UNIVERSITÉ DE LORRAINE | UFR MATHÉMATIQUES INFORMATIQUE MÉCANIQUE ET AUTOMATIQUE

Loria
Laboratoire lorrain de recherche en informatique et ses applications

PROJET TUTORÉ

Présenté et soutenu le *27/05/2014* **par :**

Quentin HOUBRE & Henri REYDON

COUPLAGE D'UN MODÈLE DE COMPORTEMENT AVEC UN MODÈLE DE PERCEPTION

FORMATION MASTER 1

Sciences de la Cognition et Applications

Année : 2013 - 2014

Établissement de formation :
UFR Mathématiques et Informatique de l'université de Lorraine

Laboratoire :
Laboratoire lorrain de recherche en informatique et ses applications (LORIA)

Équipe d'encadrement :
Christine BOURGEOT, *Vincent CHEVRIER*,
Benjamin CAMUS et *Julien VABOURG*

Rapporteur :
Armelle Brun

Remerciements

Dans un premier temps, nous voudrions exprimer nos remerciements à Christine Bourgeot et Vincent Chevrier pour l'accompagnement particulièrement suivi qu'ils nous ont offert.

Dans un second temps, nous remercions Benjamin Camus et Julien Vaubourg pour l'accueil, les conseils, et leurs grande disponibilité.

Par ailleurs, nous exprimons notre gratitude à Nassim Kaldé qui a su nous faire profiter de son expérience importante des programmes que nous avons utilisés.

Plus généralement, nous remercions toute l'équipe d'encadrement pour sa sympathie et sa patience qui a rendu la réalisation de ce travail très agréable.

Table des matières

Introduction

Lorsqu'un système est représenté numériquement, sa dynamique est simulée en conséquence de quoi, la plupart des règles de fonctionnement lui sont propres et représentent de manière imparfaite celles du monde réel. Réciproquement, l'environnement d'un système voué à une utilisation dans le monde réel est fondé sur les lois de la physique et on ne peut alors s'abstraire des notions de coûts, d'éthique... Le cadre de notre projet se situe en quelque sorte à mi-chemin entre ces deux approches. L'objectif visé est de réaliser un travail progressif et exploratoire permettant d'étudier les possibles interactions entre un monde réel et un monde virtuel.

Plus simplement, nous pourrions dire que le but est de simuler les perceptions d'un agent qui est présent dans le monde réel. En terme de représentation du problème, il nous a été exposé le phénomène du fourragement chez les fourmis. Il est très difficile de représenter ce système dans le monde réel avec des robots. Déjà parce qu'il n'est pas évident pour un robot de déposer des phéromones, mais aussi parce qu'il est encore moins évident de pouvoir les détecter. C'est pour répondre à ce problème qu'a émergé l'idée de séparer les différents processus de perception, décision et action. En effet, il est relativement simple de simuler des phéromones dans un environnement virtuel puisque les possibilités de modélisation sont nombreuses. C'est ce que fait déjà le simulateur de phénomènes collectifs Netlogo. Seulement, nous voudrions qu'il n'y ait que la perception qui soit virtuelle alors que l'action, elle, doit être effectuée par un robot dans le monde réel. L'exemple du fourragement étant un peu trop complexe, il nous a été demandé de travailler sur les machines de Braitenberg.

L'équipe chargée de nous suivre sur ce projet dispose déjà de *Agent & Artefacts for Multi-Modeling* (A4MM), un concept multi-agent permettant l'interconnexion de modèles. Le sujet que nous traitons devant permettre le couplage d'un modèle de comportement avec un modèle de perception, nous orienterons notre démarche en vue de l'utilisation de cet outil qui peut nous être grandement utile.

Afin de répondre à ces objectifs, nous allons dans un premier temps exposer quelques uns des fondements théoriques que nous avons étudiés pour l'occasion. Ensuite, nous nous attacherons à exposer les choix que nous avons fait en terme d'implémentation, les raisons qui ont motivé

ces choix ainsi que la démarche que nous avons adoptée pour tenter d'atteindre les objectifs qui nous ont étés fixés.

1 Le projet

1.1 Objectifs

Les machines de Braitenberg permettent de faire émerger un comportement complexe à partir de règles simples. Ces agents disposent à cet effet de capteurs permettant de percevoir des informations simples (luminosité) et de les traiter afin de produire des mouvements qui le sont tout autant. Cette machine est en fait un "agent intelligent", notion qui sera abordée par la suite. Elle évolue dans un "environnement" et fonctionne sur ce que l'on pourrait appeler une boucle de comportement : les capteurs de lumière perçoivent une intensité et s'en servent pour activer plus ou moins ses moteurs. La machine se déplace, ce qui modifie ses perceptions de l'environnement. Ces opérations se répètent en continu en suivant ce schéma :

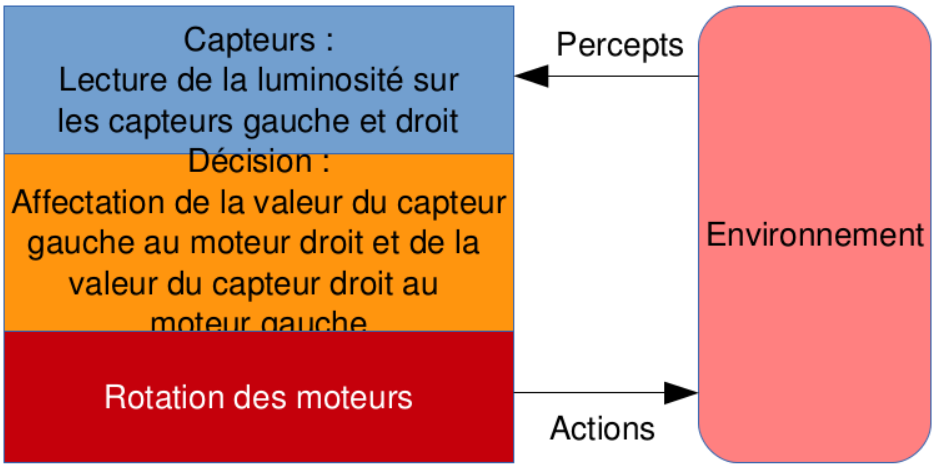

Figure 1. Boucle Perception/Action pour une machine de Braitenberg

L'objectif de ce projet est de simuler la perception de l'une de ces machines dans un environnement virtuel et de produire les mouvements correspondants dans le monde réel.

Si nous revenons sur l'idée du fouragement chez les fourmis, il n'est pas difficile de justifier cette approche. Dans le principe, une fourmi dépose des phéromones qui, en fonction de leur concentration attirent plus ou moins de congénères. Notre problème se situant sur la détection de celles-ci, l'analogie entre ce comportement et les concepts de Braitenberg se justifie par le parallèle entre intensité lumineuse et concentration de phéromones.

Pour cela, il faudra trouver une solution conceptuelle adaptée, et la développer avec la plateforme de couplage AA4MM. Dans un premier temps, la machine de Braitenberg et son environnement seront simulés dans des logiciels séparés. Puis un des logiciels de simulation sera remplacé par un robot (khepera).

1.2 Le concept "Agent Intelligent"

Nous disions juste avant que les machines de Braitenberg sont, de par leur comportement, des agents intelligents. Il convient alors de revenir en détail sur le fonctionnement d'un agent et les différents moyens de conceptualiser celui ci. Lors de nos cours, nous avons vu qu'un agent était une entité réelle ou artificielle dotée de capteurs et d'effecteurs, capables d'agir de manière autonome grâce à une fonction de décision. Ferber [Fer95] le définit comme suit :

> Un agent est une entité informatique autonome, située dans un environnement qu'elle peut percevoir et sur lequel elle peut agir.

Cela signifie que l'agent reçoit des perceptions à travers ses capteurs, qu'il les traite en interne à l'aide de fonctions de décisions, et qu'il renvoie une certaine action à faire à ses effecteurs. On appelle ce processus la boucle de perception action (Figure 2).

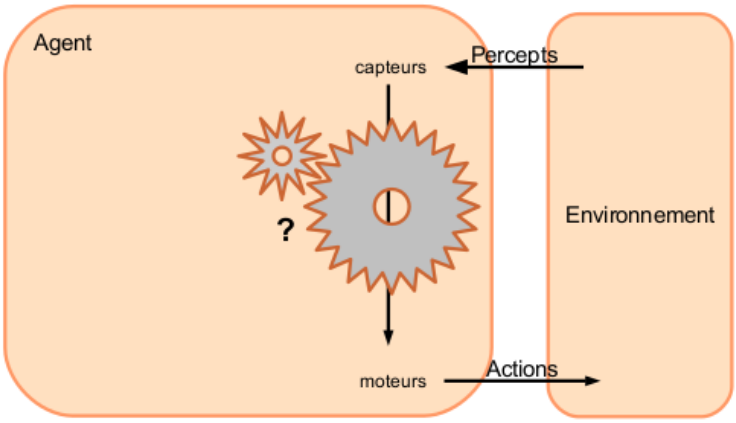

Figure 2. Boucle Perception/Action

De plus, Ferber définit ce que l'on appelle entité physique ou virtuelle à travers neuf points [Tho05] :

- Un agent est capable d'agir dans un environnement.
- Il peut communiquer directement avec d'autres agents.
- Il est mu par un ensemble de tendances (sous forme d'objectifs individuels ou d'une fonction de satisfaction).
- Il possède ses ressources propres.
- L'agent est capable de percevoir de manière limitée son environnement.
- Il ne dispose que d'une représentation partielle de son environnement (voir aucune).
- Un agent possède des compétences et offre des services.
- Il peut éventuellement se reproduire.
- Enfin, un agent possède un comportement qui tend à satisfaire ses objectifs, en tenant compte des ressources et des compétences dont il dispose, et en fonction de sa perception, de ses représentations et des communications qu'il reçoit.

Ces définitions posent un premier cadre de travail quant à la manière de penser en terme d'agents intelligents. Nous allons maintenant voir qu'il existe plusieurs façons de conceptualiser cette idée pour créer un agent.

