Petit document sur le Robot Réflexif (Behavior)
Explication du code

Alexandre Saidi
ECL-MI
07-08

2008-02-23

Ce document décrit le code C/C++ de simulation du comportement d'un Robot selon la stratégie de conception **réactive**.
Il est un complément du support des présentations effectuées dans le cadre des PEs 47 et 51 (2007-2009).

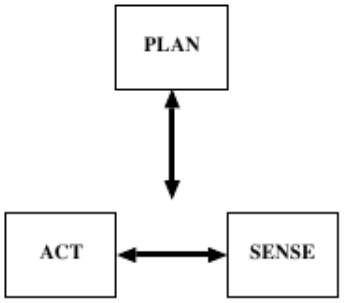
Note : _____

Table des matières

Le Robot

I Introduction

Le code fourni décrit la conception et l'implantation (en langage C) du Robot par la méthode de conception Réactive (Réflexive) .



c.

Dans cette méthode, le couple **Actuator-Sensor** (actuateur-capteur) domine la conception et la partie **Plan** (planification) devient secondaire.

Contrairement au modèle centralisé où les ordres sont décidés et transmis par un cerveau (Centre de Planification), le Robot est concu et composé d'un ensemble d'organes (un organe peut être une carte munie d'un PIC) indépendantes qui se comportent comme des *agents*.

Selon la tâche assignée à cet agent (par exemple, éviter des obstacles à l'aide de capteurs IRs), le Robot surveille son environnement à l'aide des capteurs et émet directement des commandes en réaction à cet environnement.

Par exemple, si un obstacle est en vue, la carte émet la commande TURN_LEFT, TURN_RIGHT, etc. La commande émise est transmise à un cerveau qui, en fonction d'autres éléments, décidera. Rappelons que dans une approche centralisée, la présence de l'obstacle est simplement signalée au "chef" (carte centrale) qui, elle, décide ou non d'émettre une commande au moteurs.

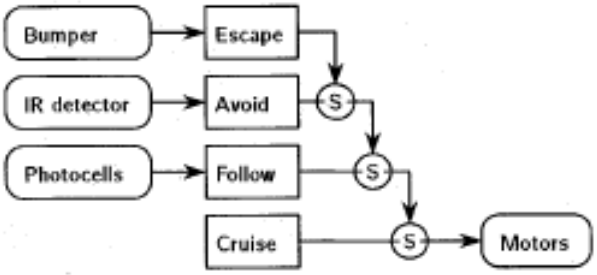
Le robot dispose de plusieurs organes fonctionnant en parallèle chacun transmettant ses commandes. Par exemple, la carte d'évitement émet TURN_LEFT alors que la carte de surveillance du niveau de la batterie émet TURN_RIGHT. Il faudra donc un arbitrage des commandes émises par différents agents permettant la mise en oeuvre d'une stratégie qui tient compte des priorités (voir plus loin).

Comme on peut le remarquer dans le schéma, une planification générale peut avoir lieu en amont et/ou pendant l'évolution du Robot dans l'environnement. Par exemple, le plan du milieu peut être constitué au départ puis mise à jour au fur et à mesure des découvertes d'obstacles par le Robot.

Habituellement, la cadence de ces mises à jours est bien moindre (e.g. toutes les 5 secondes) vs. celle d'émission des commandes par l'agent (plusiers commandes par seconde).

II L'architecture

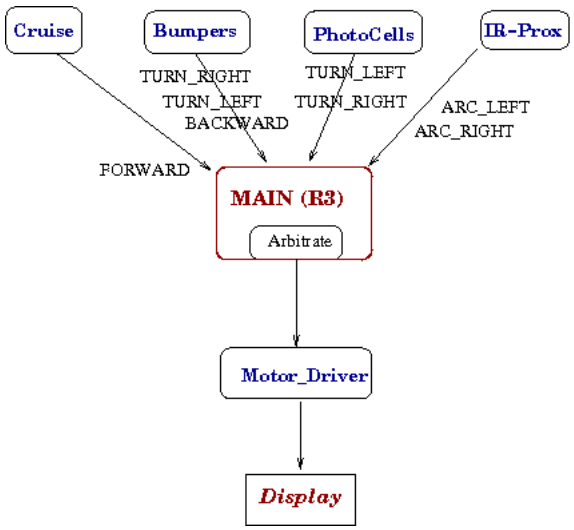
L'activité du Robot R3[1] est donné par le schéma suivant.



