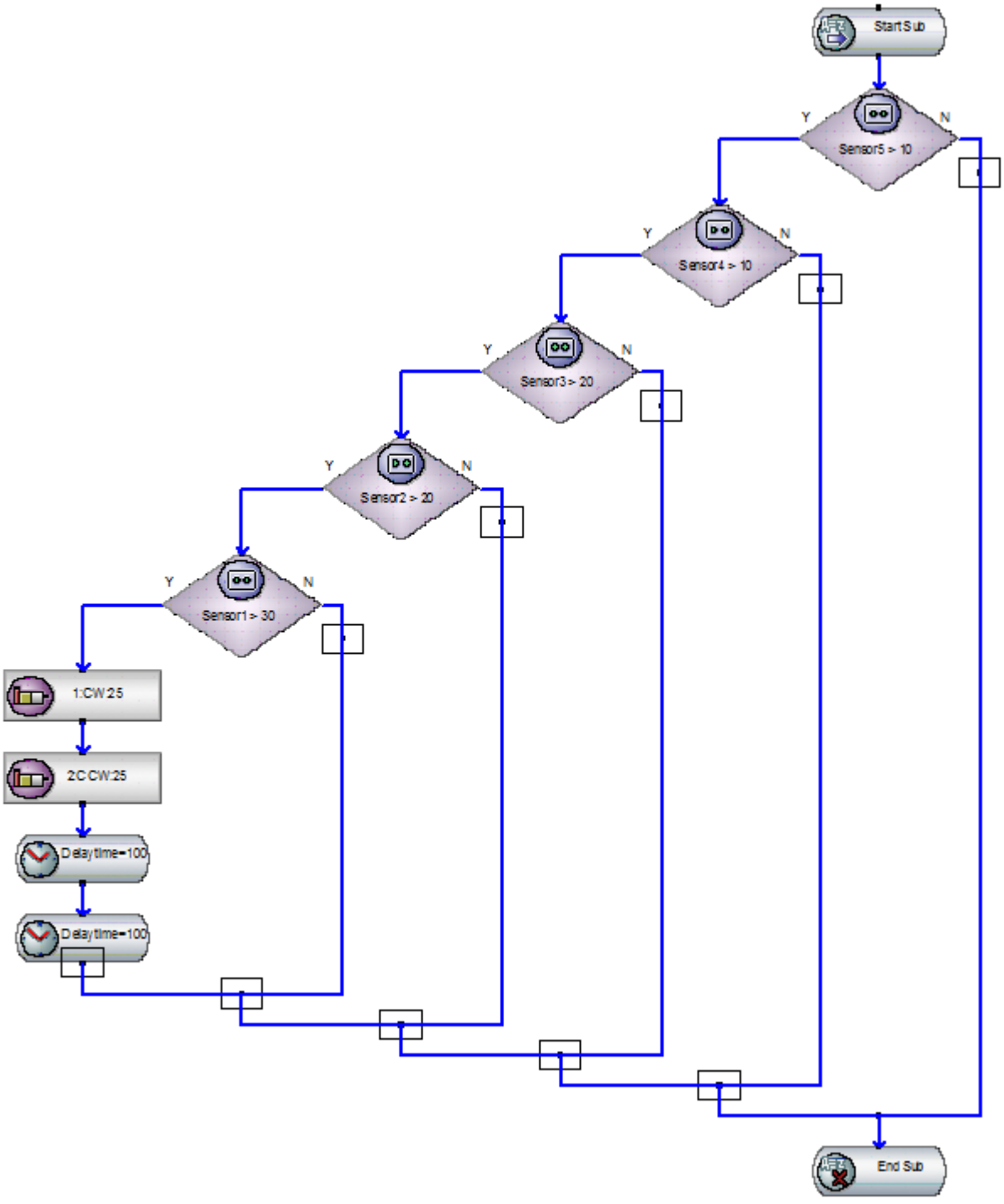


Figure III.19. *Fonction pour avancer en évitant la collision « Avancer »*

III.5. Conclusion

D'après le travail effectué et présenté dans ce chapitre, on a pu conclure ce qui suit :

Concernant IRES

- Toute distance lue par le capteur à ultrason, même affichée en tant que valeur réelle par logiciel « *Robot* » est comparée dans la cellule *« **Ultasonic »*** comme étant une valeur entière
- On peut augmenter la précision des capteurs ultrason en ajoutant *1.8 cm* à chaque valeur lue et comprise entre 3 et 27.
- Les distances lues par le capteur à infrarouge et affichées par « *Robot* » sont toutes entières et celles inferieures à *6 cm* ne sont pas bien détectées par ce dernier.
- On constate un mauvais fonctionnement des *timer*, on ne peut exploiter que les *delay*.

Concernant le robot

- Les distances mesurées par le capteur à ultrason sont influencés par la forme de l'obstacle, voir même sa visibilité.
- La nature du sol influe sur la navigation par odométrie
- Le temps de réaction (réponse) du robot qui est dû au mode de communication avec le microordinateur et avec ses différents modules (communication série).
- Pour réaliser une commande en temps réel, le temps est le facteur le plus important. Il est donc préférable de programmer avec des langages de bas niveau comme C++ et C, mais l'assembleur reste encore le plus adapté.

Chapitre IV : Développement d'applications pour ED-7273 avec les MFC sous visual C++

IV.1. Introduction

Dans ce chapitre on va s'intéresser au développement d'applications pour le robot *ED-7273* avec l'assistant ***MFC AppWizards*** de *visual C++*.

Pour commencer, on va jeter un coup d'œil sur quelques notions élémentaires de la programmation orientée objet en *C++*. Ensuite, on va présenter quelques fonctions des bibliothèques dynamiques fournies avec le logiciel *IRES*.

Enfin, on va développer quelques exemples de programmes de suivi de trajectoires pour le robot, en intégrant les bibliothèques fournies et en exploitants les fonctions déjà présentées.

IV.2. Notions de base

Dans cette section, on va d'abord donner un bref aperçu sur quelques notions de base de la programmation orientée objet en *Visual C++*, sous Windows.

IV.2.1. Les MFC (*Microsoft Foundation Classes*)

Il s'agit d'une bibliothèque de classes *C++* qui encapsulent les différentes fonctions, notamment de gestion des fenêtres ou de messages, propres à Windows. On dit aussi que les classes *MFC* encapsulent l'*API* (*Application Program Interface*, une méthode lourde, qui exige un développement de nombreuses lignes de code avant d'aboutir à un résultat concret). Avec l'assistant de *MFC AppWizards*, ces classes sont plus simples à utiliser. Car, une grande partie du code est générée automatiquement. Il suffit de spécifier le traitement à effectuer et de compléter pour le personnaliser **[Ber 07].**

IV.2.2. Notion de classe et d'objet

- *Une classe :* est le plan principal pour créer des objets. Elle est créée pour décrire les attributs et les méthodes de toute une famille d'objets. Une classe définit en particulier un type d'objet.
- *Un objet :* est une instance de classe, c'est donc une entité qui regroupe des données membre (les attributs) et des fonctions membre (les méthodes).
- *Les méthodes :* sont des fonctions qui s'appliquent à l'objet et à ses attributs. Elles sont généralement le seul moyen pour accéder aux attributs d'un objet. Puisque, ces derniers sont souvent déclarés privés «*private :*» **[Web 4]**.

IV.2.3. Droits d'accès et encapsulation

Pour se conformer au principe d'encapsulation, il est possible de masquer des attributs et des méthodes au sein d'un objet. Pour cela, on utilise les mots clés *« private* : » et « *public :* ». Les membres qui sont déclarés privés (avec *private*) ne sont accessibles que par les méthodes de l'objet lui-même et les méthodes des objets de la même classe. Tandis que, les membres public sont visibles pour tous **[Neb 11].**

IV.2.4. Notions supplémentaires

- ✓ Lorsqu'on créé un projet en *visual C++*, on remarque que des fichiers *« .h »* et *« .cpp »* sont automatiquement générés.
- ✓ Un header (fichier **.h*) contient la déclaration de la classe avec les attributs et les prototypes des méthodes. Il est recommandé de ne jamais mettre la directive *« using namespace std; »*, il faut rajouter le préfixe *« std:: »* devant chaque *string*.
- ✓ Un fichier source (fichier **.cpp*) contient la définition des méthodes et leurs implémentations.
- ✓ Un constructeur est une méthode ayant le même nom que sa classe. Elle est appelée automatiquement à chaque fois que l'on crée un objet basé sur cette classe. C'est dans cette méthode que l'on peut définir les valeurs par défaut et imposer des restrictions, lors de la construction de l'objet.
- ✓ Un destructeur est une méthode qui commence par un tilde *« ~ »* suivi du même nom que sa classe et qui ne peut prendre aucun paramètre en entrée. Cette méthode est automatiquement appelée lorsqu'un objet est détruit, par exemple à la fin de la fonction dans laquelle il a été déclaré ou lors d'un « *delete »* si l'objet a été alloué dynamiquement avec « *new »* **[Neb 11]**.

IV.2.5. Intégrer des bibliothèques « *.lib » en *Visual C++*

Concevoir des applications en *Visual C++* pour contrôler le robot ED-7273 exige l'intégration de toutes les bibliothèques nécessaires. Pour cela on va dans la barre des menus, dans « *projet⟶ setting»* après dans la fenêtre qui s'ouvrira on va à l'onglet « *Link »* et, enfin, on indique le nom complet du fichier et son emplacement dans le champ de saisie *« object/library modules : »*. Pour intégrer plusieurs bibliothèques, il suffit de les indiquer séparées par des espaces (***Figure. IV.1***).

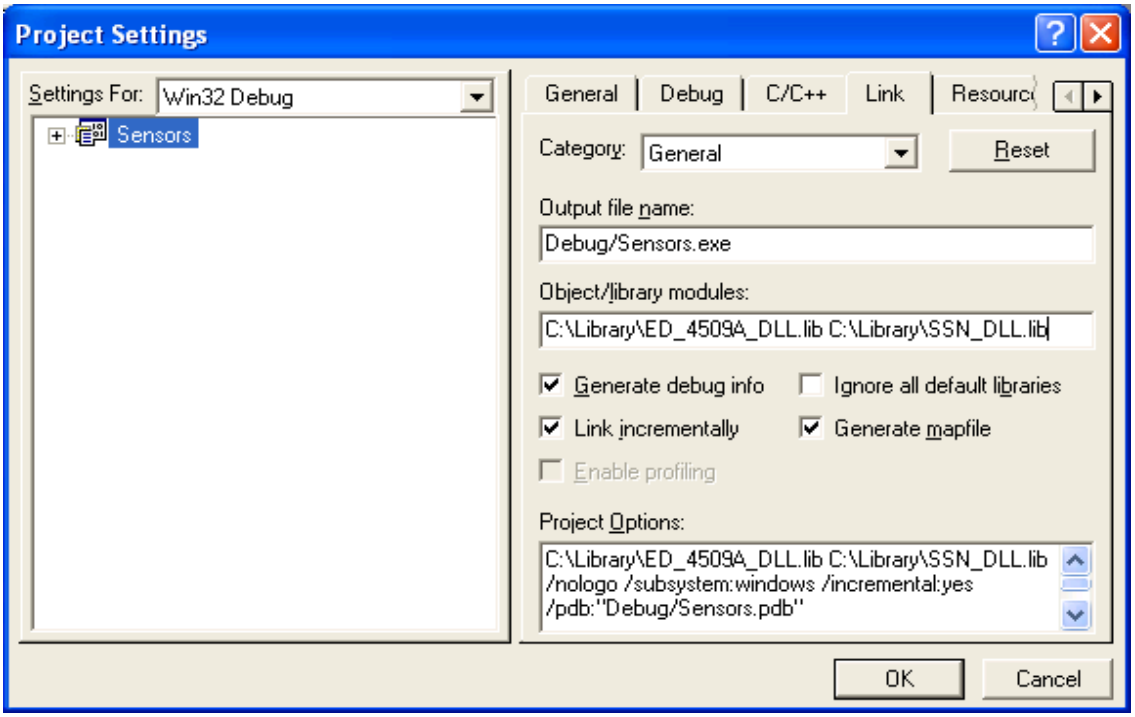


Figure IV.1.*Intégration de deux bibliothèques « *.lib » en Visual C++*

IV.3. Quelques fonctions des bibliothèques *C++* fournies avec IRES

Tout d'abord, on va présenter quelques fonctions utiles (intégrées dans les bibliothèques dynamiques fournies avec le logiciel *IRES*) pour le contrôle et la commande, des capteurs et du moteur à courant continu (DC-Motor).