1.2.1 L'agent rationnel

D'après les travaux de Vincent Thomas [Tho05], par Vincent Thomas sur la rationalité des agents, il est défini qu'un agent rationnel est un agent « qui effectue les bonnes tâches au bon moment. Il complète ensuite cette définition d'après les propos de Russel et Norvig qui préconisent d'utiliser une mesure de performance pour caractériser l'objectif de l'agent et évaluer les actions qu'il a pu choisir. Par exemple, si nous prenons le cas d'un aspirateur « intelligent », celui ci doit pouvoir évaluer des actions comme *ramasser le plus de poussière possible* ou encore *économiser son énergie*. Et c'est grâce à des fonctions d'évaluation que l'agent pourra prendre des décisions optimales concernant la gestion de l'énergie ou le calcul de trajet.

1.2.2 L'agent cognitif

Un agent cognitif est un agent « qui sait penser » dans le sens où il possède un modèle du monde et qu'il effectue des raisonnements sur celui-ci.

Pour mieux comprendre ce concept, nous allons nous intéresser au jeu du Wumpus (Hunt the Wumpus) développé par Gregory Yob. Le cadre du jeu est une grille dans laquelle évolue un héros (notre agent) ainsi qu'un monstre. On y trouve aussi des puits, de l'or et des indications telles que la présence de brise ou d'odeur (voir Figure 3). Le but est de trouver l'or sans tomber dans les puits, ni se faire tuer par le Wumpus que l'agent peut abattre en tirant une flèche dans sa direction.

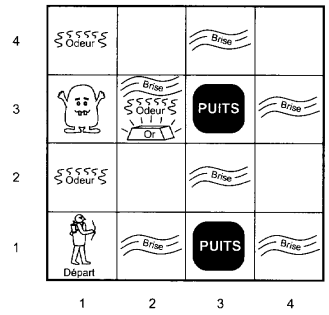

Figure 3. Jeu du Wumpus

Le Héros peut savoir ce que contient les cases autour de lui d'après le voisinage de Von Neumann [1]. S'il sent une brise, cela signifie qu'un puits est présent dans son propre voisinage. S'il sent une odeur, cela signifie que le Wumpus est présent dans celui de l'odeur en question.

Normalement, le joueur contrôle le héros et c'est lui qui fait ses propres déductions quant aux emplacements des puits, de l'or et du Wumpus. Mais ici nous voulons développer un agent cognitif pour prendre la place du joueur humain. Une des solutions serait de donner la carte du jeu à l'agent, sans lui dire où sont les puits ni l'or, mais en lui spécifiant le nombre de ceux-ci. On pourrait ainsi effectuer des déductions automatiquement en suivant les déplacements de l'agent. Le héros possède donc bien une représentation interne de la grille et raisonne sur celle ci à l'aide de déductions. Bien sûr, ce n'est pas la seule manière de faire raisonner l'agent mais cela permet tout de même de conceptualiser la manière dont un agent cognitif peut fonctionner.

1.2.3 L'agent réactif

Il existe un autre modèle d'agents, qualifié de *réactif*. Pourquoi ce terme ? Car il réagit par des réflexes en suivant les stimulus qu'il peut rencontrer. La réaction aux stimulus dépend d'un ensemble de règles programmées dans l'agent. Suite à la sélection d'une règle, les autres sont inhibées et l'action à faire est transmise aux effecteurs de l'agent. Pour notre exemple, nous utiliserons une des nombreuses variantes découlant de concept de Braitenberg. Notre machine doit percevoir des intensités lumineuses pour ensuite déclencher la rotation du moteur droit ou gauche entraînant les roues de façon indépendante. La lumière fait office de stimulus sur les capteurs. Le réflexe est représenté par une fonction de décision qui spécifie que l'intensité du capteur droit est transmise au moteur gauche et inversement. Une autre possibilité pourrait être de relier le capteur de luminosité gauche au moteur gauche et le capteur droit au moteur droit.

Il est également possible pour un agent réactif de posséder un état interne pour lui permettre de prendre d'autres décisions. Par exemple, il peut retenir un historique de ses perceptions et suivant son passé, changer ses réflexes. Comme nous le verrons plus tard, il est possible d'implémenter de nombreux comportements dans un agent réactif.

1. http ://mathworld.wolfram.com/vonNeumannNeighborhood.html

1.3 La machine de Braitenberg

Valentino Braitenberg [Bra86] a mis au point des machines dotées de capteurs et d'effecteurs capables de réflexes simples. Les machines sont en fait des petits véhicules similaires à des voitures mais possédant 2 roues reliées chacune à un moteur. Les véhicules sont également équipés de deux capteurs de luminosité à l'avant (voir Figure 4).

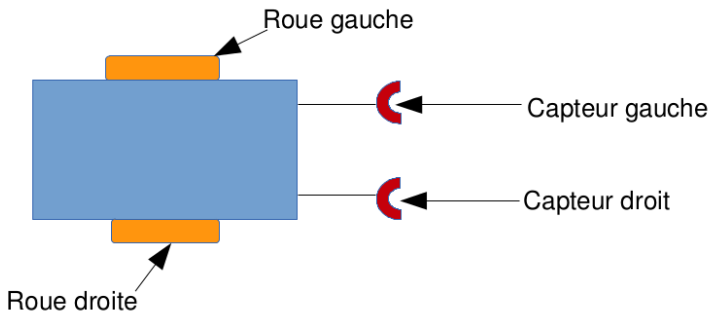

Figure 4. Machine de Braitenberg

Braitenberg a compris qu'avec cette structure, il est possible d'implémenter différents comportements. Par exemple, en reliant le capteur de luminosité gauche avec le moteur droit et en reliant le capteur droit au moteur gauche, le véhicule simulait un comportement d'agressivité en présence d'une lumière. Autrement dit, lorsqu'il capte une intensité lumineuse il va directement la retransmettre aux moteurs. Donc plus la luminosité perçue est élevée, plus la vitesse du moteur sera soutenue. Dans le cas de notre câblage (capteur droit sur moteur gauche, capteur gauche sur moteur droit), le véhicule va alors « foncer » sur la lumière comme lors d'un comportement agressif.

En changeant la manière de relier les moteurs et les capteurs entre eux, il est ainsi possible d'obtenir des comportements différents. Il est également envisageable de faire en sorte que la lumière perçue agisse plutôt de manière inhibitrice qu'excitatrice, où même de faire tourner les roues dans l'autre sens. Toute modification du câblage entraîne donc une modification du comportement de la machine, et il y a au moins huit comportements complexes identifiés (voir Figure 5).

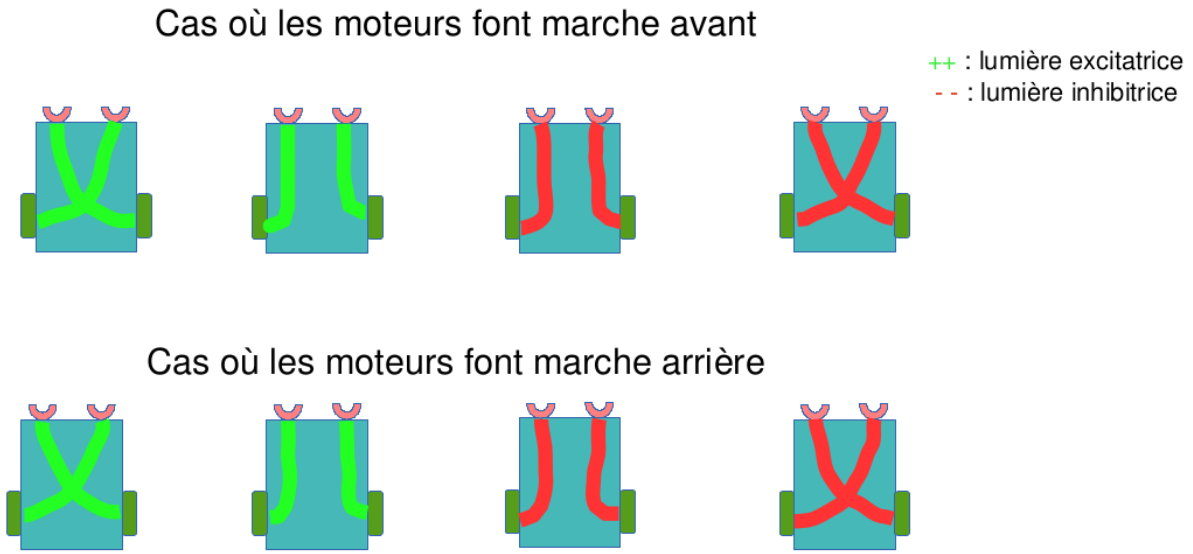

Figure 5. Boucle Perception/Action pour une machine de Braitenberg

Grâce à ses travaux, Braitenberg a montré qu'avec des machines simples et sans représentation du monde, il est possible de simuler des comportements complexes. Une machine de Braitenberg n'est en fait ni plus ni moins qu'un agent réactif.

1.4 Les systèmes "Multi-Agents" (SMA)

D'après les travaux de Vincent Thomas [Tho05], un Système Multi-Agents est une forme d'intelligence artificielle distribuée composée d'un ensemble d'agents intelligents qui interagissent au sein d'un même environnement. Lesser défini les SMA comme :

> Un réseau faiblement couplé de solveurs de problèmes qui travaillent ensemble pour résoudre des problèmes qui sont au-delà des capacités individuelles ou des connaissances de chaque solveur de problèmes.

Il est attendu une certaine autonomie de chacun des agents du système mais ceux-ci interagissent tout de même les uns avec les autres en fonction d'observations qu'ils peuvent faire soit à un niveau local (au niveau de l'agent lui même), soit à un niveau global (comportement du système ou du collectif).

Le projet que nous avons réalisé ne correspond pas exactement à un système Multi-Agent puisque nous ne disposons en réalité que d'un seul agent. Cependant, celui-ci est divisé en deux entités, que nous pourrions appeler semi-agents, dont le principe se base sur un agent et est directement transposable à plusieurs agents.

Dans cette mesure, les semi-agents n'ont pas de comportement à proprement parler puisque c'est la conjugaison des tâches complémentaires de chacune des entités qui le définissent. Dans le résultat final que nous souhaitons obtenir, l'une d'entre elle sera une machine de Braitenberg évoluant dans un environnement simulé. Le programme de simulation enverra les perceptions à la seconde entité lui permettant ainsi de prendre une décision et d'agir dans un environnement différent, potentiellement réel. Une fois l'action exécutée, l'entité ayant effectué le déplacement renvoie sa nouvelle position et son orientation à son « partenaire » et l'opération recommence en boucle.

Cette partie du dossier s'attachera donc à exposer quelques aspects des SMA censés nous permettre de mettre en place l'interconnexion des deux entités.