Ce schéma est largement détaillé dans les présentations faites aux élèves, dans le cadre des PEs.

III La Programmation

Découpage en processus.



[1] Robot dans sa 3e version, les 2 premières étaient munis chacun d'une seule carte

III.A Le processus principal

```
1  Recupération des paramètres d'appel
2  Initialisation des paramètres  : position, direction,
       obstacles, source lumineuse, taille échiquier, ...
3
4  Séquence de création et de lancement des processus :
5      - Driver moteur : déplacement du robot
6      - Cruise : toujours en avant toutes
7      - Follow : suivre une source lumineuse
8      - Avoid : éviter les obstacles et les bords de l'
           échiquier (à l'aide des IRs)
9      - Escape : éviter les obstacles et les bords de l'
           échiquier (à l'aide des Bumpers + IRs)
10     - Affichage : affichage du robot sur l'échiquier
11
12 Ces processus sont lancés par le "Pere" qui  lui même prend
       en charge le processus Arbitrate
```

IV Description des processus

A noter :

- Chaque processus simule le comportement d'une carte indépendante.
- Chaque processus émet un drapeau (true/false) et une commande (TURN_RIGHT, BACKWARD, etc.). Cette commande n'est prise en compte (par l'arbitre, pour une décision finale) que si le drapeau=true.

IV.A Avoid

Ce processus utilise deux capteurs de proximité infra rouges.

Dans la simulation, l'état de ces capteurs est défini en fonction des coordonnées $< ligne, Colonne >$ du Robot, sa direction ainsi que l'emplacement des obstacles (et les bords de l'échiquier).

Selon ces coordonnées, on définit les valeurs 0..3 émises par les capteurs IRs où :

- 0 : absence de détection ;
- 1 : le capeur droit voit un obstacle (et il faut tourner à gauche) ;
- 2 : le capeur gauche voit un obstacle (et il faut tourner à droite) ;
- 3 : les deux capeurs voient un obstacle (on tournera arbitrairement à gauche) ;

Les commandes transmises par Avoid sont : **TURN_LEFT** et **TURN_RIGHT**.

Détails programmation :

La fonction *avoid()* récupère dans un tableau les coordonnées des obstacles (le nombre d'obstacles peut être variable). Les coordonnées des obstacles sont partagées dans une zone de mémoire partagée (*shared Memory)* [2].

On récupère également le rayon de sensibilité aux obstacles (par défaut, le capteur IR réagit à un obstacle lorsque le Robot est ans une case autour de l'obstacle). On fait ensuite appel à la fonction ir_detect(). Au retour de cet appel, on fixe le drapeau d'avoid ainsi que la commande d'avoid.

La fonction **ir_detect()** récupère les coordonnées $< ligne, Colonne >$ du Robot ainsi que sa *Direction*. On teste si le Robot est dans le rayon de sensibilité ; et on transmet une des valeurs 0..3 (0 = le robot n'est proche d'aucun obstacle). De même, une détection (seulement frontale) des bords de l'échiquier est opérée.

La fonction main de ce module procède (comme les autres) à s'abonner aux zones de mémoire partagées puis appelle la fonction *avoid()*. Cette dernière s'enferme dans une boucle infinie et remplit sa tâche (décrite ci-dessus).

Remarque sur le cas réel : dans un robot réel, les détections ont lieu à l'aide de vrais capteurs. La fonction *ir_detect()* transmettra au PIC chargé des évitements les valeurs 0..3 en fonction des détections réelles. On n'aura donc pas besoin de connaître l'emplacement des obstacles, ni la position du robot ni sa direction ni les coordonnées de l'échiquier. Le rayon de sensibilité est également une valeur inhérente aux capteurs IRs.

[2] Cette zone que l'on peut assimiler à une boîte aux lettres permet l'échange entre les processus qui, chacun dispose de sa propore mémoire.

IV.B Follow

Ce processus utilise deux capteurs de lumière. L'état de ces capteurs est défini sur deux canaux (0 et 1)

Dans la simulation, l'état de ces capteurs est défini en fonction des coordonnées $< ligne, Colonne >$ du Robot ainsi que l'emplacement (ligne et colonne) de la source lumineuse (une lampe placée par défaut en haut à droite de l'échiquier, mais dont la position est paramètrable à l'appel).