IV.3.1. Quelques fonction de la bibliothèque « Motion_Library.h »

- ***DCServo01_Init(HWND m_hWnd, MOTION_SESSION *pMotion) :*** Cette fonction doit être appelée au moins une fois pour pouvoir utiliser les autres fonctions qui contrôlent le moteur. Comme le moteur à courant continu est commandé par le port USB, cette fonction permet d'initialiser la connexion. Elle retourne la valeur *« EDROBOT_API_OK »* si elle s'est bien exécuté (initialisation bien faite). Sinon, elle retourne la valeur *« EDROBOT_API_ERROR »*.
  Le paramètre *m_hWnd* est un *handle* pour l'objet de fenêtre d'application
  Le paramètre *pMotion* est un objet de la classe *MOTION_SESSION*, qui doit être déclarée auparavant dans le *header* du fichier dialogue *« *Dlg.h »*. C'est comme un *handle* de session pour commander le moteur *DCServo* . Il renvoie les informations du *handle* de la session au cours du processus initial.

- ***DCServo01_ReadPosition(MOTION_SESSION pMotion, unsigned char nSelectMotor, double &dbPosition):*** Cette fonction permet de lire la position du servomotor.
  nSelectMoto : elle prend les valeurs *1*, 2, 3 ou *4*, pour désigner le numéro du moteur.
  dbPosition : c'est le paramétré de sortie de type double qui contient la valeur de la position du moteur.

- ***DCServo01_ReadVelocity(MOTION_SESSION pMotion, unsigned char nSelectMotor, double &dbVelocity):*** Cette fonction est utilisée pour lire la vitesse (en *tours/mn*) du moteur sélectionné.

dbVelocity : c'est le paramètre de sortie, qui contient la vitesse actuelle du moteur.

- ***DCServo01_ReadRPM (MOTION_SESSION pMotion, unsigned char nSelectMotor, double &dbRPM):*** C'est une autre fonction pour lire la vitesse (fréquence en *tours/mn*) du moteur sélectionné.

dbRPM : c'est le paramètre de sortie qui contient la fréquence actuelle en *tour/ mn*.

- ***DCServo01_StopSmoothly (MOTION_SESSION, pMotion, unsigned char nSelectMotor):***

Cette fonction est utilisée pour arrêter doucement le moteur sélectionné.

- ***DCServo01_StopAbruptly (MOTION_SESSION, pMotion, unsigned char nSelectMotor):***
  Une autre fonction utilisée pour arrêter le moteur, mais de manière brusque.

- ***DCServo01_SetAbsoluteVelocity(MOTION_SESSION pMotion, unsigned char nSelectMotor, double dbVelocity):*** Cette fonction permet de fixer la valeur de la vitesse du moteur *en tour/mn*.

dbVelocity : c'est la variable qui contient la valeur de la vitesse (*en tour/mn*) a affecté au moteur.

- ***DCServo01_SetRelativeVelocity(MOTION_SESSION pMotion, unsigned char nSelectMotor, double dbVelocity):*** Permet d'incrémenter la vitesse (fréquence *tours/mn*) du moteur.

dbVelocity : c'est la variable qui contient la valeur relative de la vitesse.

- ***DCServo01_ForwardStart(MOTION_SESSION pMotion, unsigned char nSelectMotor):***

Cette fonction permet de faire tourner le moteur dans le sens des aiguilles de la montre *CW* (*Clock-Wise*).

- ***DCServo01_BackwardStart(MOTION_SESSION pMotion, unsigned char nSelectMotor):*** Elle fait tourner le moteur dans le sens trigonométrique *CCW* (contre horaire).

Toutes ces fonctions retournent *« EDROBOT_API_OK »* quand elles se sont bien exécutées. Il ne faut pas oublier d'inclure la bibliothèque *« ED_4509A_DLL.lib »*, de copier la bibliothèque « ED_4509A_DLL.dll » dans le dossier du projet et d'inclure le *header* (*#include"Motion_Library.h"*) au fichier « *StdAfx.h* » du projet.

IV.3.2. Quelques fonctions de la bibliothèque *« SSN_Library.h »*

- ***SSN_Initialize(SSN_SESSION *pSensor,BOOL bAutoSerch):*** C'est la fonction qui permet d'initialiser la communication entre le PC et les capteurs.

bAutoSerch : il prend deux valeurs logiques, en cas de saisie de *FALSE*, connecter le dernier port COM enregistré à la résistance. Dans le cas contraire *TRUE*, connectez le COM3.

Le système de capteur, auquel la communication série (*RS-232*) est appliquée, nécessite le processus d'initialisation du port série, mis à l'intérieur de la bibliothèque. Ça ne nécessite pas d'autres configurations par l'utilisateur sauf dans le cas de l'utilisation du port COM3.

- ***SSN_SetUltrasonic(SSN_SESSION pSensor , unsigned char*SelectSensor, unsigned char nSelectCnt):*** Cette fonction est utilisée pour définir les capteurs à utiliser et leurs ordres de lecture.

SelectSensor : contient les numéros des capteurs à ultrasons définis par l'utilisateur, pour la lecture de leurs données, dans l'ordre de leurs inscriptions.

nSelectCnt : Entrez le nombre des capteurs à ultrasons désignés.

- ***SSN_GetUltrasonic(SSN_SESSION pSensor, float *SensorData):*** C'est la fonction utilisée pour retourner les données des capteurs dans le vecteur *SensorData.*

SensorData : vecteur de sorties, désigné par l'utilisateur, pour contenir les données des capteurs à ultrasons.

.

IV.4. Programme contrôle et commande de DC-Motor

Dans ce programme, on va créer une fenêtre de dialogue avec des boutons, des boites de dialogue et du texte statique, pour le contrôle et la commande des moteurs. En se basant sur les exemples fournis dans le *Manuel ED-7273*, on va tester quelques fonctions de la bibliothèque *« Motion_Library.h »*.

IV.4.1. Implémentation du code en *C++*

Pour voir l'essentiel du code qu'on a implémenté voir *«Annexe.B.1»*. Des commentaires y sont insérés. Pour ne pas encombrer et garder l'enchainement dans ce chapitre, on a annexé ce code, même si ça représente la partie la plus importante qui concerne les fonctions liées au contrôle du robot.

IV.4.2. Les résultats obtenus

Après compilation, édition des liens, construction et exécution du programme, apparait la fenêtre de dialogue représenté dans la ***Figure IV.2.***

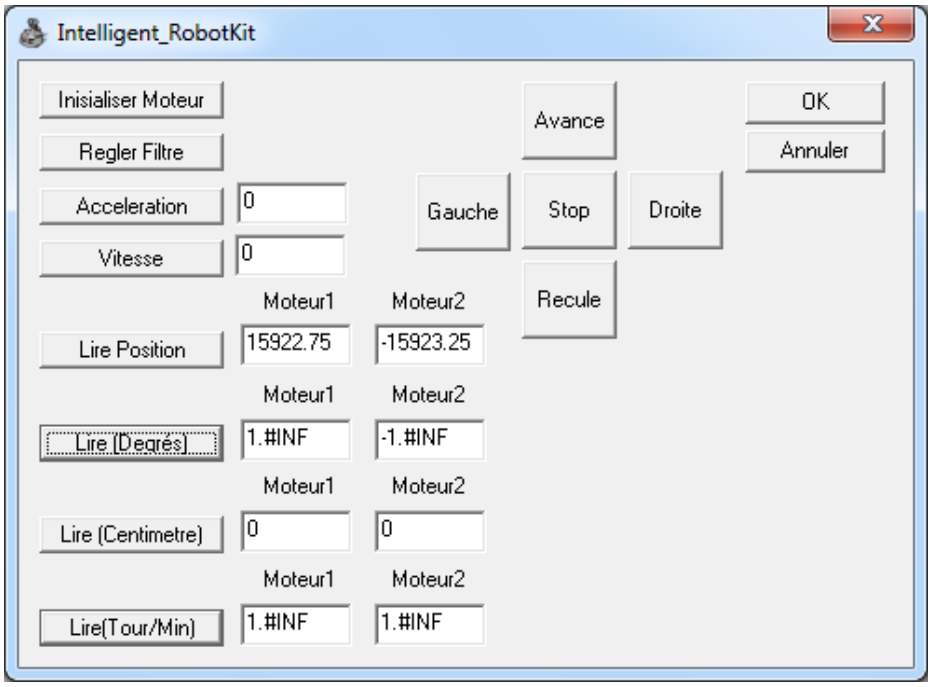


Figure IV.2. *Fenêtre d'exécution du premier programme*

Au début, pour tester le bouton *« Initialiser Moteur »*, on a d'abord appuyé sur les boutons disposé en croix pour lancer le robot (les moteurs), mais ces boutons n'avaient aucun effet jusqu'à ce qu'on appuie sur le bouton *« Initialiser Moteur »*. Là, on a pu le faire avancer, reculer, aller à droite, aller à gauche et l'arrêter.

Ensuite, on est passé aux boutons *« Acceleration »* et *« Vitesse »*. On a affecté la valeur *100* pour l'accélération et *10 tours/mn* pour la vitesse. On a constaté que l'appuie sur ces deux boutons n'a aucun effet. D'ailleurs, même dans le manuel de *ED-7273*, ils ont indiqué que le paramètre introduit dans la fonction *« DCServo01_SetAbsoluteAcceleration »* est considéré comme étant une vitesse.

Apres, vient le tour du boutons *« Lire Position »*, pour afficher la position en nombre d'impulsions émises par l'encodeur optique. L'appuie successif sur ce bouton fait incrémenter le nombre d'impulsions affichées.

Enfin, le reste des boutons qui utilisent des fonctions liées aux encodeurs optiques, soit qu'ils affichent des valeurs illisibles *«1. #INF»* ou *« 1.QNAN »* (problème de format d'affichage ou mal utilisation de pointeur) et c'est le cas aussi des boutons *«Lire (Tour/Min) »* et *« Lire (Degrés) »* Le bouton *« Lire (Centimètre) »* n'affiche rien.