Concernant la notion d'autonomie, deux définitions sont possibles. Notre système se rapproche de celle proposée par Jennings [Jen98] consistant à dire que chaque agent décide de manière autonome de son action sans intervention extérieure. Cette définition implique que toute action émise dans le système est uniquement à l'initiative d'un agent. Elle est plus ou moins en opposition avec celle proposée par Russel, qui se base sur les expériences passées pour améliorer les capacités d'adaptation de l'agent.

Dans un Système Multi-Agent, les interactions peuvent prendre des formes très diverses qu'il est nécessaire de définir. Celles-ci peuvent se « résumer » à travers deux grands principes :

- Une interaction indirecte se caractérise normalement par une interaction se faisant par le biais d'un changement d'état observable. Dans le concept multi-agents, chacun des agents peut avoir des perceptions sur les autres, qui font partie intégrante de l'environnement. Dans notre cas, nous disposons de deux environnements « distincts » et les interactions ne dépendent pas d'un changement d'état des entités. Effectivement, les opérations à effectuer sont transmises directement sans avoir besoin de passer par les perceptions

propres de chaque entité.

- Une interaction directe est envoyée à un destinataire précis pour influencer son comportement. Elle est ponctuelle et n'affecte pas l'environnement. Elle se fonde sur l'échange de messages et d'informations qui permettront à une entité d'adopter un comportement particulier. De nombreux moyens existent pour mettre en place ce type d'échange, avec des canaux et des supports de différentes sortes. Nous y reviendrons par la suite.

Par analogie, nous pourrions comparer le système que nous devons concevoir à la situation suivante : imaginons deux amis, l'un en fauteuil roulant et l'autre qui est aveugle le pousse. La personne à mobilité réduite doit guider son partenaire en fonction de ce qu'il perçoit. Quand une information lui est communiquée, le non-voyant prend une décision en conséquence et effectue le déplacement.

1.5 Les outils utilisés

- **ArchLinux** : Dans la mesure ou nous avions déjà effectué quelques travaux ensemble, nous disposons du même système d'exploitation qui se trouve être ArchLinux, en 64 bits. Lors de notre première rencontre avec les encadrants, nous les avons questionnés sur les outils que nous aurions besoin d'utiliser pour nous assurer que ce système permettrait bien d'assumer les exigences de ce travail et notamment l'installation des logiciels requis.
- **NetLogo** : Selon sa propre documentation, il «est un environnement de modélisation programmable permettant de simuler des phénomènes naturels et sociaux. Il a été créé par Uri Wilenski en 1999 et son développement est poursuivi de manière continue par le Center for Connected Learning and Computer-Based Modeling. » Au delà de cette simple définition, Netlogo permet, pour sa modélisation, d'utiliser les concept « agent » et « multi-agent ». Ces concepts sont déjà implémentés et peuvent êtres manipulés simplement par des utilisateurs de tout niveaux via les fonctions prévues à cet effet.
- **V-REP** : De son nom entier « Virtual Robot Experimentation Platform » ce programme est, selon sa propre définition, le simulateur disposant du plus grand nombre de fonctions et fonctionnalités ainsi que l'API la plus étayée. Chacun de ses objets peut être manipulé indépendamment des autres, via des scripts appelés « contrôleurs » pouvant êtres écrits

en C/C++, Python, Java, Lua, Matlab, Octave ou Urbi. Aussi, ce logiciel est parfaitement interopérable dans le sens où il peut être installé et utilisé sur tous les systèmes d'exploitation. Enfin, cet outil comprend un plugin pouvant gérer des nœuds ROS, ce qui nous permettra par la suite de mettre en place l'interconnexion de plusieurs simulateurs, véritable objectif de ce projet.

- **ROS** : Robot Operating System est un système d'exploitation appliqué à la robotique. Il comprend un ensemble de fonctionnalités permettant de simplifier la création d'applications. Complexes, robustes elles peuvent être développées sur une large variété de logiciels de simulation robotique. Il va nous permettre de mettre en place l'interconnexion de plusieurs simulateurs par la gestion de nœuds de communication.
- **Arduino** : Arduino est un circuit imprimé Open Hardware permettant d'analyser et de délivrer des signaux électriques. Les plans sont donc disponibles sous licence libre mais certains de ses composants comme le micro-contrôleur ne le sont pas. Nous avons voulu utiliser cette plate-forme pour construire notre propre machine de Braitenberg et ainsi observer concrètement où se situaient les processus de perception-décision-action.

Nous reviendrons plus en détails sur chacun de ces outils dans la seconde partie du dossier en explicitant précisément le rôle de chacun. Après quelques rencontres avec l'équipe d'encadrement, il a été décidé que les objectifs tels qu'ils étaient présentés seraient évolutifs. L'intérêt premier de notre travail étant de mettre en place un couplage d'informations entre deux entités indépendantes l'une de l'autre.

2 Travaux réalisés

2.1 La séparation des tâches "Perception / Décision / Action"

Nous avons vu précédemment où se situaient les processus de perception décision action dans un véhicule de Braitenberg. Maintenant, il faut mettre au point des scénarios pour faire en sorte de simuler une lumière dans un monde virtuel et que l'action de l'agent se produise dans le monde réel. Pour cela, trois scénarios de séparation de la boucle perception-action ont été étudiés :

2.1.1 Couplage : Perception-décision d'un côté ; Action de l'autre.

Dans ce premier scénario, nous avons envisagé de réunir ensemble les « briques » de la perception et de la décision dans le monde simulé, et de placer l'action au niveau du robot.

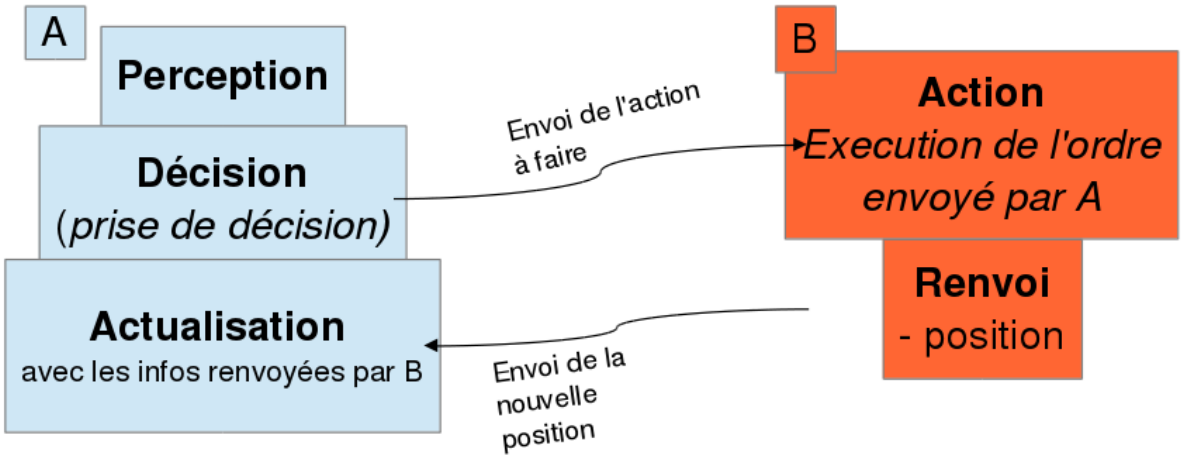

Figure 6. Schéma de séparation n ° 1

La lumière simulée est d'abord perçue par les capteurs du véhicule virtuel. La fonction de décision intervient ensuite en affectant la valeur de l'intensité du capteur gauche au moteur droit et inversement. Le but de la décision est de transmettre ces informations au processus d'action qui va s'occuper de faire tourner les moteurs gauche et droit avec les valeurs associées. Le robot B devra ensuite renvoyer sa nouvelle position à la machine A pour qu'elle actualise sa position dans le simulateur. Une fois que la position est actualisée, la boucle reprend avec la perception de nouvelles intensités lumineuses.

2.1.2 Couplage : Perception d'un côté ; Décision-Action de l'autre.

Dans un second temps, nous avons imaginé isoler la perception des deux autres processus :

Ce scénario est pratiquement le même que le premier, en dehors du fait que ce n'est plus la décision que A envoie à B, mais les valeurs reçues par les capteurs de luminosité gauche et droit. La machine réelle s'occupe ensuite d'affecter les valeurs reçues aux moteurs.

C'est l'option que nous avons retenue. Il nous a semblé plus logique d'intégrer les décisions dans le robot plutôt que dans le simulateur. En effet, celui-ci devant évoluer dans le monde réel, il

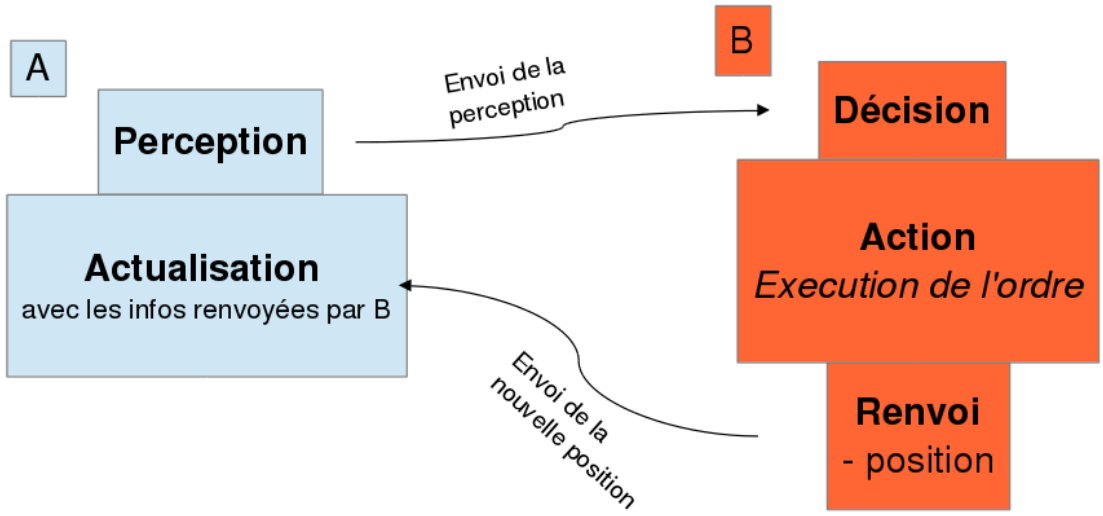

Figure 7. Schéma de séparation n ° 2

serait plus cohérent de pouvoir paramétrer son processus décisionnel en fonction des contraintes qui en découlent.