Selon ces coordonnées, on définit les valeurs qui permettent de savoir lequel des cellules photo-électriques a été frappée par la lumière.

Un seuil de différence est défini (pour la simulation et le cas réel) permettant de maîtriser l'écart entre les deux valeurs émises par ces capteurs. L'écart entre les deux cellules permet de connaître la direction dans laquelle les rayons lumineuses ont ils frappé les capteurs.

Si la différence entre les valeurs affichées par les deux capteurs est inférieure à ce seuil, on considèrera que l'on n'a *rien vu* permettant de se diriger vers la source de lumière.

Pour la simulation, on affecte des valeurs aux canaux en utilisant ce seuil : par exemple, pour signifier que le capteur gauche est été frappé (plus que celui de droite), on affecte au canal 0 la valeur *seuil+1* (et 0 au canal 1).

Ainsi, Selon la position et l'angle de frappe des capteurs, on décide de transmettre des valeurs (sur les deux canaux) qui permettent au Robot de s'approcher de la source lumineuse.

Les commandes transmises par Follow sont : **TURN_LEFT** et **TURN_RIGHT** [3].

Détails de programmation :

La fonction *follow()* récupère les coordonnées de la source lumineuse ainsi que le rayon de sensibilité par rapport à la source (par défaut proche de 5 cases autours mais la valeur est fonction de la taille de l'échiquier et peut être fixée à l'appel).

Puis dans l'itération, on récupère les coordonnées $< ligne, Colonne >$ du Robot ainsi que sa direction. Ces coordonnées sont disponibles dans une zone de mémoire partagée.

On fait ensuite appel à la fonction *analog()* pour intérroger l'état des canaux. Au retour de cet appel, on décide d'une commande et on partage le drapeau de Follow ainsi que sa commande.

La fonction **analog()** teste si le Robot est dans le rayon de sensibilité de la lumière ; et transmet (vu ci-dessus), en fonction de la position du Robot et de sa direction, une valeur sur les capteurs de lumière. Un écart inférieur au seuil = le robot n'est proche de la source.

Aussi, pour les besoins d'affichage, on signale lequel des deux capteurs a été frappé[4].

Remarque sur le cas réel : dans un robot réel, la détection de la lumière a lieu à l'aide des capteurs. La fonction *analog()* transmettra au PIC chargé du suivi de la lumière les valeurs des capteurs en fonction des détections réelles. Le seuil permettant de distinguer la présence d'un rayon lumineux devra être défini par échantillonage. On n'aura donc pas besoin de connaître l'emplacement de la source lumineuse, la position du robot ou sa direction (pour la détection). Le rayon de sensibilité est également une valeur inhérente aux capteurs photo-électriques.

[3] On peut aussi bien utiliser les commandes ARC_LEFT et ARC_RIGHT. Ces deux dernières font tourner le Robot de 45° alors que TURN_LEFT opère un virage de 90°

[4] Cette information est utile pour la simulation. Elle permet d'éviter de transmettre plusieurs données qui auraient été nécessaires à l'afficheur pour connaître le canal analogique touché.

IV.C Escape

Le but de ce processus est d'éviter les bords du terrain (échiquier) ainsi que les obstacles (lorsque les capteurs IRs n'ont pas pu les détecter à temps[5]).

Ce processus utilise, pour la détection de collisions, une ceinture de Bumpers (deux devant et un derrière). Ces bumpers sont activés lorsque le Robot "se cogne" contre les murs (autour) ou contre l'un des obstacles (discutés dans le processus Avoid).

Dans la simulation, l'état de ces capteurs est défini en fonction des coordonnées $< ligne, Colonne >$ du Robot, la direction du Robot, la commande actuelle sur les moteurs (nécessaire à une meilleure détection pendant la simulation) ainsi que l'emplacement des obstacles. Les murs délimitent l'échiquier.

On récupère également le rayon de sensibilité des obstacles (par défaut = 1 case autour). On utilise ces valeurs pour détecter une collision entre le Robot, les obstacles ou les bords de l'échiquier.