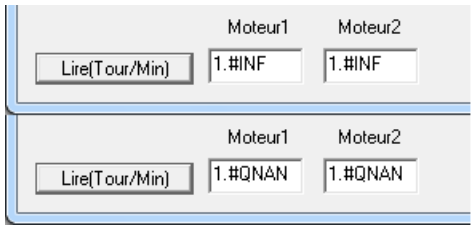


Figure IV.3. *Disfonctionnement d'affichage.*

D'après le travail fait dans ce premier programme et tous les tests qu'on a effectué, on constate un mauvais fonctionnement des fonctions suivantes :

- *DCServo01_SetAbsoluteAcceleration*
- *DCServo01_SetAbsoluteVelocity*
- *DCServo01_ReadDegree*
- *DCServo01_ReadCentimeter*
- *DCServo01_ReadRpm*

Malheureusement, on n'a pas accès à leurs définitions détaillées. Puisque, les bibliothèques sont fournies sous formes de fichiers *« *.lib »* ou *« *.dll »*.

Comme on prévoit de faire des programmes de planification de trajectoires, avoir accès et agir sur les vitesses est une priorité. Alors, on a décidé de tester le reste de ces fonctions sur la vitesse.

IV.4.3. Tests des fonctions pour le contrôle de vitesse

Tout d'abord, on teste la fonction *« DCServo01_ReadVelocity »* pour lire la vitesse des moteurs, en remplaçant le code du bouton *«Lire (Tour/Min) » par :*

```
//=====================================================================;
// aller vers l'avant. =========================;
DCServo01_ForwardStart(pMotion, 1 ); // cw: moteur1 "gauche"
DCServo01_BackwardStart(pMotion, 2 ); // ccw: moteur1 "droite"
// ============================================;
// lecture de la vitesse du DC ServoMotor 1. ===;
if( DCServo01_ReadVelocity(pMotion, 1, m_ReadRPM) != EDROBOT_API_OK )
      ::MessageBox( NULL, "Erreue de Lecture de Vitesse!!", "FunctionError",
MB_OK );
// ============================================;
// lecture de la vitesse du DC ServoMotor 2. ===;
if( DCServo01_ReadVelocity(pMotion, 2, m_ReadRPM2) != EDROBOT_API_OK )
      ::MessageBox( NULL, "Erreue de Lecture de Vitesse!!", "FunctionError",
MB_OK );
// ============================================;
UpdateData(FALSE);
//=====================================================================;
```

Ensuite, on passe à la fonction *« DCServo01_SetRelativeVelocity»* pour changer la vitesse des moteurs, en remplaçant le code du bouton*« Vitesse »* par :

```
//=====================================================================;
UpdateData(TRUE);
// Incrementer la vitesse du DC ServoMotor.1 ====;
if( DCServo01_SetRelativeVelocity( pMotion, 1, m_Velocity ) != EDROBOT_API_OK )
      ::MessageBox( NULL, "Erreue de Lecture de Position!!", "FunctionError",
MB_OK );
// ============================================;
// Incrémenter la vitesse du DC Servo Motor.2 ===;
if( DCServo01_SetRelativeVelocity( pMotion, 2, m_Velocity ) != EDROBOT_API_OK )
      ::MessageBox( NULL, "Erreue de Lecture de Position!!", "FunctionError",
MB_OK );
// ============================================;
UpdateData(FALSE);
//=====================================================================;
```

Le clic sur les deux boutons créés précédemment, bien sûr après l'initialisation des moteurs, n'avait rien donné. Alors ces deux fonctions ne sont pas exploitables.

IV.5. Programme contrôle des capteurs à ultrasons

Dans ce deuxième programme, on veut tester les fonctions de la bibliothèque « *SSN_DLL.lib* » concernant les capteurs. Pour cela, on va faire simple : juste une fenêtre, un seul bouton et une seule boite de dialogue.

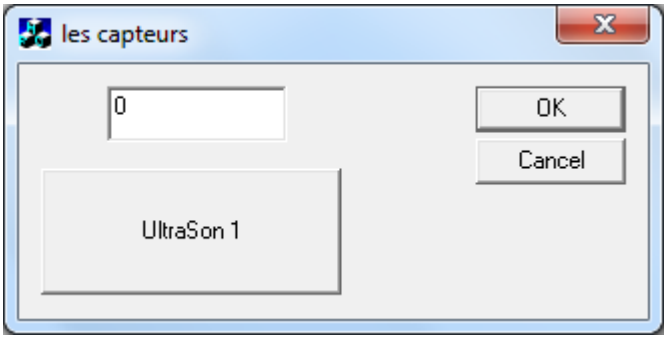


Figure IV.4. *Fenêtre de dialogue programme 2*

IV.5.1. Le code utilisé

Le code utilisé est listé ci-dessous :

```
voidCLescapteursDlg::OnBtUltraSon1()
{
//    définir l'ordre des capteurs===============;
      nSelectSensor[0] = 1;
      nSelectSensor[1] = 2;
      nSelectSensor[2] = 3;
      nSelectSensor[3] = 4;
      nSelectSensor[4] = 5;
      nSelectSensor[5] = 6;
      nSelectSensor[6] = 7;
      nSelectSensor[7] = 8;
      nSelectSensor[8] = 9;
//    Initialiser les capteurs.==================;
      If( SSN_Initialize(&pSensor, FALSE)!= EDROBOT_API_OK )            //motor
init
            ::MessageBox( NULL, "Erreur d'Initialisation Capteurs!!",
"FunctionError", MB_OK );
      SSN_SetUtrasonic(pSensor, nSelectSensor, 8 );          // Utrasonic set
      SSN_Start(pSensor);     // demarer le capteurs
//    ================================;
//    afficher la valeur lu du capteur dans la boite de dialogue
      SSN_GetUtrasonic(pSensor, fUltrasonic );
      m_UltraSon_1 = fUltrasonic[0];
      UpdateData(FALSE);
}
```

IV.5.2. Interprétation des résultats

Apres exécution de ce programme et clic sur le bouton « *UltraSon 1*», on voit que le programme s'interrompe et affiche le message d'erreur, comme dans la figure ci-dessous.

Figure IV.5. *Fenêtre du message d'erreur du deuxième programme.*

On a pensé à une erreur de segmentation de la mémoire, alors on a retiré la ligne *« nSelectSensor[8] = 9; »* du programme en la mettant en commentaire, mais ça ne règle pas le problème. On a décidé de faire une initialisation manuelle en utilisant la fonction *« SSN_ManualInitialize (SSN_SESSION *pSensor, intnPort) »* sans pour autant résoudre ce problème. Pourtant on avait choisi les numéros des derniers ports COM reconnus (enregistrés) après qu'on ait connecté le robot au PC.

Le fait qu'on n'ait pas pu faire fonctionner cette fonction d'initialisation de la communication, des capteurs à ultrasons, on ne pourra plus exploiter le reste des fonctions sur ces capteurs.

IV.6. Programme pour trajectoires en formes géométriques

Dans ce troisième programme, on va faire suivre au robot des trajectoires en formes géométriques telles que le rectangle, triangle et cercle. Au début, on voulait exploités les fonctions des deux bibliothèques *« ED_4509A_DLL.lib »* et *« SSN_DLL.lib »* qui concerne les moteurs et les capteurs.

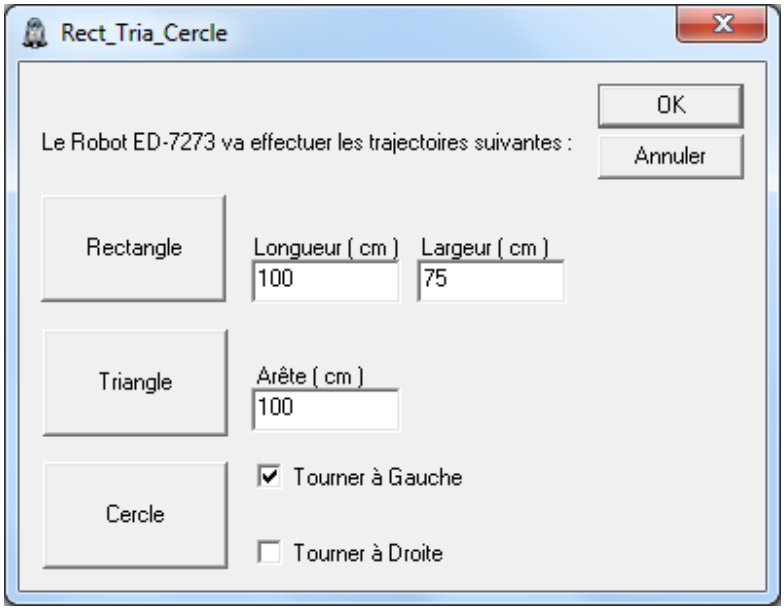


Figure IV.6. *Fenêtre de dialogue troisième programme*

IV.6.1. Implémentation du code en *C++*

Comme pour le premier exemple, l'essentiel du code qu'on a implémenté est dans *« AnnexeB.2»*.

IV.6.2. Les résultats obtenus

Pour tester le programme, on a d'abord noté les valeurs désirées pour définir les trajectoires (*100 cm* pour *Longueur*, *75 cm* pour *Largeur* et *100 cm* pour *Arête*). On a ensuite tracé les trajectoires prévues sur le sol et placer le robot sur le point de départ. Enfin, en appuyant sur les boutons de la fenêtre, on a lancé le robot sur les deux premières trajectoires et pris les photos suivantes.