2.1.3 Séparation de chacun des processus dans une machine différente.

Avec ce scénario, nous avons poussé la logique au plus loin en se disant qu'il était tout à fait possible de capter les perceptions lumineuses dans un simulateur, qui les envoie ensuite à une autre machine qui fera office de processus de décision. Une fois la décision effectuée, elle est ensuite transmise au robot du monde réel pour que l'action s'exécute. Enfin, comme dans les autres scénarios, la nouvelle position du robot réel est envoyée au simulateur afin qu'il positionne la machine simulée d'après le déplacement qu'il a opéré.

La phase d'actualisation est très importante dans tous les cas puisqu'il faut replacer le véhicule de Braitenberg virtuel selon le déplacement qu'a effectué le véhicule réel. Sans cela, les perceptions resteraient les mêmes et le déplacement ne serait pas représentatif d'une machine de Braitenberg. C'est ainsi que nous nous sommes heurtés au format de la nouvelle position. Comment la machine qui exécute l'action sait où elle se trouve ? On pourrait très bien l'équiper d'un capteur GPS, mais la mesure de la position n'est pas très précise. De plus, un capteur GPS ne fonctionne pas à l'intérieur d'un bâtiment. C'est pour l'instant un problème qui reste sans solution technique.

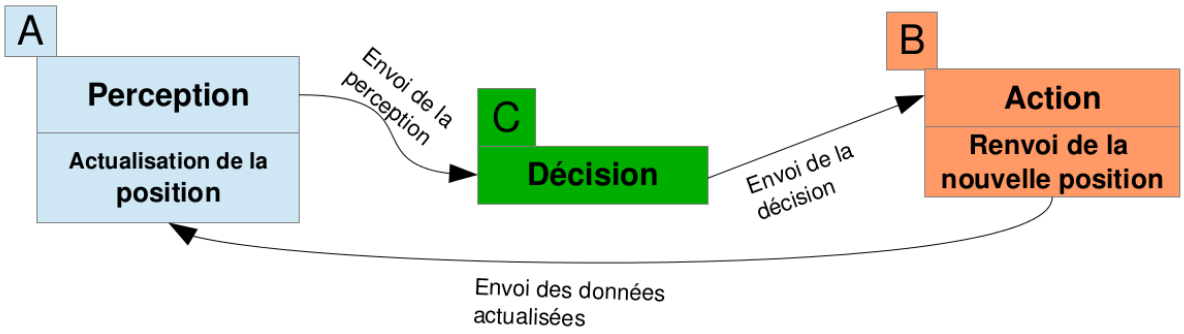

Figure 8. Schéma de séparation n ° 3

2.2 Réalisation de la machine de Braitenberg

2.2.1 Développement avec le langage "Bash"

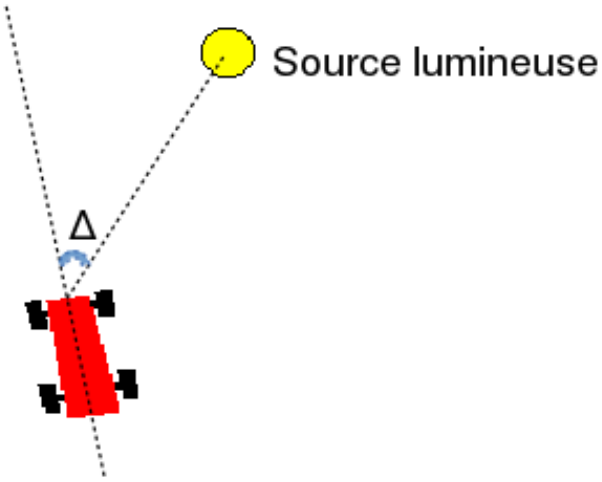

Figure 9. Représentation de la machine de Braitenberg sous Bash

Pour pouvoir simuler une machine de Braitenberg, il est important de savoir comment la programmer afin d'identifier la marge qu'il pourrait y avoir entre la théorie et la pratique. Afin de répondre à cette question, nous avons adapté un script bash existant [2] qui reproduit un comportement agressif. Il est évident que cela n'est pas la seule manière de développer une

2. http ://wiki.tcl.tk/9047

machine de Braitenberg, néanmoins le langage de script bash permet de le faire plus rapidement qu'avec d'autres langages de script ou de programmation. Nous avons alors constaté la difficulté de modéliser le plus fidèlement possible un véhicule de Braitenberg. Dans un premier temps, il a fallu modéliser la perception des capteurs. La lumière est représentée dans le programme par un cercle jaune qui ne délivre pas d'intensité lumineuse. Nous avons alors décidé de capter la lumière comme une distance entre le véhicule et la source lumineuse. Au delà d'une distance arbitraire, le véhicule ne perçoit rien. Passé en dessous de celle ci, ses deux capteurs mesurent lequel d'entre eux est le plus proche afin de voir quelle roue est actionnée. Si le capteur gauche est le plus proche, il actionnera la roue droite et inversement. La valeur de la vitesse est ensuite inversement proportionnelle à la distance entre les capteurs et la source lumineuse. Dans un deuxième temps, la trajectoire prise par le véhicule ne reflète pas non plus la réalité puisque celle ci s'effectue selon un angle Δ (voir Figure 9), et non d'après la vitesse transmise aux roues. Le programme fonctionne finalement comme suit : la voiture roule à une vitesse moyenne en ligne droite. Il est possible de lui mettre des sources lumineuses à proximité en cliquant sur le terrain. Si le véhicule détecte la lumière, il se dirige alors vers elle en accélérant. Enfin, il peut rouler sur la source lumineuse plusieurs fois avant que celle ci ne disparaisse.

Nous avons vu que pour modéliser une vraie machine de Braitenberg, il faudrait principalement modéliser les lois de la physique (gravité, frottement, circonférence des roues etc). Il aurait fallu également modéliser une source de lumière diffuse, dans le sens ou son intensité s'atténue avec la distance. Le véritable objectif n'était donc pas de développer un simulateur, mais d'identifier la boucle de perception/action. Ce qui n'a malheureusement pas été possible du fait du manque de fidélité de la modélisation. Mais nous avons usé d'autres moyens à notre disposition pour appréhender cette boucle.

2.2.2 Développement avec "Arduino"

Afin d'identifier concrètement la boucle perception action dans un véhicule de Braitenberg, nous avons décidé d'en fabriquer un avec le circuit imprimé Arduino. Nous avons donc branché deux photorésistances [3] afin de capter la luminosité. Pour ce qui est des moteurs, nous avons décidé d'utiliser le rotoshield [4] fourni par Snootlab [5]. Un shield est un circuit électronique que

3. http ://snootlab.com/composants/97-photoresistance.html
4. http ://snootlab.com/shields-snootlab/87-rotoshield-fr.html
5. http ://snootlab.com/

l'on branche sur l'Arduino et qui offre de nouvelles capacités suivant le type de shield. Ici, le rotoshield permet de contrôler jusqu'à quatre moteurs à courant continu ou deux moteurs pas à pas. Il possède un quadruple pont en H avec le circuit L293DNE [6]. Plus simplement, le rotoshield nous permet de changer le sens de rotation des moteurs sans avoir à inverser leurs polarités. Les moteurs sont pilotables via l'utilisation d'une librairie développée par le concepteur. Le langage utilisé pour l'Arduino est le Processing [7](basé sur JAVA). Sans entrer dans les détails du code fournis en annexe, nous pouvons néanmoins indiquer où se situent les processus de perception/décision/action. Voici la perception dans un premier temps :

```
1  // on lit la valeur du capteur gauche.
2  gauche = analogRead(photocellgauche);
3
4  // on lit la valeur du capteur droit.
5  droit = analogRead(photocelldroite);
```

Dans un deuxième temps arrive la prise de décision :

```
6  /*
7  * La valeur du capteur varie entre 0 et 1023, il faut l'adapter
8  * pour la vitesse qui ne prend qu'une valeur entre 0 et 255
9  */
10 gauche = gauche/4;
11 droit = droit/4;
12
13 //  Affectation des vitesses
14 Motor_D.setSpeed(gauche);
15 Motor_G.setSpeed(droit);
```

Puis arrive le moment de l'action, là où l'agent transmet l'action à faire à ses effecteurs (moteurs) :

6. http ://www.ti.com/lit/ds/symlink/l293d.pdf
7. http ://www.processing.org/

```
16 // fait tourner le moteur droit pour une marche avant
17 Motor_D.run(FORWARD);
18
19 // fait tourner le moteur gauche pour une marche avant
20 Motor_G.run(FORWARD);
```

2.2.3 Développement avec "NetLgog"

Le concept que nous avons créé sous netLogo n'est pas véritablement une machine de Braitenberg mais en simule le comportement. Ce travail à été effectué par analogie sur la base d'une librairie présente par défaut dans le programme. [8] Au démarrage de la simulation, la machine est positionnée aléatoirement dans l'environnement, la source de lumière devant impérativement rester fixe. Lorsque le programme démarre, notre agent commence à se déplacer en s'orientant vers la lumière (figure 10).

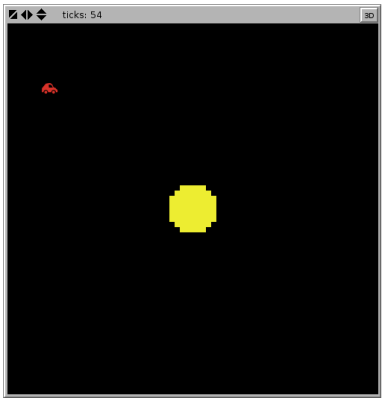

Figure 10. Représentation graphique de la machine

Pour que la machine puisse se diriger, nous avons cartographié l'environnement de façon à permettre une prise de décision. L'environnement de NetLogo est en réalité une sorte de grille sur laquelle il est possible de se déplacer par le biais de coordonnées en abscisses et en ordonnées. À l'initialisation de la simulation, un poids est attribué à chacune des cases à la manière d'une tache d'encre (figure 11) :

8. Pour y accéder : *File* ⟶ *Model Library* ⟶ *Sample Model* ⟶ *Biology* ⟶ *Ants*

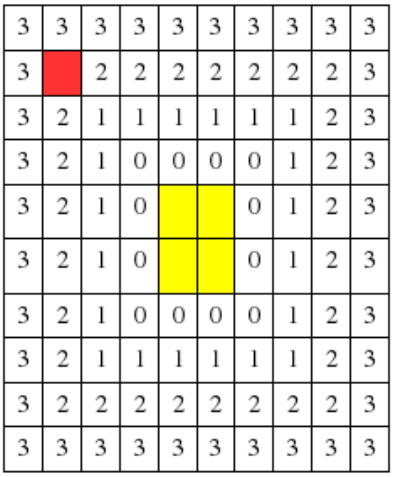

Figure 11. Effet tache d'encre

La case en rouge représente la machine qui, pour se déplacer, va consulter les valeurs contenues dans les cases environnantes et se rendre sur celle qui a le poids minimum. Le comportement obtenu simule bien le comportement d'une machine de Braitenberg et ce n'est que la stratégie pour l'obtenir qui s'en écarte.