Selon ces données, les commandes transmises par Escape sont : **TURN_LEFT**, **TURN_RIGHT** et **BACKWARD**.

Détails de programmation :

La fonction *escape()* récupère dans un tableau les coordonnées des obstacles (le nombre d'obstacles peut être variable). Les coordonnées des obstacles sont partagées dans une zone de mémoire partagée et seront utilisés par *bump_check()*, chargée de la détections effective.

les Bumpers sont touchés par les obstacles ou les bords de l'échiquier, en fonction de la position, la direction et la commande actuelle du Robot[6].

Au retour de l'appel à *bump_check()*, on teste l'état des Bumpers et on émet une des commandes suivantes :

Bumpers Gauche et droit touchés : émettre la commande BACKWARD pendant environ 0.5 secondes puis TURN_LEFT ;

Bumper Gauche (seul) touché : émettre la commande TURN_RIGHT ;

Bumper Droit (seul) touché : émettre la commande TURN_LEFT ;

Bumper arrière touché : émettre la commande TURN_LEFT (décision arbitraire, on peut aussi bien tourner à droite).

La fonction *bump_check()* récupère les coordonnées $< ligne, Colonne >$ du Robot et sa direction. Pour les besoins de la simulation, on récupère également la commande actuelle du Robot (émise par l'Arbitre) pour déterminer quels Bumpers seront touchés. Cette commande sera inutile dans le cas du Robot réel.

Selon l'ensemble de ces données, on détermine quels Bumpers sont touchés par les bords du terrain ou par les obstacles.

N.B. : on tient compte des obstacles car les capteurs IRs peuvent ne pas les détecter dans le rayon de proximité défini (cf. Avoid).

La fonction *bump_check()* transmet donc l'état des 3 Bumpers LEFT, RIGHT ou BACK.

La fonction main de ce module procède (comme les autres) à s'abonner aux zones de mémoire partagées puis appelle la fonction escape().

[5] selon leur qualité, les capteurs IRs ont des ratés !

[6] La commande est nécessaire pour la simulation. Par exemple, une direction=Nord et des coordonnées à la limite du terrain devraient permettre de détecter une collision sauf si la commande actuelle = BACKWARD : le Robot recule et donc les Bumpers ne seront pas touchés

Remarque sur le cas réel : dans un robot réel, les Bumpers seront touchés en présence réelle d'un obstacle ou d'un mur (bord). Il suffira donc de connaitre l'état de chaque Bumper réel pour décider de la commande. L'ensemble des coordonnées utilisé pour la simulation sera donc inutile au vrai Robot.

IV.D Le Driver Moteur (simulateur)

Pour les besoins de la simulation, un processus *Driver-Moteur* est défini. Celui-ci sera remplacé par le Servo-moteur du Robot Réel.

Le rôle de ce processus est de définir la prochaine position du Robot en fonction de la position actuelle, le direction et la commande courante.

La fonction *driver_moteur()* commence par récupérer les données partagées (position du Robot, direction et la commande).

Ensuite, dans une boucle infinie, un appel à la fonction *calculer_new_position_et_dir_robot*() permet de calculer les valeurs $< new\_ligne, new\_colonne >, new\_direction$ à l'aide de la commande actuelle (et l'ancienne position).

Au retour de cet appel, la nouvelle position et la nouvelle direction sont partagées (en particulier pour l'affichage).

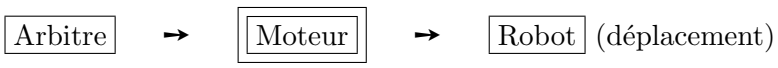
La fonction **calculer_new_position_et_dir_robot()** permet de calculer, sur l'échiquier (où les déplacement se font case par case) la nouvelle position et la nouvelle direction du Robot en fonction de l'actuelle position, direction et la commande.

L'utilité de la commande actuelle est évidente : si celle-ci est FORWARD, la nouvelle position sera totalement différente du cas de la commande BACKWARD.

La Direction actuelle a également son importance. Elle peut être modifié dans ce processus : par exemple, une commande telle que TURN_LEFT modifie totalement la direction.

N.B. : on veille à conserver la Robot sur l'échiquier. Si la nouvelle ligne (ou la colonne) devait être inférieure à 1 ou plus grande que la Dimension de l'échiquier, elle sera corrigée et placée sur la valeur la plus proche, à l'intérieur de l'échiquier.