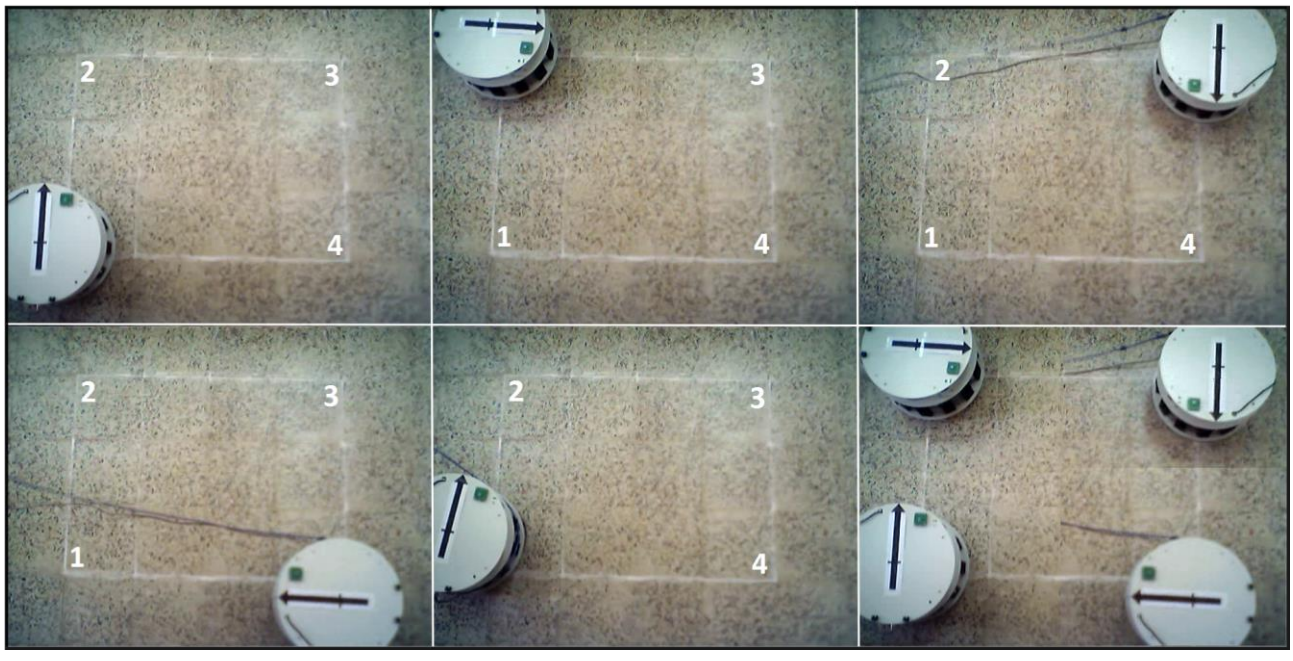


Figure IV.7. *Exécution de la trajectoire rectangulaire*

La ***Figure IV.7*** montre que le robot suit bien la trajectoire prévue. Néanmoins, on remarque un petit décalage dû à la manière d'arrêter le robot. En changeant dans le code toutes les commandes « *DCServo01_StopSmoothly* » par « *DCServo01_StopAbruptly* », on a pu améliorer la précision sur la trajectoire, bien qu'on ne pourra jamais la suivre de manière parfaite ; vu l'état du sol et la mesure imprécise de la vitesse du robot.

Figure IV.8. *Amélioration de la précision sur la trajectoire rectangulaire*

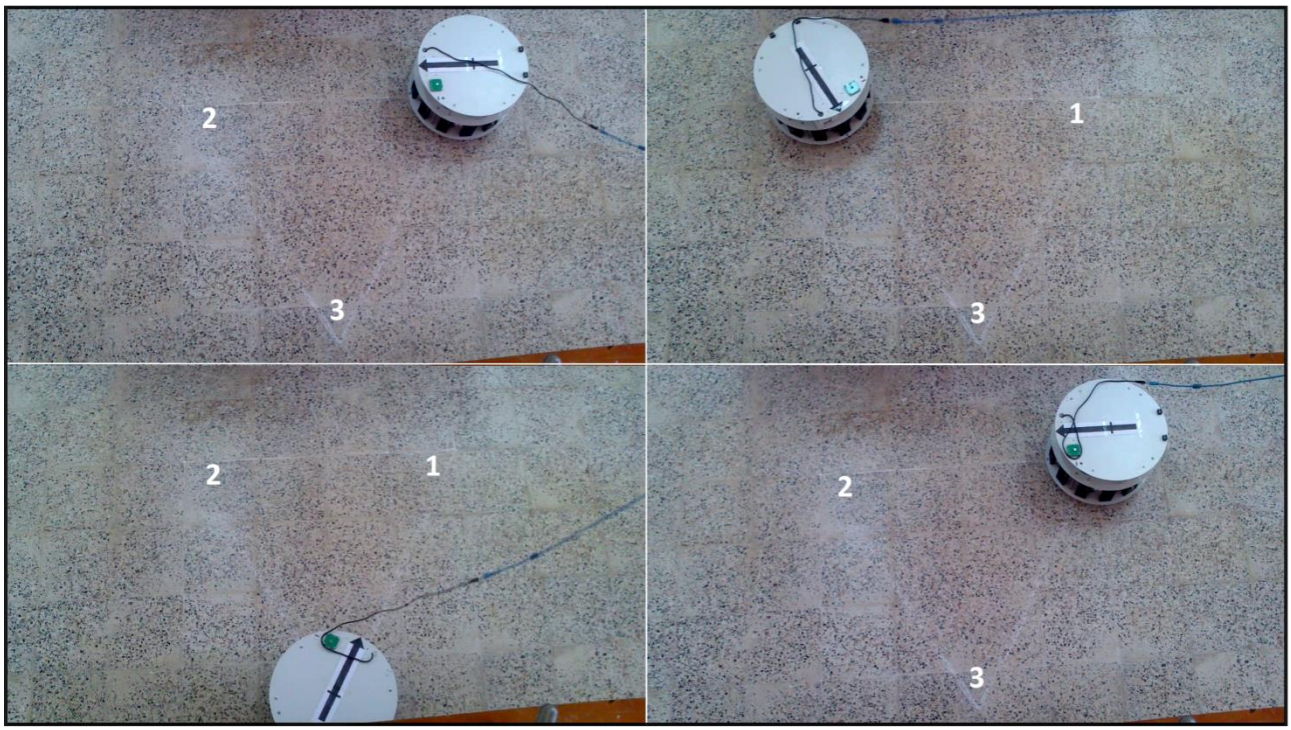


Figure IV.8. *Exécution de la trajectoire triangulaire*

La ***Figure IV.8*** nous montre l'exécution de la trajectoire triangulaire, après l'utilisation de l'arrêt brusque des moteurs.

Enfin, on passe à la réalisation de la trajectoire circulaire. Comme on n'a pas le contrôle sur la vitesse des moteurs, on ne peut pas maîtriser le rayon de rotation. Tout ce qui nous reste est de tourner autour d'une roue (un rayon de *24 cm*), mais on peut tout de même choisir un sens de rotation.

IV.7. Programme pour trajectoires polygonales

Dans ce quatrième programme, on va faire suivre au robot un parcours polygonal. Tout d'abord, on sélectionne un sens de rotation. Puis, le programme nous donne le choix entre deux différents parcours : soit en en donnant directement le nombre et la langueur d'arête ou bien en donnant le nombre d'arêtes le rayon du cercle qui le délimite.

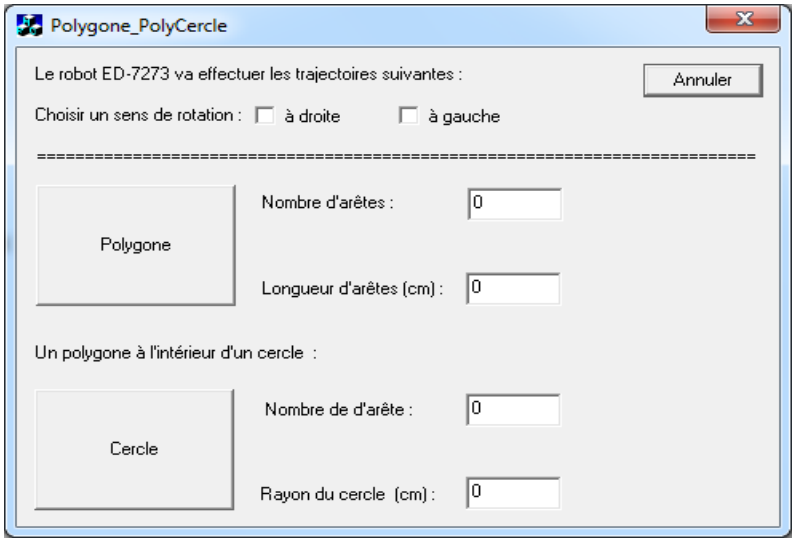


Figure IV.9. *Fenêtre de dialogue quatrième programme*

IV.7.1. Implémentation du code en *C++*

Comme pour le premier exemple, l'essentiel du code qu'on a implémenté est dans *«Annexe B.4»*.

IV.7.2. Interprétation des résultats

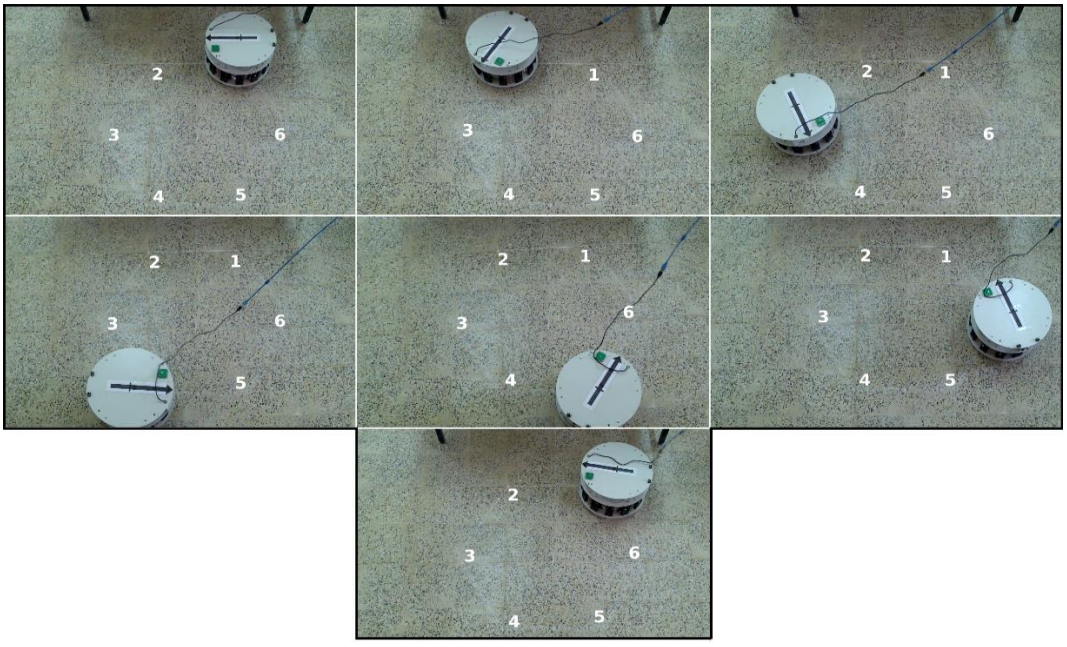


Figure IV.9. *Exécution de la trajectoire hexagonale*

On voit bien que la trajectoire c'est bien exécutée avec un léger décalage.

IV.8. Conclusion

Avec ***MFC AppWizards***, il est plus facile de faire de la programmation orientée objet avec le *C++*. Le langage *C++* offre une meilleure prise en main du robot. Si seulement les bibliothèques fournies sont plus exploitables, on aurait pu faire la détection et l'évitement d'obstacles pour une navigation autonome. Avec des moyens limités, on a quand même pu faire quelques fonctions sur la planification de trajectoire.

Avons de conclure ce dernier chapitre, on va faire un listing des fonctions qui ne marchent pas, en plus de toutes les fonctions qui agissent sur les capteurs, à savoir :

- *DCServo01_ReadVelocity*
- *DCServo01_SetAbsoluteVelocity*
- *DCServo01_SetRelativeVelocity*
- *DCServo01_ReadDegree*
- *DCServo01_ReadCentimeter*
- *DCServo01_ReadRpm*
- *DCServo01_SetAbsolutAcceleration*

Conclusion générale

Conclusion générale

Ce travail a exigé beaucoup de documentation sur la robotique et sur la programmation, notamment la programmation orientée objet et la robotique mobile.