Ce travail a été réalisé tout au début du projet, afin de mieux appréhender la notion d'agent intelligent. Le véritable intérêt de cet exercice résidait donc dans le fait de parvenir à un comportement utilisant une boucle de 3 fonctions : Perception (consulter les cases environnantes), Décision (Choisir la case ayant le poids le plus faible) et Action (Se rendre sur la case choisie).

2.2.4 Développement avec "V-REP"

Une fois le concept d'Agent Intelligent et le fonctionnement d'une machine de Braitenberg compris, nous avons pu envisager de travailler avec le logiciel de simulation robotique V-REP. Sur les conseils de l'un des membres de l'équipe d'encadrement, nous avons suivi un tutoriel pour acquérir quelques bases sur l'utilisation de ce programme. À l'aide de ces connaissances et d'une machine dysfonctionnelle que nous avons trouvée sur un forum [9], nous sommes parvenus à créer la machine suivante (figure 12) :

9. http ://www.forum.coppeliarobotics.com/viewtopic.php ?f=9&t=394 : La boucle perception action était manquante et la machine tournait en rond

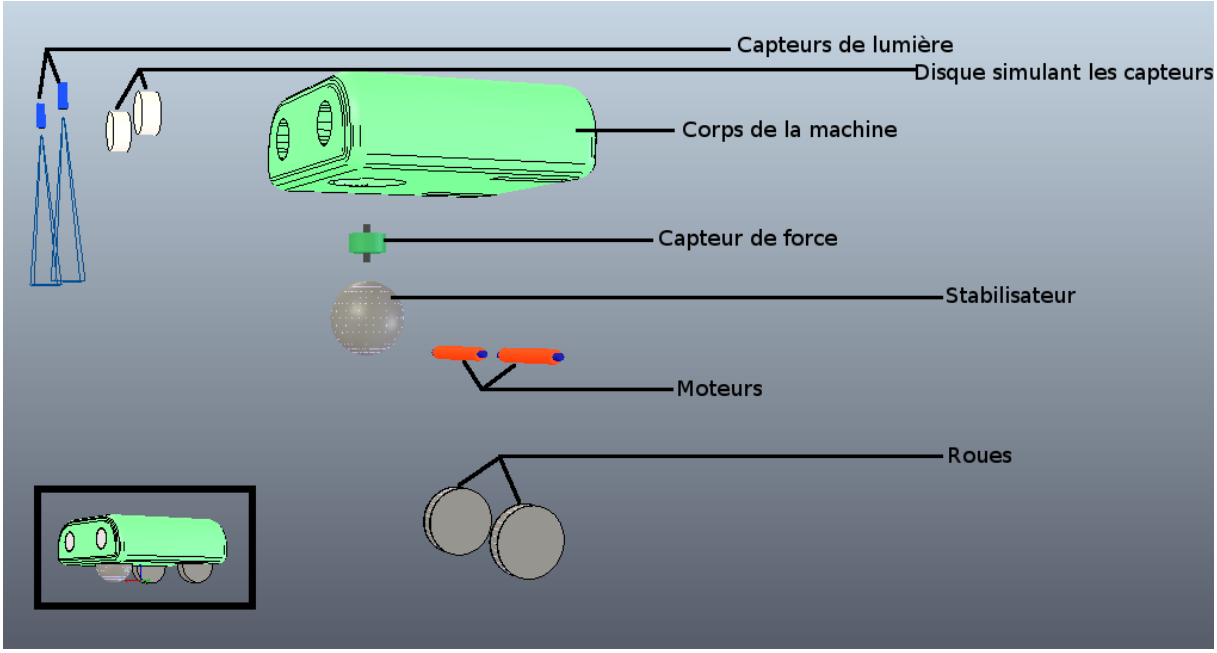

Figure 12. Machine de Braitenberg sous V-REP

Le logiciel V-REP simule un grand nombre de lois physiques pour se rapprocher au maximum de la réalité. La première difficulté à été de vérifier que les différentes propriétés de la machine étaient bien configurées. En cochant les options prévues à cet effet, nous avons fait en sorte que les éléments externes de la machine respectent ces différentes caractéristiques :

OPTION V-REP	NOM EXPLICITE	EFFET	PIÈCES
Collidable	Détecteur de collision	Évite de passer au travers des autres éléments *Collidable*	**Corps** **Stabilisateur** **Roues** **Fixations**
Measurable	Détecteur de distance	Permet de calculer une distance minimum pour éviter les autres éléments *Measurable*	
Renderable	Visibilité	Les éléments *Renderable* peuvent être perçu par les *Vision Sensors*, comme les caméras.	
Detectable	Détecteur de proximité	Les éléments *détectable* peuvent être perçus pas les *proximity sensors*, comme les capteurs Ultra-Son.	

Dans la mesure où tous ces éléments ont un contact avec le monde extérieur, nous avons trouvé plus cohérent de leur donner ces propriétés. Les autres pièces ne servant en réalité qu'à

réaliser une tâche précise, elle ne sont pas apparentes lors de la simulation (moteurs, capteur de force). Après les avoir liées aux objets sur lesquels elles agissent (Moteur pour les roues, Capteur de force pour le stabilisateur et Capteur de luminosité pour les disques) nous avons ensuite rendues invisibles différentes pièces dont la visibilité est facultative. Une fois la vitesse des moteurs et la sensibilité des différents capteurs paramétrées, nous avons pu programmer son comportement. Voici le code commenté [10] de l'implémentation :

```
1  # Premier passage : initialisation des variables
2  if (simGetScriptExecutionCount()==0) then
3          # Recuperation des controleurs de moteur
4          leftmotor=simGetObjectHandle("left_motor")
5          rightmotor=simGetObjectHandle("right_motor")
6
7          # Recuperation des controleurs de capteurs
8          leftsensor=simGetObjectHandle("left_lightsensor")
9          rightsensor=simGetObjectHandle("right_lightsensor")
10 end
11
12 # PERCEPTION
13 leftresult,data1=simReadVisionSensor(leftsensor)
14 rightresult,data2=simReadVisionSensor(rightsensor)
15
16 # DECISION
17 speed_left = data2[1]*5
18 speed_right = data1[1]*5
19
20 # ACTION
21 simSetJointTargetVelocity(leftmotor,speed_left)
22 simSetJointTargetVelocity(rightmotor,speed_right)
```

Les capteurs de luminosité sont en fait des caméras. Pour obtenir une valeur de luminosité,

10. Pour permettre la coloration syntaxique les "- -" servant à faire des commentaires en Lua ont été remplacés pas des "#"

il faut l'extraire du flot de données qui est capturé par les caméras. En effet, la structure de données renvoyée par une caméra est un tableau de quinze valeurs flottantes.
Après quelques test, il reste encore quelques incohérences dans divers comportements mais ils ne méritent pas que l'on s'attarde dessus. Il s'agit de petites choses qui relèvent plus de la finalisation que d'une erreur de conception du projet en lui même. En l'occurrence, la machine continue d'avancer en permanence car elle perçoit la lumière ambiante. Sa vitesse restant très lente dans ce cas de figure, nous parvenons tout de même à obtenir un comportement satisfaisant en utilisant un spot dont l'intensité est bien plus importante.

2.3 Déploiement des concepts

2.3.1 Séparation de l'agent dans V-REP

Dans cette partie, nous allons voir la mise en œuvre du second scénario avec le simulateur V-REP. Comme expliqué plus tôt, nous avons modélisé une machine de Braitenberg avec deux caméras et deux moteurs. Le fonctionnement du véhicule en tant qu'agent réactif développé, nous pouvions tenter de séparer la perception de la décision et de l'action. Pour cela, nous avons dupliqué notre machine pour en obtenir une seconde et disposer chacune d'entre elles de part et d'autre de la scène. Pour s'assurer qu'aucune interférence n'existe, nous avons également placé un mur entre les deux.
Étant donné que les deux machines sont présentes dans la même scène, il est possible d'accéder aux objets qui les contrôlent directement dans le code source.

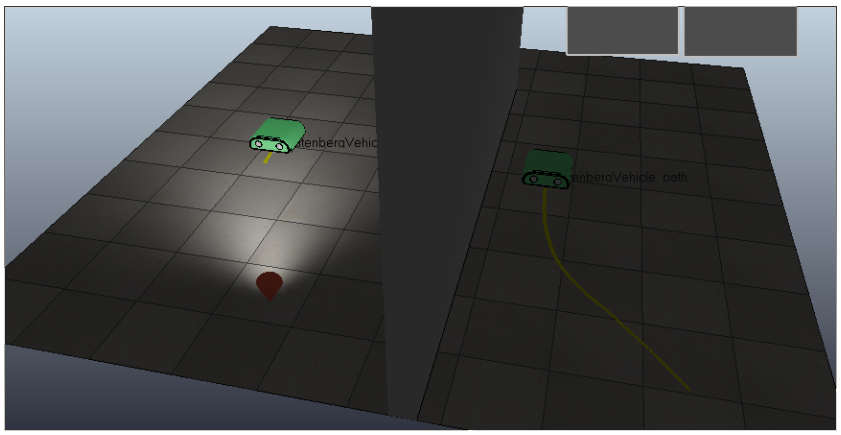

Figure 13. Séparation de l'agent

Chaque machine possède son propre fonctionnement, et donc son propre script Lua. Le véhicule présent à gauche va s'occuper de la perception, et la machine de droite, quant à elle, se verra assumer les processus de décision et d'action.
Le code implémenté du véhicule de droite signifie que nous allons accéder aux caméras du véhicule de gauche. Par ce biais, nous récupérons directement la perception de la machine de gauche. Il suffit ensuite d'affecter les valeurs de la luminosité aux moteurs.
Le principe est le même pour la machine de gauche. Ce dont nous avons besoin pour elle, c'est de récupérer la position et l'orientation de la machine à droite. Pour cela, on utilise les fonctions : *simGetObjectPosition(objet_recup,referentiel)* et *simGetObjectOrientation(objet_recup,referentiel)*.
Le paramètre *obj_recup* est l'objet qui gère le véhicule pour lequel nous voulons récupérer la position/orientation. Le paramètre est référentiel, autrement dit c'est un objet qui permet d'obtenir la position/orientation. Dans notre cas, l'objet récupéré est le robot et son référentiel est le sol.
La fonction *simGetObjectPosition(objet_recup,referentiel)* renvoie un tableau de trois flottants contenant respectivement des coordonnées X,Y et Z. Les valeurs retournées par l'autre fonction *simGetObjectOrientation(objet_recup,referentiel)* sont regroupées dans un tableau et représentent les angles d'Euler, utilisés pour décrire l'orientation d'un solide.