D'une manière général, ce processus imite le moteur du Robot. Les données modifiées sont celles qui le seront dans le cas du Robot réel.



Remarque : une possibilité (non implantée) est de différencier les 2 servomoteurs et intervenir individuellement sur chaque roue motrice ; ce qui permettrait de réaliser les commandes telles que TURN_LEFT.

Remarque sur le cas réel : dans un robot réel, les déplacements ont effectivement lieu par les roues. Il n'y a donc pas de calcul de la nouvelle position : celle-ci sera *de facto* la nouvelle position du Robot physique.

Par contre, il sera nécessaire de connaître à tout moment la position du Robot. Ces coordonnées sont en général calculés par des sonares et autres dispositifs de localisation tels qu'un télémètre Infrarouge (avec éventuellement une triangulation).

IV.E Cruise

Ce processus se contente d'émettre la commande FORWARD en permanence.

Cruise → Arbitre (déplacement)

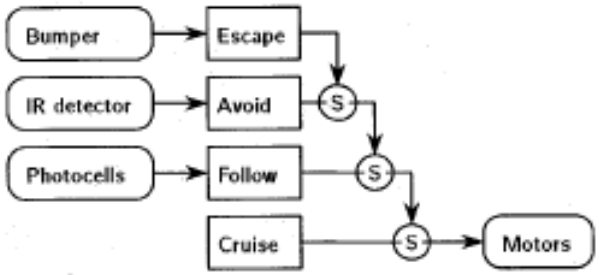
Remarque sur le cas réel : le même code sera implanté dans un robot réel. On peut également éviter la création de ce processus en initialisant par défaut à FORWARD la commande émise par l'arbitre (voir ci-dessous).

IV.F L'arbitre

Le rôle de l'arbitre est de récupérer toutes les commandes venant des cartes (les autres processus) et d'en choisir une en fonction de la priorité prévue lors de la conception.

L'arbitre consulte l'ensemble de ces commandes et met en place les noeuds S (suppress) de l'architecture.

Comme on peut le remarquer dans la figure de l'architecture du Robot R3, c'est la commande du module Escape qui l'emporte.



La prise en compte des noeuds S est simplement réalisée par une batterie de tests de la forme (il n'y a pas de 'sinon')

```
Si drapeau de CRUISE est levé, la commande finale = la commande de CRUISE
Si drapeau de FOLLOW est levé, la commande finale = la commande de FOLLOW
Si drapeau de AVOID est levé, la commande finale = la commande de AVOID
Si drapeau de ESCAPE est levé, la commande finale = la commande de ESCAPE
```