D'abord, on a introduit des notions sur la robotique mobile telles que la locomotion, la perception, la localisation et la navigation. On a aussi vu que la robotique mobile englobe plusieurs disciplines et a vu d'énorme progrès.

Le deuxième chapitre expose les différents outils qu'on a utilisé, à savoir les deux robots *ED-7271* et *ED-7273* ainsi que l'environnement de développement (*IRES*). On a, notamment, détaillé les différents modules des deux robots ainsi que l'installation et la présentation du logiciel *IRES*.

Le troisième chapitre présente la première partie de notre travail. D'abord, on s'est intéressé à l'exploitation des capteurs télémétriques et des moteurs pour élaborer des programmes d'évitement d'obstacles et de suivi d'objets. On a mieux assimilé le fonctionnement d'*IRES*. On a, surtout, testé la programmation en mode *Script*. Malgré la commodité qu'offre ce mode, *IRES* reste limité pour développer des applications avancées (navigation, planification,…) ; du fait de son jeux d'instructions très limité. Comme on a pu faire un constat sur le mauvais fonctionnement des capteurs à ultrason qui, influencés par l'état du sol et la nature des obstacles, affichent des valeurs instables.

La dernière partie de notre travail présentée dans le chapitre IV, concerne la programmation du robot ED-7273 avec les *MFC*. On a fait un bilan sur les fonctions des bibliothèques pour servomoteurs à courant continu et les capteurs à ultrasons. On a pu développer des applications de commande et de planification de trajectoires. On est arrivé au résultat probant de quelques programmes de suivi de trajectoires, malgré le non fonctionnement de quelques fonctions, notamment, les problèmes d'initialisation du robot et des capteurs ultrasonores.

Malgré les problèmes liés aux outils utilisés (robot et logiciel), on a pu réaliser un travail satisfaisant. Mais, il reste à exploiter plein d'autres fonctions, une fois que les problèmes sont résolus. Etant les premiers à avoir travaillé sur ces deux robots, on espère que notre étude servira pour de futurs travaux sur ces derniers.

Références bibliographiques

Références bibliographiques

[Ahr 10] I. Ahriz Roula**, «** *Application des techniques d'apprentissage à la géolocalisation par radio fingerprint,*» Docteur de L'université Pierre et Marie Curie, Paris 6, 2010.

[Ana 12] E. F. Anas, *« Navigation Globale d'un Fauteuil Roulant Motorise dans de Grands Espaces Intérieurs, »* Mémoire de Maitrise en Sciences Appliquées, Ecole Polytechnique de Montréal, 2012.

[Avi 05] J. G. Avina Cervantes, « *Navigation visuelle d'un robot mobile dans un environnement d'extérieur semi-structuré, »* Thèse Doctorat de l'Institut National Polytechnique de Toulouse, 2005.

[Bay 10] B. BAYLE, « Cours de *Robotique mobile,*» Université de Strasbourg, 2010.

[Bea 01] J. Beaudry « *Conception et Contrôle d'un Robot Mobile à Vitesses Différentielles,*» Mémoire de fin d'études pour l'obtention du diplôme d'ingénierie, Université de Montréal, 2001.

[Bel 11] I. Belkhouche, « *Planification de la trajectoire et évitement d'obstacle par les réseaux de neurones pour les robots mobile autonome,* » Mémoire de Master en Informatique, Université de Tlemcen, 2011.

[Ber 07] C. Bertrand, « *Cours Programmes Windows sous Visual C++ avec les MFC,* » Institut des Sciences et Techniques de l'Ingénieur d'Angers, 2007.

[Che 14] L. Cherroun, « *Navigation autonome d'un robot mobile par des techniques neuro_floues,* » Thèse de Doctorat en Automatique**,** Université de Biskra, 2014.

[Clo 07] R. Cloutier, « *Conception électronique et informatique d'un robot mobile pour usage dans un environnement domiciliaire,* » Mémoire de Maitrise à Université De Sherbrooke (Québec) Canada, 2007.

[Dav 13] F. David, « *Cours Robotique Mobile,* » Ecole Nationale Supérieure de Techniques Avancées Paris, 2013.

[Ham 12] O. El Hamzaoui, *« Localisation et cartographie simultanées pour un robot mobile équipé d'un laser à balayage,»* Thèse de Doctorat de l'École Nationale Supérieure des Mines à Paris, 2012.

[Lef 06] O. Lefebvre, « *Navigation autonome sans collision pour robots mobiles non holonomes,»* thèse Doctorat de l'Institut National Polytechnique de Toulouse, 2006.

[Mal 11] F. Malartre, *« Perception intelligente pour la navigation rapide derobots mobiles en environnement naturel,* » Thèse Doctorat à Université Blaise Pascal - Clermont Ii, 2011.

[Man 71] « *Robot Assembly Kit Trainer ED-7271User's Manuel*» ED corporation, Korea, 3rd edition, Nov.2010.

[Man 73] « *Intelligent Robot Kit, ED -7273 Manuel*» 4th edition, Jan.2012.

[Men 14] B. MENDIL, Cours de Robotique, Université Targa ouzemour Bejaia, 2014.

[Neb 11] Mathieu Nebra et Matthieu Schaller **«** *Programmez avec le langage C++* **»**, www.siteduzero.com, 2011.

[Pey 06] T. Peynot, « *Sélection et contrôle de modes de déplacement pour un robot mobile autonome en environnements naturels*, » Thèse de Doctorat de l'Institut National Polytechnique de Toulouse, 2006.

[Sli 05] N. Slimane, « *Système de localisation pour robots mobiles*, » Thèse de Doctorat Université de Batna, 2005.

[Zid 09] G. ZIDANI, *« Exécution de trajectoire pour robot mobile d'intérieurs - réseaux de Neurones,* » Mémoire de Magister en Robotique, Université de Batna, 2009.

[Web 1] http://fr.wikipedia.org/wiki/Robot#Historique

[Web 2] http://locomotionhorsnormes.e-monsite.com/pages/robots/locomotion-et-moyens-de-franchissement-des-robots-terrestres-et-planetaires.html