2.3.2 Mise en place avec ROS

Comme nous l'avons défini plus haut, ROS est l'outil que nous avons utilisé pour aborder le problème de l'interconnexion entre nos deux entités. Comme pour V-REP, ROS est un programme relativement complexe, nous avons aussi suivi un tutoriel pour appréhender ses fonctionnalités de base. Celui qui nous a été conseillé reprend conceptuellement ce que nous devons mettre en place à la différence que les deux entités sont le clavier d'une part, et une tortue (nous l'appellerons Turtlesim) à déplacer dans un environnement « vierge » d'autre part. Théoriquement, une fois le tutoriel réalisé, il nous suffira de remplacer les deux entités par les nôtres et de modifier le type d'information échangé.

- **Les nœuds**

Pour contrôler un robot, ROS permet de mettre en place un système de commandes à base de nombreux nœuds ayant chacun une fonction bien précise. Seulement, notre situation n'est pas aussi complexe que celle du fonctionnement global d'un robot. Nous ne créerons donc que deux nœuds au sein desquels seront traitées des informations que nous aurons pris soin d'organiser. Aussi, en fonction de la situation, une entité est soit l'émetteur soit le récepteur de l'information. En quelques lignes de commandes, nous avons rapidement obtenu la fenêtre voulue et vérifié que le nœud était correctement créé.

- **Les topics**

Les Topics sont des sortes de rubriques auxquelles vont pouvoir « s'inscrire » nos deux entités soit pour y publier des informations soit pour en récupérer. Il n'y a pas nécessité pour celle qui récupère les informations de savoir d'où elles proviennent, du moment que le topic choisi est le bon. De la même manière, celle chargée de publier des informations n'a pas besoin de savoir qui va les récupérer. La communication au sein d'un topic est unilatérale. En multipliant les topics, il devient très simple de multiplier les informations échangées. Il est important de comprendre ici que les machines peuvent jouer 2 rôles distincts :

— Publisher : envoie des informations à travers un topic

— Subscriber : Récupère les informations sur un topic

- **Organisation**

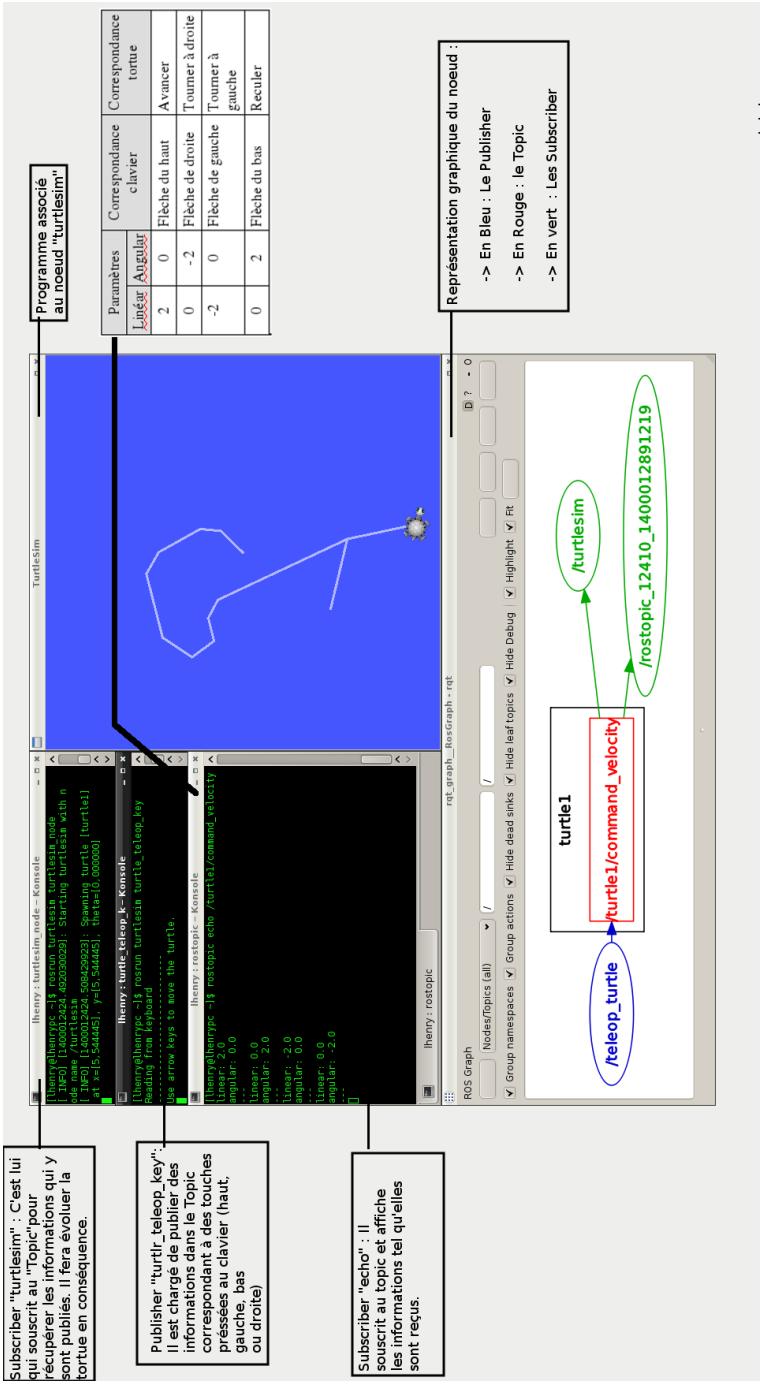

Figure 14. Principe de fonctionnement des nœuds ROS

- **Notre implémentation**

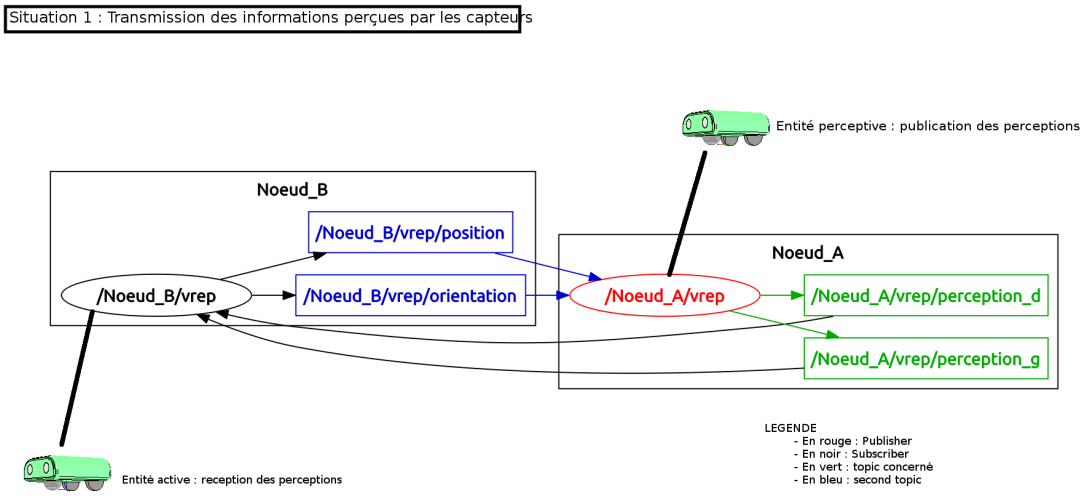

Figure 15. Envoi de perception

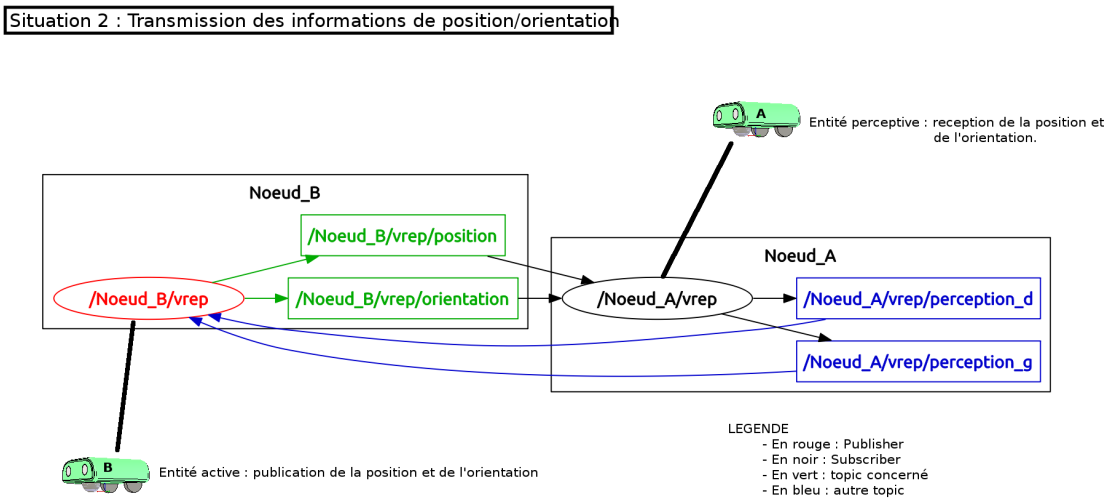

Figure 16. Envoi des position/orientation

Conclusion

La conclusion de ce rapport nous amène à se questionner sur le travail accompli durant ces cinq derniers mois. Les objectifs du projet étaient volontairement assez larges afin de laisser une bonne marge de manœuvre. Le principal objectif, lui, a été atteint puisqu'il s'agissait de pouvoir séparer les différents processus de perception-décision-action en deux, voir trois « semi-agents ». On aurait évidemment pu développer directement une méthode permettant à deux agents de communiquer leurs perceptions et leurs positions, mais la solution conceptuelle n'aurait pas permis d'obtenir un système générique d'interactions. Maintenant, il est conceptuellement possible de remplacer une machine de Braitenberg par d'autres modèles d'agents. Par ailleurs, la communication de nos « semi-agents » étant formalisée avec le middleware ROS, nous avons également atteint un autre objectif. En effet, ce système étant largement utilisé dans les projets de l'équipe, il était plus que bienvenu de l'utiliser.