[Web 3] http://fr.wikipedia.org/wiki/Navigation

[Web 4] http://frog.isima.fr/antoine/base.shtml

Annexes

Annexe A

Annexe A.1

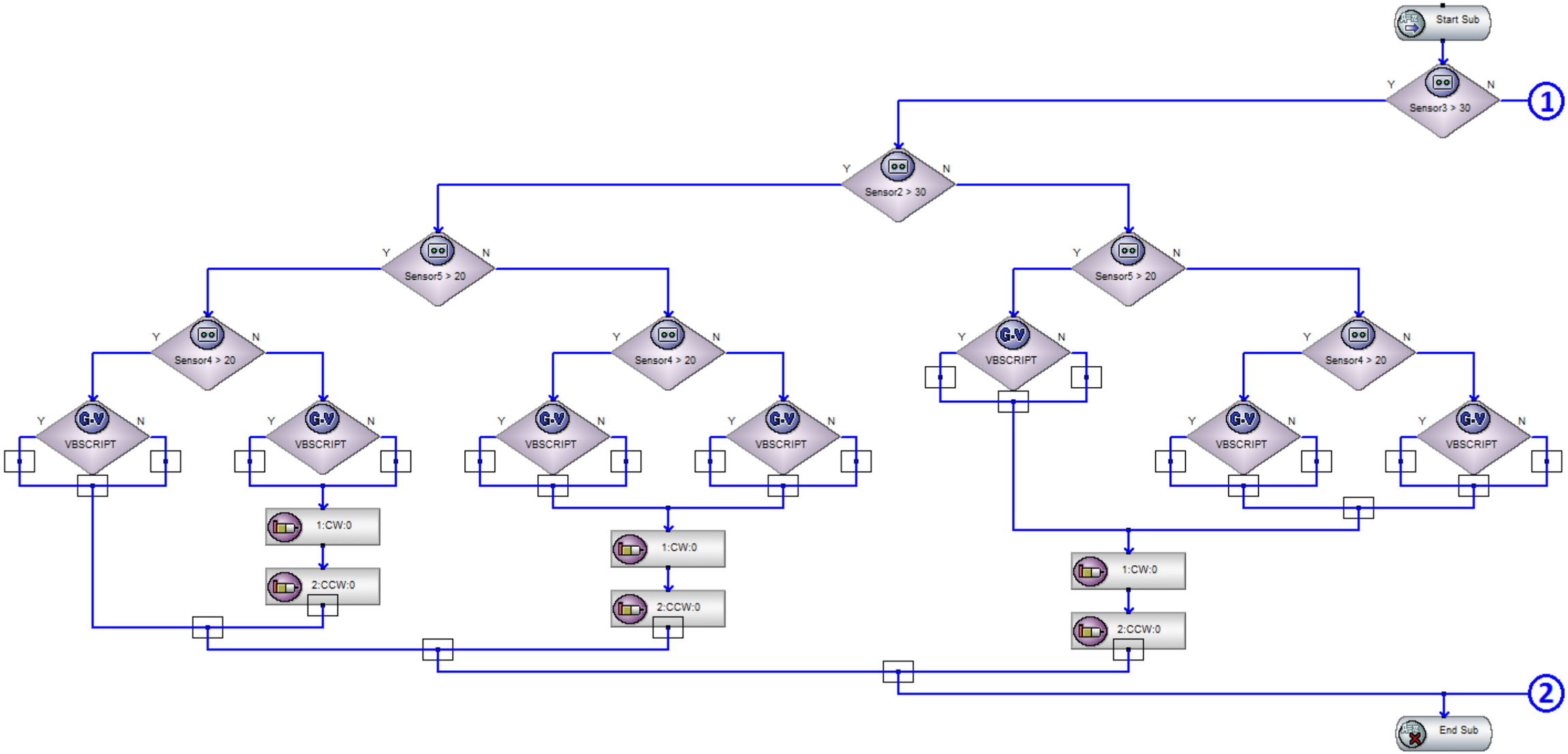


Figure. III.12.A.1. *Sous-programme « Gauche » partie 1.*

Annexe A.2

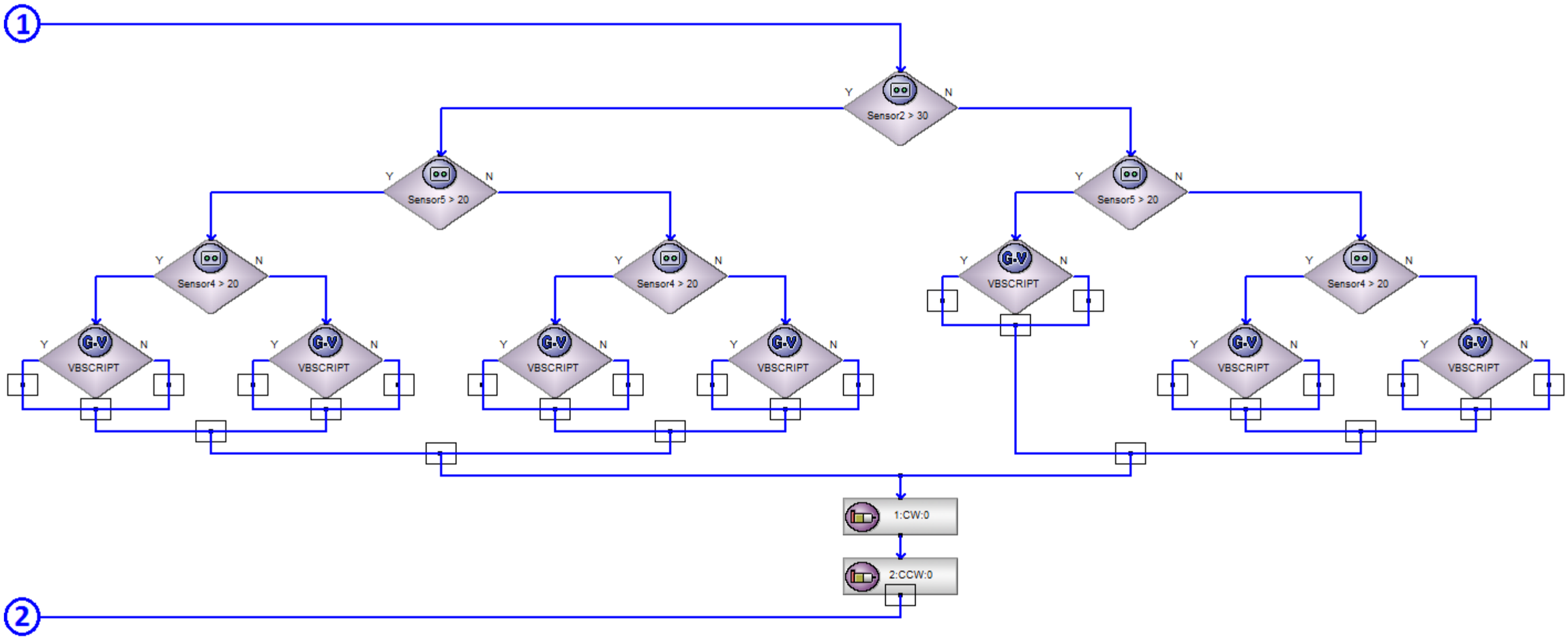


Figure. III.12.A.2. *Sous-programme « Gauche » partie 2.*

Annexe A.3

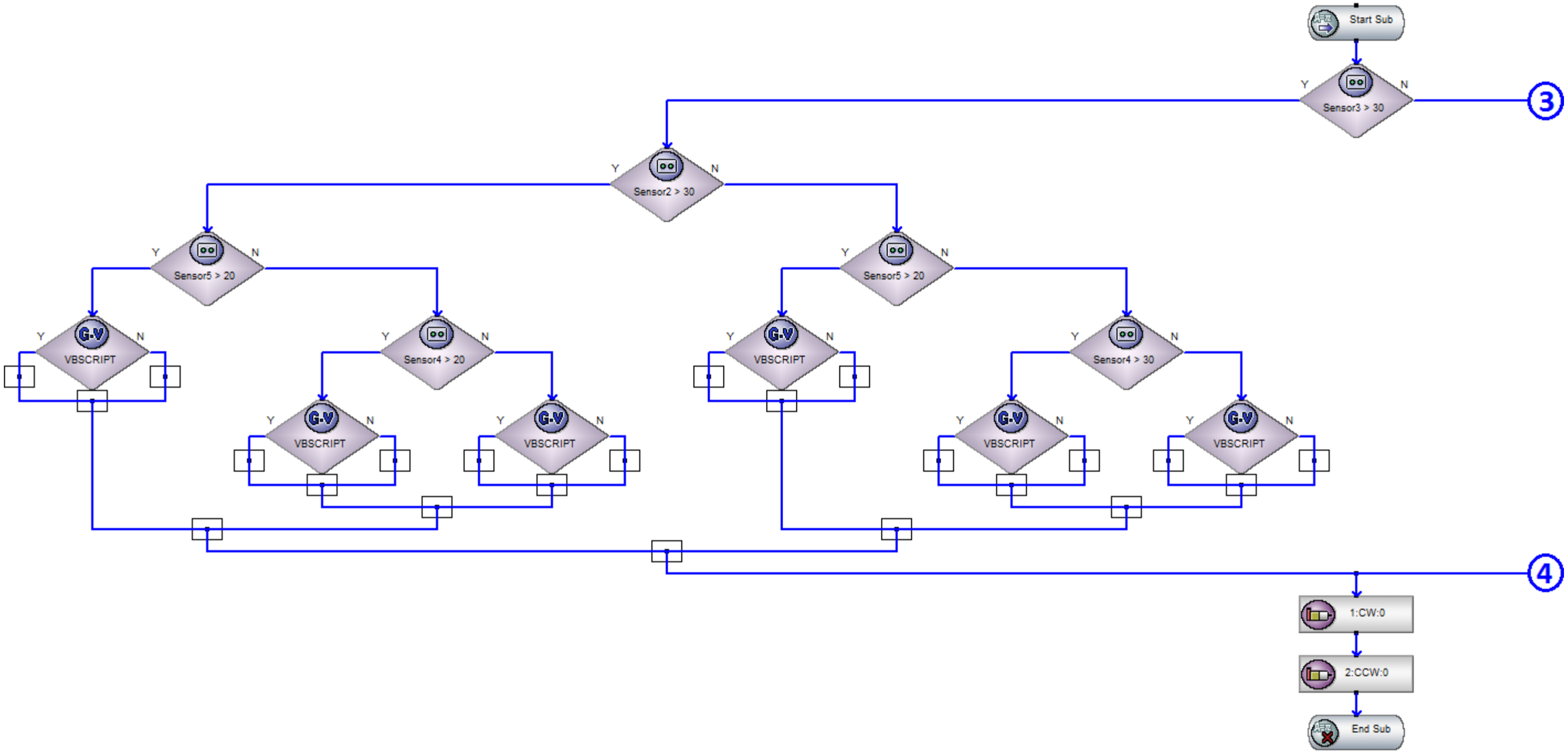


Figure. III.13.A.3. *Sous-programme « Droite » partie 1.*

Annexe A.4

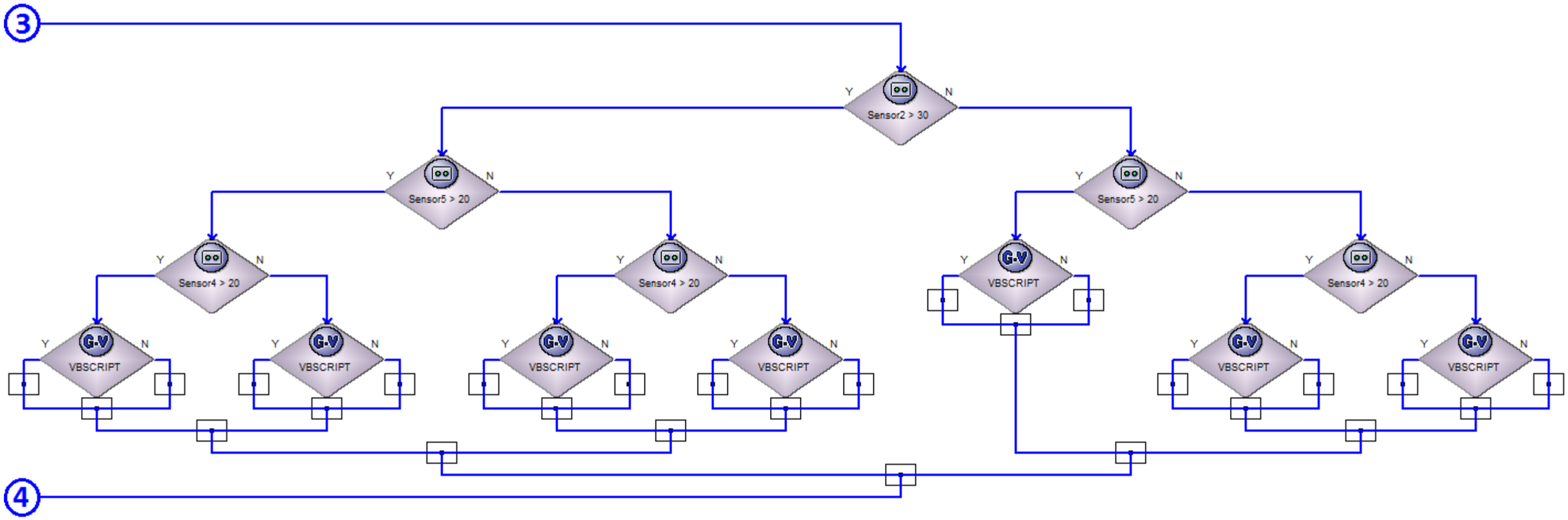


Figure. III.13.A.4. *Sous-programme « Droite » partie 2.*

Annexe B.1 : Programme contrôle et commande de DC-Motor

```
void CIntelligent_RobotKitDlg::OnMotorInit()
{
	// Initialisation Moteurs
	pMotion = NULL;  // initialisation de l'objet
	if( DCServo01_Initialize( m_hWnd, &pMotion ) != EDROBOT_API_OK ) //
initialiser moteur
		::MessageBox( NULL, "DCServo01_Init Error", "Function Error", MB_OK );
}

void CIntelligent_RobotKitDlg::OnFillterSet()
{
	// Réglage du filtre PID
	UpdateData(TRUE);// charger les valeurs dans les boites de dialogue vers les
variables
	DCServo01_FillterSet( pMotion, 1, 13, 51, 200, 0, 0, 0 );
	DCServo01_FillterSet( pMotion, 2, 13, 51, 200, 0, 0, 0 );
}

void CIntelligent_RobotKitDlg::OnBtAcceleration()
{
	// Affecter la valeur contenue dans la variable (m_Acceleration) a
l'accélération des deux moteurs
	UpdateData(TRUE);// charger les valeurs dans les boites de dialogue vers les
variables (m_Acceleration)

	DCServo01_SetAbsoluteAcceleration( pMotion, 1, m_Acceleration );// Affecter
l'accélération
	DCServo01_SetAbsoluteAcceleration( pMotion, 2, m_Acceleration );

	UpdateData(FALSE);//charger les variables dans la boite de dialogue (inutile
dans ce cas)
}

void CIntelligent_RobotKitDlg::OnBtVelocity()
{
	// définir la vitesse des deux moteurs (m_Velocity)
	UpdateData(TRUE);
	DCServo01_SetAbsoluteVelocity(pMotion, 1, m_Velocity);// Affecter la vitesse
	DCServo01_SetAbsoluteVelocity(pMotion, 2, m_Velocity);// Affecter la vitesse
	UpdateData(FALSE); // transfert des données vers les boite de dialogues
}

void CIntelligent_RobotKitDlg::OnBtUP()
{
	// Aller vers l'avant
	DCServo01_ForwardStart( pMotion, 1 ); // moteur 1 au sens CW
	DCServo01_BackwardStart( pMotion, 2 ); // moteur 2 au sens CCW
}
void CIntelligent_RobotKitDlg::OnBtStop()
{
	// Arrêter doucement les moteurs
	DCServo01_StopSmoothly( pMotion, 1 ); // Arrêter doucement moteur 1
	DCServo01_StopSmoothly( pMotion, 2 );
}

void CIntelligent_RobotKitDlg::OnBtRight()
{
	// Tourner à droite
	DCServo01_ForwardStart( pMotion, 1 ); // moteur 1 au sens CW
	DCServo01_ForwardStart( pMotion, 2 ); // moteur 2 au sens CW
```

```
}

void CIntelligent_RobotKitDlg::OnBtLeft()
{
	// Tourner à gauche
	DCServo01_BackwardStart( pMotion, 1 ); // moteur 1 au sens CCW
	DCServo01_BackwardStart( pMotion, 2 ); // moteur 2 au sens CCW
}
void CIntelligent_RobotKitDlg::OnBtBack()
{
	// Faire marche arriére
	DCServo01_BackwardStart( pMotion, 1 ); // moteur 1 au sens CCW
	DCServo01_ForwardStart( pMotion, 2 ); // moteur 2 au sens CW
}

void CIntelligent_RobotKitDlg::OnBtReadPosition()
{
	// Lire et afficherles positions des moteurs (impulsion)
	UpdateData(TRUE);// Initialization (facultative)

	if( DCServo01_ReadPosition( pMotion, 1, m_ReadPosition ) != EDROBOT_API_OK )
		::MessageBox( NULL, "DCServo01_ReadPosition Error", "Function Error",
MB_OK );// lire la	position sinon afficher message d'erreur dans une boite de
message.
	if( DCServo01_ReadPosition( pMotion, 2, m_ReadPosition2 ) != EDROBOT_API_OK
)
		::MessageBox( NULL, "DCServo01_ReadPosition Error", "Function Error",
MB_OK );

	UpdateData(FALSE);// Afficher les variables m_ReadPosition , m_ReadPosition
2
}

void CIntelligent_RobotKitDlg::OnBtReadDegree()
{
	 // Lire et afficher les position en degré
	UpdateData(TRUE);
	if( DCServo01_ReadDegree( pMotion, 1, m_ReadDegree )
		!= EDROBOT_API_OK ) ::MessageBox( NULL, "DCServo01_ReadDegree Error",
"Function Error", MB_OK );

	Sleep(50); // attendre pendant 50 Tics

	if( DCServo01_ReadDegree( pMotion, 2, m_ReadDegree2 )
		!= EDROBOT_API_OK ) ::MessageBox( NULL, "DCServo01_ReadDegree Error",
"Function Error", MB_OK );
	UpdateData(FALSE);// afficher la position des moteurs en degré
}

void CIntelligent_RobotKitDlg::OnBtCentimter()
{
	// Lire et afficher les positions en centimetre
	UpdateData(TRUE);
	if( DCServo01_ReadCentimeter( pMotion,1 , m_Read_Centimter ) !=
EDROBOT_API_OK )
		::MessageBox( NULL, "DCServo01_ReadCentimeter Error", "Function
Error", MB_OK );

	Sleep(50); // Attendre 50 Tics

	if( DCServo01_ReadCentimeter( pMotion, 2, m_Read_Centimter2 ) !=
EDROBOT_API_OK )
```

```
            ::MessageBox( NULL, "DCServo01_ReadCentimeter Error", "Function
Error", MB_OK );

      UpdateData(FALSE);

}

void CIntelligent_RobotKitDlg::OnBtReadRPM()
{
      // Lire et afficher les fréquences en tours/minute
      UpdateData(TRUE);

      if( DCServo01_ReadRpm( pMotion, 1, m_ReadRPM ) != EDROBOT_API_OK )
            ::MessageBox( NULL, "DCServo01_ReadRpm Error", "Function Error", MB_OK
);

      Sleep(50);

      if( DCServo01_ReadRpm( pMotion, 2, m_ReadRPM ) != EDROBOT_API_OK )
            ::MessageBox( NULL, "DCServo01_ReadRpm Error", "Function Error", MB_OK
);


      UpdateData(FALSE);
}
```

Annexe B.2 : Programme pour trajectoires en formes géométriques

```
void CRect_Tria_CercleDlg::OnBtRectangle()
{
// Moteur initialisation ====================;
pMotion = NULL;  // object init
if( DCServo01_Initialize( m_hWnd, &pMotion ) != EDROBOT_API_OK)//moteur init
       ::MessageBox( NULL, "Erreur d'Initialisation!!", "Function Error", MB_OK );
//==========================================;

UpdateData(TRUE); // m_Rect_Longueur; m_Rect_Largeur; m_Vitesse;
//    R = 7.5; (cm)
//    L = 12.0; (cm)
//    V = m_Vitesse; (tour/ min) une fréquence
//    V = 13.0544 cm/s

double V(13.30),t(0.);
// CLOCKS_PER_SEC = 1000;     nombre de TIC par seconde
// Comme on a converti le temps en nombre de TIC (clock) alors il suffit de
//                            travaillé en TIC

for (int i_boucler = 1; i_boucler <= 2; i_boucler++)
{

      long tf(0);
      t = CLOCKS_PER_SEC*m_Rect_Largeur/V;//(TIC)cm/cm donner t en unités de TIC
      tf = long(t);     //    une erreur entre 0 et -1 TIC.
      tf += clock();    // ajouter le temps que doit rester la roue en rotation.

      // aller vers l'avant. ======================;
      DCServo01_ForwardStart( pMotion, 1 ); // moteur1 de gauche au sens cw.
      DCServo01_BackwardStart( pMotion, 2 );// moteur2 de droite au sens ccw.
      // ==========================================;
      // commencer le comptage du temps==================;

      while(tf > clock())
            {
            // Tester les capteur pour éviter la collision!
            }

      // arrêter les 2 moteurs. ====================;
      DCServo01_StopSmoothly( pMotion, 1 );
      DCServo01_StopSmoothly( pMotion, 2 );
      //==========================================;

      // Tourner de 90° (pi/2 );    12*pi/2 = 6*pi.
      t = CLOCKS_PER_SEC*6.0*pi/V;        //    (TIC) cm/cm.
      tf = long(t);     // Affecter t (TIC) à tf « le temps que doit rester la roue
                        //                          en rotation ».
      tf += clock();    // Ajouter le temps actuel à tf

      // Tourner à gauche =========================;
      DCServo01_BackwardStart( pMotion, 2 );
      DCServo01_BackwardStart( pMotion, 1 );
      // ==========================================;

      while(tf > clock())
            {
            // Rien faire
            }
      // Arrêter les 2 moteurs. ====================;
      DCServo01_StopSmoothly( pMotion, 2 );
```

```
	DCServo01_StopSmoothly( pMotion, 1 );
	//=========================================;

	t = CLOCKS_PER_SEC*m_Rect_Longueur/V;  //(TIC) cm/cm
	tf = long(t);	//	une erreur entre 0 et 1 TIC.
	tf += clock();	// ajouter le temps que doit rester la roue en rotation.

	// Aller vers l'avant. ======================;
	DCServo01_BackwardStart( pMotion, 2 );
	DCServo01_ForwardStart( pMotion, 1 );
	// =========================================;

	// Commencer le comptage du temps=================;
	while(tf > clock())
		{
		// Tester les capteurs pour éviter la collision!
		}
	// arreter les 2 moteurs. ====================;
	DCServo01_StopSmoothly( pMotion, 2 ); // DCServo01_StopAbruptly(pMotion, 2);
	DCServo01_StopSmoothly( pMotion, 1 ); // DCServo01_StopAbruptly(pMotion, 1);
	//=========================================;

	// Tourner de 90° (pi/2);	12*pi/2 = 6*pi.
	t = CLOCKS_PER_SEC*6.0*pi/V;  //	(TIC) cm/cm.
	tf = long(t);// Calculer t (TIC)le temps que doit tourner la roue
	tf += clock();	// Ajouter le temps actuel à tf

	// Tourner à gauche ==========================;
	DCServo01_BackwardStart( pMotion, 1 );
	DCServo01_BackwardStart( pMotion, 2 );
	// =========================================;

	while(tf > clock())
		{
		//	Ne rien faire
		}
	// arreter les 2 moteurs. ====================;
	DCServo01_StopSmoothly( pMotion, 1 );
	DCServo01_StopSmoothly( pMotion, 2 );
	//=========================================;
	}
}

void CRect_Tria_CercleDlg::OnBtTriangle()
{
// Moteur initialisation ====================;
pMotion = NULL;  // object init
if( DCServo01_Initialize( m_hWnd, &pMotion ) != EDROBOT_API_OK)//moteur init
	::MessageBox( NULL, "Erreur d'Initialisation!!", "Function Error", MB_OK );
//=========================================;

UpdateData(TRUE); // m_Tria_Arete;
//	R = 7.5; (cm)
//	L = 12.0; (cm)
//	V = 13.0544 cm/s « vitesse mesurée ».

double V(13.25),t(0.);

for (int i_boucler = 1; i_boucler <= 3; i_boucler++)
{
	long tf(0);
	t = CLOCKS_PER_SEC*m_Tria_Arete/V;//(TIC) cm/cm donner t en unités de TIC
	tf = long(t);	//	une erreur entre 0 et -1 TIC.
```

```
	tf += clock();    // ajouter le temps que doit rester la roue en rotation.

	// Aller vers l'avant. ======================;
	DCServo01_ForwardStart( pMotion, 1 );
	DCServo01_BackwardStart( pMotion, 2 );
	// ==========================================;
	// Commencer le comptage du temps=================;
	while(tf > clock())
		{
		// Tester les capteurs pour éviter la collision!
		}
	// Arrêter les 2 moteurs. ====================;
	DCServo01_StopSmoothly( pMotion, 1 );// DCServo01_StopAbruptly(pMotion, 1);
	DCServo01_StopSmoothly( pMotion, 2 );// DCServo01_StopAbruptly(pMotion, 2);
	//==========================================;

	// Tourner de 120° (pi/3);    12*2*pi/3 = 8*pi.
	t = CLOCKS_PER_SEC*8.0*pi/V;  //    (TIC).
	tf = long(t);// t (TIC) le temps que doit tourner la roue.
	tf += clock();// Ajouter le temps actuel à tf.

	// Tourner à gauche ==========================;
	DCServo01_BackwardStart ( pMotion, 2 );
	DCServo01_BackwardStart ( pMotion, 1 );
	// ==========================================;
	while(tf > clock())
		{
		// Rien faire le robot entrain de tourner autour de lui-même.
		}
		// Arrêter les 2 moteurs. ========================;
		DCServo01_StopSmoothly( pMotion, 2 );
		DCServo01_StopSmoothly( pMotion, 1 );
		// ==========================================;
	}
}
void CRect_Tria_CercleDlg::OnBtCercle()
{
// Moteur initialisation ====================;
pMotion = NULL;  // objet init
if( DCServo01_Initialize( m_hWnd, &pMotion ) != EDROBOT_API_OK)//moteur init
       ::MessageBox( NULL, "Erreur d'Initialisation!!", "Function Error", MB_OK );
//==========================================;
UpdateData(TRUE); // m_Gauche; m_Droite;
//    R = 7.5; (cm)
//    L = 12.0; (cm)
//    V = 13.0544 cm/s; « vitesse mesurée »

double V(13.30),t(0.);

long tf(0);
// Tourner de 360° (2*pi);    24*2*pi = 48*pi.
t = CLOCKS_PER_SEC*48.0*pi/V; //    (TIC) cm/cm.
tf = long(t);     //    une erreur entre 0 et -1 TIC.
if (m_Gauche)
	{
	tf += clock(); // ajouter le temps que doit rester la roue en rotation.
	// Tourner à gauche au tour de la roue Gauche ===============;
	DCServo01_BackwardStart( pMotion, 2 ); // moteur2: ccw
	// ==========================================;
	while(tf > clock())
		{
		//    Rien faire le robot entrain de tourner autour de lui-même.
		}
```

```
            // arreter le moteurs 2. ====================;
            DCServo01_StopAbruptly( pMotion, 2 );
            // ==========================================;
      }
if (m_Droite)
      {
      tf = long(t) + clock();// ajouter le temps que doit tourner la roue.
      // Tourner à gauche au tour de la roue Gauche ===============;
      DCServo01_ForwardStart( pMotion, 1 ); // moteur1: CW
      // ==========================================;
      while(tf > clock())
            {
            //    Rien faire le robot entrain de tourner autour de lui-même.
            }
            // arreter le moteurs 1. ====================;
            DCServo01_StopAbruptly( pMotion, 1 );
            // ==========================================;
      }
}
```