Ce projet a aussi été un vrai travail de recherche, puisqu'il a fallu trouver et expérimenter une méthode permettant de simuler les perceptions d'un agent réel. C'est pourquoi il a fallu utiliser des approches aussi différentes comme la simulation robotique, l'électronique à travers Arduino, ou même encore la modélisation avec Netlogo.

Comme nous l'avons déjà dit, le projet avait essentiellement un but exploratoire étant donné l'absence de certitudes quant à la finalité de celui-ci. Ce qui laisse néanmoins place à une certaine frustration au vu de l'achèvement qu'il pourrait avoir. On peut sans hésiter affirmer qu'il manque très peu de temps pour réussir à mettre en place la solution conceptuelle actuelle avec le système AA4MM développé par l'équipe MAIA. Car en effet, au fur et à mesure de l'avancement de nos travaux, il a été envisagé d'utiliser le système de couplage d'artefacts multi-agents AA4MM afin de synchroniser les interactions de nos « semi-agents ». Puisque nous n'avons pu le réaliser, cet aspect n'est pas développé dans ce rapport.

Bibliographie

[Fer95] Jacques Ferber. Les systèmes Multi Agents : vers une intelligence collective. 1995.InterEditions.

[Tho05] Vincent Thomas. Proposition d'un formalisme pour la construction automatique d'interactions dans les systèmes multi-agents réactifs. 2005. Loria.

[Von98] John Von Neumann. Théorie générale et logique des automates. 1998. Champ Vallon.

[Ast09] H. Aydt, S.J Turner, Wentong Cai, and M.Y.H. Low. Research issues in symbiotic simulation. In Simulation Conference (WSC), Proceedings of the 2009 winter, 1213-1222, 2009

[Bra86] Valentino Braitenberg. Vehicles : Experiments in synthetic psychology. MIT press. 1986.

[Jen98] N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. Journal of Autonomous Agents and Multi-Agent Systems, 1(1) :7-38, 1998.

Annexes

2.4 Script bash pour la machine de Braitenberg

```
#!/bin/sh
 #\
 exec wish "$0" ${1+"$@"}
 # Simulation de machines de Braitenberg.
 # modeles des voitures disponibles - http://wiki.tcl.tk/4364
 package require Tk
 set tcl_precision 17
 namespace eval car {
     variable cars [list]
 }
 # Create a new car:
 # params:
 #   name    - nom de la voiture
 #   c       - canvas pour dessiner
 #   color   - couleur de la voiture
 #   angle   - direction initiale du mouvement
 #   inputs  - liste des entrees
 # left input light + -> left wheel
 proc car::new {name c x y color angle inputs} {
     variable cars
     lappend cars $name
     interp alias {} $name {} car::dispatch $name
     namespace eval $name {
         variable lspeed 10 rspeed 10
         variable ldiff 0 rdiff 0
         variable inputs [list]
     }
     namespace eval $name [list variable canvas $c angle $angle]
```

```
29      interp alias {} $name: {} namespace eval ::car::$name
30      $c create line $x [expr {$y + 5}] [expr {$x + 20}] [expr {$y
            + 5}] \
31          -tag [list $name object] -width 3
32      $c create line [expr {$x - 2}] [expr {$y + 40}] [expr {$x +
           22}] \
33          [expr {$y + 40}] -tag [list $name object] -width 3
34      $c create poly [expr {$x + 2}] $y [expr {$x + 18}] $y \
35          [expr {$x + 20}] [expr {$y + 50}] $x [expr {$y + 50}]\
36          -fill $color -tags [list $name object]
37      # Creer les roues
38      wheel $name $c [expr {$x -  3}] [expr {$y + 5}]
39      wheel $name $c [expr {$x + 23}] [expr {$y + 5}]
40      wheel $name $c [expr {$x -  5}] [expr {$y + 40}]
41      wheel $name $c [expr {$x + 25}] [expr {$y + 40}]
42      canvas'rotate $c $name $angle
43      # parcours parmis les sections d entree
44      set RE \
45      {(left|right|center)\s+input\s+(.+)\s+(\+|-)\s+->\s+(left|
           right)\s+wheel}
46
47      foreach line [split $inputs \n] {
48          if {[regexp $RE $line -> iside type posneg wside]} {
49              # Creee lentree
50              lappend ${name}::inputs [list $iside $type $posneg
                    $wside]
51          }
52      }
53  }
54  proc car::wheel {name c x y {dx 3} {dy 6}} {
55      set x0 [expr {$x - $dx}]
```

```
56      set y0 [expr {$y - $dy}]
57      set x1 [expr {$x + $dx}]
58      set y1 [expr {$y + $dy}]
59      $c create poly $x0 $y0 $x0 $y1 $x1 $y1 $x1 $y0 -fill black \
60          -tag [list $name object]
61  }
62  proc car::dispatch {name cmd args} {eval ::car::$cmd $name $args
        }
63  proc car::move {} {
64      variable cars
65      global hits
66      foreach name $cars {
67          set c [$name: set canvas]
68          # etabli langle en comparant la droite et la gauche
69          # vitesse.
70          set angle [$name: set angle]
71          set rspeed [expr {[$name: set rspeed] + [$name: set
                rdiff]}]
72          set lspeed [expr {[$name: set lspeed] + [$name: set
                ldiff]}]
73          set nangle [expr {($rspeed - $lspeed)/5.}]
74          # vitesse moyenne
75          set speed [expr {($rspeed + $lspeed)/2.}]
76          canvas'rotate $c $name $nangle
77          $name: set angle [expr {$angle + $nangle}]
78          set dx [expr {-$speed * sin([$name: set angle])}]
79          set dy [expr {-$speed * cos([$name: set angle])}]
80          $c move $name $dx $dy
81          foreach {x0 y0 x1 y1} [$c bbox $name] { break }
82          if {$x1 < 0} {
83              $c move $name [winfo width $c] 0
```

```
84          }
85          if {$x0 > [winfo width $c]} {
86              $c move $name -[winfo width $c] 0
87          }
88          if {$y1 < 0} {
89              $c move $name 0 [winfo height $c]
90          }
91          if {$y0 > [winfo height $c]} {
92              $c move $name 0 -[winfo height $c]
93          }
94          # Regarde si on rentre dans la lumiere
95          foreach {x y} [canvas'center $c $name] { break }
96          set start {}
97          set closest [$c find closest $x $y 30]
98          while {$closest != $start} {
99              if {[info exists hits($closest)]} {
100                 if {[incr hits($closest) -1] <= 0} {
101                     # Item "mort"
102                     set next [$c find closest $x $y 30 $closest]
103                     $c delete $closest
104                     set closest $next
105                     continue
106                 }
107                 # Flash litem t
108                 set color [$c itemcget $closest -fill]
109                 $c itemconfigure $closest -fill white
110                 after 20 [list $c itemconfigure $closest -fill
                        $color]
111             }
112             if {![string length $start]} {set start $closest }
113             set closest [$c find closest $x $y 30 $closest]
```

```
114         }
115         # cherche lobjet le plus proche et adapte la vitesse
116         # cherche la position entre la droite et la gauche
117 
118         set hypot 25.0
119         set phi   0.411516846067
120         set rx [expr {$x + $hypot * sin([$name: set angle] +
               $phi)}]
121         set ry [expr {$y + $hypot * cos([$name: set angle] +
               $phi)}]
122         set lx [expr {$x + $hypot * sin([$name: set angle] -
               $phi)}]
123         set ly [expr {$y + $hypot * cos([$name: set angle] -
               $phi)}]
124         set cx [expr {$x + $hypot * sin([$name: set angle])}]
125         set cy [expr {$y + $hypot * cos([$name: set angle])}]
126         array set totals {
127             right,inputs    0
128             left,inputs     0
129             right,total     0
130             left,total      0
131         }
132         foreach item [$name: set inputs] {
133             foreach {iside type posneg wside} $item { break }
134             # cherche parmis tous les items
135             # (50 pixels)
136             set start {}
137             if {"$iside" == "left"} {
138                 set sx $lx
139                 set sy $ly
140             } elseif {"$iside" == "right"} {
```

```
141                 set sx $rx
142                 set sy $ry
143             } else {
144                 # Centre
145                 set sx $cx
146                 set sy $cy
147             }
148             set closest [$c find closest $sx $sy 100]
149             while {$closest != $start} {
150                 # Cherche le tag
151                 if {[lsearch [$c gettags $closest] $name] == -1}
                        {
152                     if {[lsearch [$c gettags $closest] $type] >
                            -1} {
153                         # Distance du capteur
154                         foreach {ox oy} [canvas'center $c
                                $closest] { break }
155                         set distance \
156                             [expr {150-hypot(abs($ox-$sx),abs($oy
                                    -$sy))}]
157                         # Ajouter au total
158                         set totals($wside,total) [expr $totals(
                                $wside,total) \
159                                  $posneg $distance]
160                         incr totals($wside,inputs)
161                     }
162                 }
163                 if {![string length $start]} { set start
                        $closest }
164                 set closest [$c find closest $sx $sy 100
                        $closest]
```

```
165             }
166         }
167         if {$totals(right,inputs) > 0} {
168             set dr [expr {$totals(right,total) / $totals(right,
                    inputs)}]
169             if {$dr < -100} { set dr -100 }
170             $name: set rdiff [expr {$dr / 10.0}]
171         } else {
172             $name: set rdiff 0
173         }
174         if {$totals(left,inputs) > 0} {
175             set dl [expr {$totals(left,total) / $totals(left,
                    inputs)}]
176             if {$dl < -100} { set dl -100 }
177             $name: set ldiff [expr {$dl / 10.0}]
178         } else {
179             $name: set ldiff 0
180         }
181     }
182 }
183 proc object {c x y} {
184     global source hits
185     switch $source {
186         object  { set color blue }
187         light   { set color yellow }
188         temp    { set color red }
189         oxygen  { set color cyan }
190         organic { set color green }
191 
192     }
193     set id [$c create oval $x $y [expr {$x + 20}] [expr {$y +
```

```
        20}] \
194          -fill $color -tags $source]
195     set hits($id) 15
196     $c bind $id <Button-3> [list $c delete $id]
197 }
198 proc canvas'center {w tag} {
199     foreach {x0 y0 x1 y1} [$w bbox $tag] { break }
200     list [expr {($x0 + $x1) / 2.}] [expr {($y0 + $y1) / 2.}]
201 }
202 proc canvas'rotate {w tag angle} {
203     foreach {xm ym} [canvas'center $w $tag] { break }
204     foreach item [$w find withtag $tag] {
205         set coords {}
206         foreach {x y} [$w coords $item] {
207             set rad [expr {hypot($x-$xm, $y-$ym)}]
208             set th  [expr {atan2($y-$ym, $x-$xm)}]
209             lappend coords [expr {$xm + $rad * cos($th - $angle)
                }]
210             lappend coords [expr {$ym + $rad * sin($th - $angle)
                }]
211         }
212         $w coords $item $coords
213     }
214 }
215 proc every {ms body} {eval $body; after $ms [info level 0]}
216
217 set source "light"
218 frame .s
219 label .s.source -text "Source:"
220 radiobutton .s.light    -text "Light"  -variable ::source -value
    "light"
```

```
221
222 eval [list pack] [winfo children .s] [list -side left]
223 pack .s -side top
224
225 # Creer le paysage
226 pack [canvas .c -width 600 -height 400 -bg white] -fill both -
       expand 1
227
228 car::new foo .c 200 200 red 0.2 {
229     left input light + -> right wheel
230     right input light + -> left wheel
231     left input temp + -> left wheel
232     right input temp + -> right wheel
233     left input oxygen - -> right wheel
234     right input oxygen - -> left wheel
235     left input organic - -> left wheel
236     right input organic - -> right wheel
237 }
238 bind .c <Button-1> {
239     object .c %x %y
240 }
241 bind .c <Button-3> { break }
242 every 50 {car::move}
```