IV.1. Annexe B.3 : Programme pour trajectoires polygonales

```
void CPolygone_PolyCercleDlg::OnBtPolygone()
{
// Moteur initialisation ====================;
pMotion = NULL;  // object init
if( DCServo01_Initialize( m_hWnd, &pMotion ) != EDROBOT_API_OK )// moteur init
       ::MessageBox( NULL, "Erreur d'Initialisation!!", "Function Error", MB_OK );
//==========================================;

UpdateData(TRUE); // charger m_Gauche, m_Droite, m_Longueur et m_Poly_nb.
//    R = 7.5; (cm)
//    L = 12.0; (cm)
//    V = m_Vitesse; (tour/ min) une fréquence
//    V = 13.0544 cm/s

double V(13.05),t(0.);
long tf(0);
if (m_Gauche)// si « à gauche » est cochée
      {
      for (int i_boucler = 1; i_boucler <= m_Poly_nb; i_boucler++)
      {
            tf = 0;
            t = CLOCKS_PER_SEC*m_Longueur/V;//(TIC) donner t en unités de TIC
            tf = long(t);     //une erreur entre 0 et -1 TIC.
            tf += clock();// ajouter le temps que doit rester la roue en rotation.

            // aller vers l'avant. ======================;
            DCServo01_ForwardStart( pMotion, 1 ); // cw: moteur1 "gauche"
            DCServo01_BackwardStart( pMotion, 2 ); // ccw: moteur1 "droite"
            // ==========================================;
            // commencer le comptage du temps==================;
            while(tf > clock())
                  {
                  // Tester les capteurs pour éviter la collision!
                  }

            // arreter les 2 moteurs. ====================;
            DCServo01_StopAbruptly( pMotion, 1 );
            DCServo01_StopAbruptly( pMotion, 2 );
            //==========================================;

            // Tourner de 360/m_Poly_nb° ( 2*pi/m_Poly_nb );
            t = CLOCKS_PER_SEC*24.0*pi/m_Poly_nb/V;   //    (TIC) cm/cm.
            tf = long(t);// t (TIC) « le temps que doit tourner la roue».
            tf += clock();    // Ajouter le temps actuel à tf

            // Tourner à guache =========================;
            DCServo01_BackwardStart( pMotion, 2 );
            DCServo01_BackwardStart( pMotion, 1 );
            // ==========================================;

            while(tf > clock())
                  {
                  //    Rien faire le robot entrain de tourner autour de lui-même.
                  }
            // arreter les 2 moteurs. ====================;
            DCServo01_StopAbruptly( pMotion, 2 );
            DCServo01_StopAbruptly( pMotion, 1 );
            // ==========================================;
      }
```

```
	}

		if (m_Droite) // si « à droite » est cochée
	{
	for (int i_boucler = 1; i_boucler <= m_Poly_nb; i_boucler++)
	{
		tf = 0;
		t = CLOCKS_PER_SEC*m_Longueur/V;//(TIC) calculer t en unités de TIC
		tf = long(t);	// une erreur entre 0 et -1 TIC.
		tf += clock();	// ajouter le temps que doit tourner la roue.
		// Aller vers l'avant. ======================;
		DCServo01_ForwardStart( pMotion, 1 );
		DCServo01_BackwardStart( pMotion, 2 );
		// ==========================================;
		// Commencer le comptage du temps=================;
		while(tf > clock())
			{
			// Tester les capteurs pour éviter la collision!
			}

		// arreter les 2 moteurs. ====================;
		DCServo01_StopAbruptly( pMotion, 1 );
		DCServo01_StopAbruptly( pMotion, 2 );
		//==========================================;

		// Tourner de 360/m_Poly_nb° (2*pi/m_Poly_nb);
		t = CLOCKS_PER_SEC*24.0*pi/m_Poly_nb/V;  //    (TIC).
		tf = long(t); // Affecter t (TIC) "le temp que doit tourner la roue.
		tf += clock();// Ajouter le temps actuel à tf.

		// Tourner à droite ==========================;
		DCServo01_ForwardStart( pMotion, 2 );
		DCServo01_ForwardStart( pMotion, 1 );
		// ==========================================;

		while(tf > clock())
			{
			//    Rien faire le robot entrain de tourner autour de lui-même.
			}
		// arreter les 2 moteurs. ====================;
		DCServo01_StopAbruptly( pMotion, 2 );
		DCServo01_StopAbruptly( pMotion, 1 );
		// ==========================================;
	}
	}
}

void CPolygone_PolyCercleDlg::OnBtCercle()
{
// Initialisation Moteur ====================;
pMotion = NULL;  // object init
if( DCServo01_Initialize( m_hWnd, &pMotion ) != EDROBOT_API_OK )//motor init
	::MessageBox( NULL, "Erreur d'Initialisation!!", "Function Error", MB_OK );
//==========================================;

UpdateData(TRUE); // m_Rayon; m_Cercle_nb;
//    R = 7.5; (cm)
//    L = 12.0; (cm)
//    V = m_Vitesse; (tour/ min) une frequence
//    V = 13.0544 cm/s

double V(13.05),t(0.),d(0.);
long tf(0);
```

```
d = m_Rayon*2*sin(pi/double(m_Cercle_nb));
if (m_Gauche)
{
for (int i_boucler = 1; i_boucler <= m_Cercle_nb; i_boucler++)
      {
      tf = 0;
      t = CLOCKS_PER_SEC*d/V;///(TIC) donner t en unitées de TIC
      tf = long(t);     // une erreur entre 0 et -1 TIC.
      tf += clock();    // ajouter le temps que doit rester la roue en rotation.

      // Aller vers l'avant. ======================;
      DCServo01_ForwardStart( pMotion, 1 );
      DCServo01_BackwardStart( pMotion, 2 );
      // ==========================================;
      // Commencer le comptage du temps============;
      while(tf > clock())
            {
            // Tester les capteurs pour éviter la collision!
            }
      // ==========================================;
      // Arrêter les 2 moteurs. ====================;
      DCServo01_StopAbruptly( pMotion, 1 );
      DCServo01_StopAbruptly( pMotion, 2 );
      //==========================================;

      // Tourner de 360/m_Poly_nb° (2*pi/m_Poly_nb);
      t = CLOCKS_PER_SEC*24.0*pi/m_Cercle_nb/V;       //    (TIC) cm/cm.
      tf = long(t);     // t (TIC) le temps que doit tourner la roue.
      tf += clock();    // Ajouter le temps actuel à tf

            // Tourner à gauche ========================;
            DCServo01_BackwardStart( pMotion, 2 );
            DCServo01_BackwardStart( pMotion, 1 );
            // ==========================================;
            // Commencer le comptage du temps============;
            while(tf > clock())
                  {
                  //    Rien faire le robot entrain de tourner autour de lui-même.
                  }
            // Arrêter les 2 moteurs. ====================;
            DCServo01_StopAbruptly( pMotion, 2 );
            DCServo01_StopAbruptly( pMotion, 1 );
            // ==========================================;
      }
      }
      if (m_Droite)
      {
      for (int i_boucler = 1; i_boucler <= m_Cercle_nb; i_boucler++)
      {
            tf = 0;
            t = CLOCKS_PER_SEC*d/V;///(TIC) calculer t en unités de TIC
            tf = long(t);     //    une erreur entre 0 et -1 TIC.
            tf += clock();    // ajouter le temps que doit tourner la roue.

            // Aller vers l'avant. ======================;
            DCServo01_ForwardStart( pMotion, 1 );
            DCServo01_BackwardStart( pMotion, 2 );
            // ==========================================;
            // Commencer le comptage du temps============;
            while(tf > clock())
                  {
                  // Tester les capteurs pour éviter la collision!
                  }
```

```
            // Arrêter les 2 moteurs. ====================;
            DCServo01_StopAbruptly( pMotion, 1 );
            DCServo01_StopAbruptly( pMotion, 2 );
            //===========================================;

            // Tourner de 360/m_Poly_nb° (2*pi/m_Poly_nb);
            t = CLOCKS_PER_SEC*24.0*pi/m_Cercle_nb/V;//    (TIC) cm/cm.
            tf = long(t);// calculer t (TIC) le temps que doit tourner la roue.
            tf += clock();// Ajouter le temps actuel à tf

            // Tourner à droite ==========================;
            DCServo01_ForwardStart( pMotion, 2 );
            DCServo01_ForwardStart( pMotion, 1 );
            // ===========================================;

            while(tf > clock())
                  {
                  //    Rien faire le robot entrain de tourner autour de lui-même.
                  }
            // Arrêter les 2 moteurs. ====================;
            DCServo01_StopAbruptly( pMotion, 2 );
            DCServo01_StopAbruptly( pMotion, 1 );
            // ===========================================;
      }
      }
}
```

Résumé :

Ce travail concerne le développement d'applications pour le robot ED-7273. Après l'exploration du domaine de la robotique mobile, on a procédé à l'étude des deux robots ED-7271 et ED-7273 (leurs descriptions et les modules qui les composent) et l'outil de leur programmation, à savoir le logiciel IRES (Intelligent Robot Education Studio).

Dans la partie développement d'applications. On a testé d'abord, sous IRES, les capteurs à infrarouge et à ultrason, et élaboré un programme d'évitement d'obstacles et un autre de suivi d'objet en mouvement, en utilisant les logigrammes et les scripts. Enfin, sous Visual C++, on a donné quelques notions de la programmation orientée objets et sur l'intégration des bibliothèques. On a fait un bilan sur les fonctions pour les moteurs et les capteurs. On a terminé avec quelques programmes de suivi de trajectoires.

Abstract :

This work deals with application development for the ED-7273 robot. After a brief survey of the mobile robots field, we have introduced the two robots ED-7271 and ED-7273 (their descriptions and their modules) and their programing tool, *i.e.*, the IRES software (Intelligent Robot Education Studio).

In the application development part, first we have tested, under IRES, infrared and ultrasonic sensors, and developed a program for obstacles avoidance and other moving object tracking, using logic diagram (flow charts) and scripts. Finally, under Visual C ++, we have introduced some basic notions on object-oriented programming and the libraries integration. We made an assessment on the functions for motors and sensors. Finally, we developed some path tracking programs.