2.5 Script de la modélisation NetLogo

```
1  patches-own [
2    nest?
3    nest-scent
4  ]
5
6  globals [
7    detecter          # permet de savoir si la lumiere est a gauche
       ou a droite de la machine
8    decider           #permet de savoir si l'on doit tourner a droite
         ou a gauche
9  ]
10
11 to setup
12   __clear-all-and-reset-ticks
13   set-default-shape turtles "bug"
14   crt population
15   [
16     setxy random-xcor random-ycor
17     set size 3
18     set color red
19   ]
20   setup-patches
21 end
22
23 to setup-patches
24   ask patches
25   [
26     setup-nest
27     recolor-patch
28   ]
```

```
end

to setup-nest
  set nest? (distancexy 0 0) < 5
  set nest-scent 200 - distancexy 0 0
end

to recolor-patch
  if nest?
  [
    set pcolor yellow
  ]
end

to go
  ask turtles
  [
    return-to-nest
    fd 0.2
  ]
  tick
end

to return-to-nest
  ifelse nest?
  [
    rt 180
  ]
  [
    code-agent
  ]
```

```
end

to code-agent
  perception
  decision
  action
end

to-report nest-scent-at-angle [angle]
  let p patch-right-and-ahead angle 1
  if p = nobody [ report 0 ]
  report [nest-scent] of p
end

to perception
  let scent-ahead nest-scent-at-angle   0
  let scent-right nest-scent-at-angle  45
  let scent-left  nest-scent-at-angle -45
  if (scent-right > scent-ahead) or (scent-left > scent-ahead)
  [ ifelse scent-right > scent-left
    [ set detecter 0 ]
    [ set detecter  1 ] ]
end

to decision
  set decider detecter
end

to action
  ifelse decider = 0
    [ rt 2 ]
```

```
  [ lt 2 ]
end
```

2.6 Code de l'implémentation avec Arduino

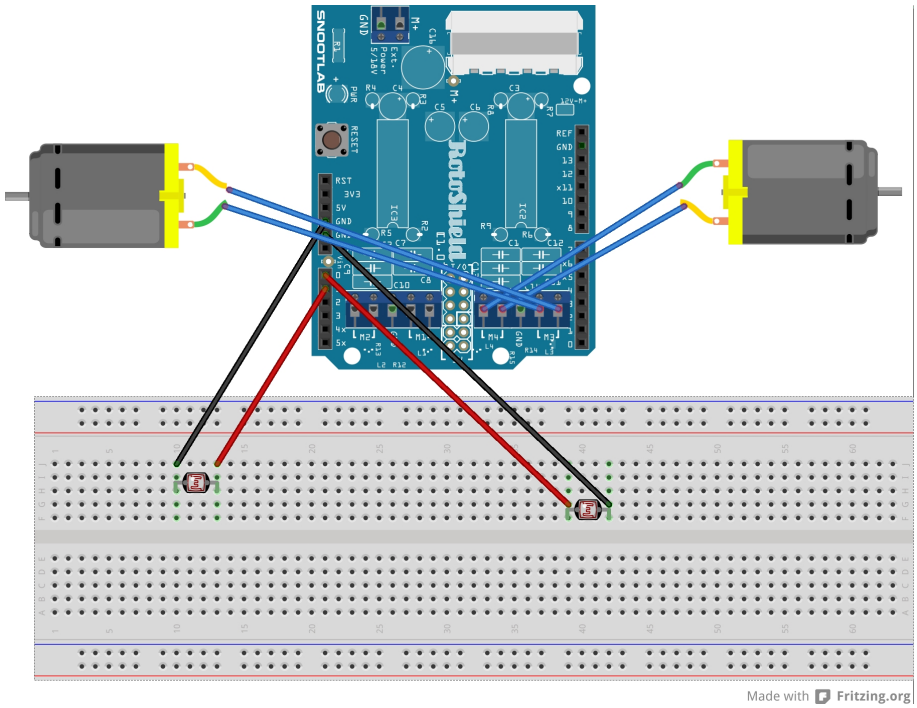

Figure 17. Cablâge de la machine de Braitenberg sur Arduino

```
1  // need to include this first
2  #include <Wire.h>
3
4  // this is the snootor library
5  #include <snootor.h>
6
7  // Declaration of 2 motor objects
8
9  SnootorDC Motor_G;
10 SnootorDC Motor_D;
11 int photocellgauche = 0; // connexion sur A0
12 int photocelldroite = 1; //connexion sur A1
13 int droit;
```

```
14 int gauche; // resultat de la lecture analogique
15
16 void setup(){
17
18 // Initialisation des entrees sorties
19 Serial.begin(115200);
20  Wire.begin();
21   Motor_D.init(3);
22   Motor_G.init(4);
23   Serial.println("init ok");
24 }
25
26 void loop(){
27   SC.dump();
28   //PERCEPTION
29   gauche = analogRead(photocellgauche);
30   droit = analogRead(photocelldroite);
31   //DECISION
32   Serial.print("Valeur brute :");
33   Serial.println(gauche);
34   gauche = gauche/4;
35   droit = droit/4;
36   Serial.print("Valeur transformee :");
37   Serial.println(gauche);
38   Motor_D.setSpeed(gauche);
39   Motor_G.setSpeed(droit);
40   //ACTION
41   Motor_D.run(FORWARD);
42   Motor_G.run(FORWARD);
43 }
```

2.7 Code de l'implémentation avec V-REP et ROS

2.7.1 Machine perceptrice

```
# Si c'est la premiere execution
if (simGetScriptExecutionCount()==0) then
        # Controleur de la machine
        BraitenbergPer = simGetObjectHandle("BraitenbergP")

        # Controleurs des capteurs luminosite
        leftsensor=simGetObjectHandle("left_lightsensorP")
        rightsensor=simGetObjectHandle("right_lightsensorP")

        # Etablissement des publishers de perceptions
        perceptg = simExtROS_enablePublisher("perception_g",1,
           simros_strmcmd_get_float_signal,-1,-1,"captg")
        perceptd = simExtROS_enablePublisher("perception_d",1,
           simros_strmcmd_get_float_signal,-1,-1,"captd")

        # Etablissement du subscriber pour les positions
        result = simExtROS_enableSubscriber('/Noeud_B/vrep/
           Position'1, simros_strmcmd_set_object_pose,
           BraitenbergPer,-1,'sign_d')
end

# Perception => Recuperation des informations des capteurs
leftresult,data_l=simReadVisionSensor(leftsensor)
rightresult,data_r=simReadVisionSensor(rightsensor)

# Envoi des perception dans la noeud ROS
simSetFloatSignal("captg",data_l[1])
simSetFloatSignal("captd",data_r[1])
```

2.7.2 Machine de décision/action

```
1  # Si c'est la premiere execution
2  if (simGetScriptExecutionCount()==0) then
3          # Controleur de la machine
4          machine = simGetObjectHandle("Braitenbergbot")
5
6          # Controleurs des moteurs
7          leftmotor=simGetObjectHandle("left_motor")
8          rightmotor=simGetObjectHandle("right_motor")
9
10         # Etablissement des publishers pour les positions
11         local effectiveTopicName=simExtROS_enablePublisher('
              Position',1,simros_strmcmd_get_object_pose,machine,-1,'
              ')
12
13         # Etablissement des subscriber pour les perceptions
14         result = simExtROS_enableSubscriber('/Noeud_A/vrep/
              perception_g',1,simros_strmcmd_set_float_signal,-1,-1,'
              sign_g')
15         result = simExtROS_enableSubscriber('/Noeud_A/vrep/
              perception_d',1,simros_strmcmd_set_float_signal,-1,-1,'
              sign_d')
16 end
17
18 # Decision => Calcul des vitesses
19 vitesse_d = simGetFloatSignal('sign_g')*10
20 vitesse_g = simGetFloatSignal('sign_d')*10
21
22 # Action => Accplication des vitesses au moteur
23 simSetJointTargetVelocity(leftmotor,vitesse_g)
24 simSetJointTargetVelocity(rightmotor,vitesse_d)
```