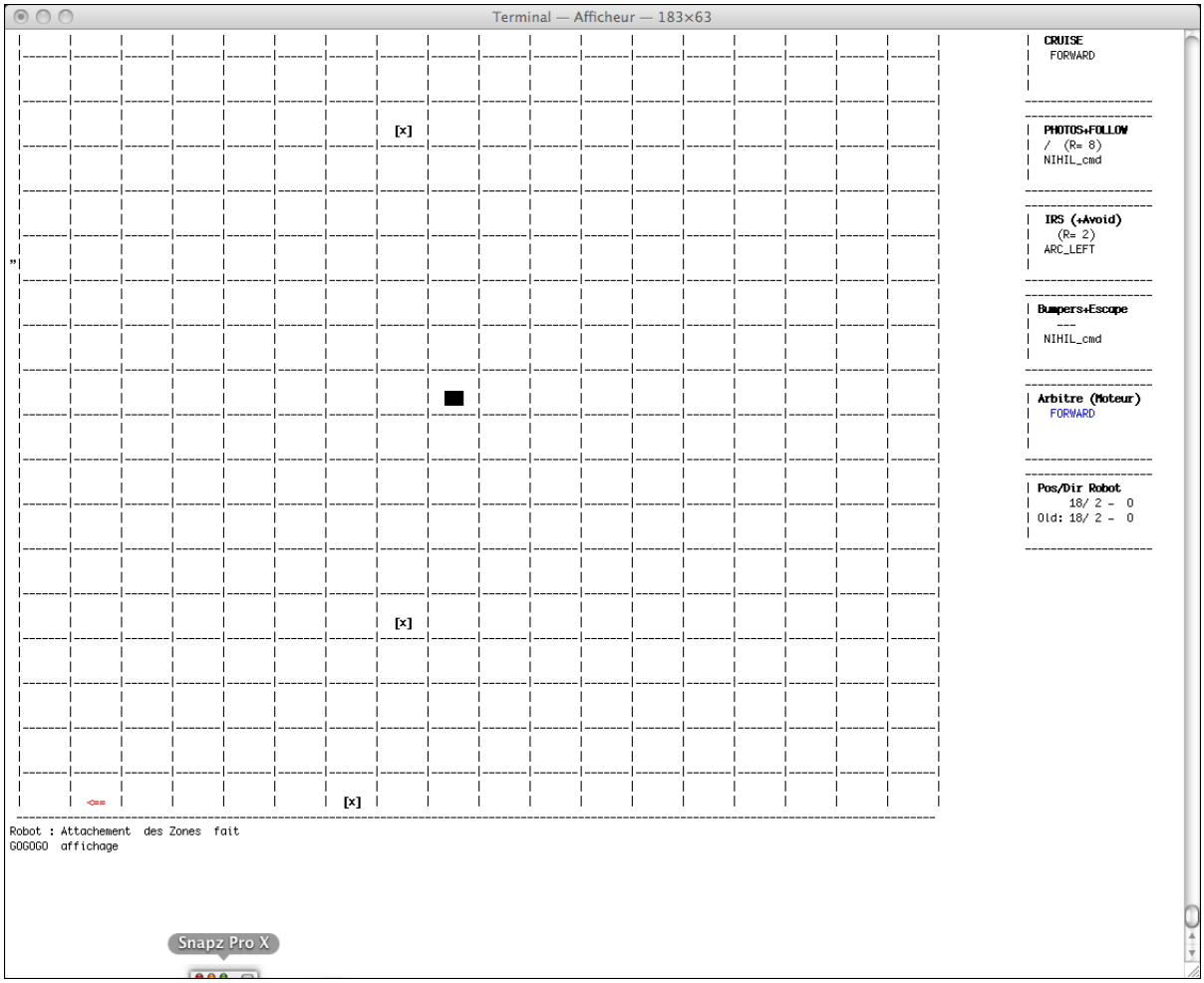
Du fait de cette séquence, la commande finale sera celle de la carte Escape qui vient en dernier dont le drapeau est levé (la commande finale = celle émise au moteur).

Remarque sur le cas réel : dans un robot réel, le même code de l'arbitre s'applique. Bien entendu, les zones consultées par l'arbitre pour connaître les commandes des cartes indépendantes seront celles (par exemple le bus I2C) connectées à la carte maîtresse.

... → Arbitre → Moteur (déplacement)

IV.G Affichage

Ce processus indépendant est chargé d'afficher l'ensemble des informations concernant le Robot. Ce processus ne modife aucune coordonnée, ni commande ni direction.



Dans la capture d'écarn ci-dessus, l'échiquier de taille 18x18 contient 3 onstacles (représentés par **[X]**), une source lumineuse au centre représnetée par un X clignotant et en vidéo inversée ainsi que les paneaux à droite qui signalent l'état de chacun des processus (simulation des cartes).

Dans la figure, le Robot en déplacement est en bas à gauche, signalé par <== rouge dans cet exemple. Le dessin simple qui représente le Robot prend différentes formes en fonction de la direction du Robot. Ici, la direction est Oest.

Remarque sur le cas réel : dans un robot réel, un dispositif d'affichage (LCD) peut être prévu. Ici, le processus d'affichage est avant tout prévu pour les besoins de la simulation. Si le Robot réel dispose d'un écran d'affichage de taille intéressante (par exemple, 15 pouces), on peut utiliser le code ci-dessus pour les besoin d'affichagee (*tracking* en temps réel) du Robot réel.

A noter que les affichages sont faits en utilisant les code d'échappement ANSI (voir le fichier mon_screen.h). Il n'est pas raisonnable d'envisager un affichage graphique (type X11) pour un dispositif embarqué comme un Robot autonome sans rencontrer des problème d'alimentation (batterie).

IV.H Quelques remarques

- Utilité et la place des sémaphores : voir enseignant.
- D'une manière générale, les variables sont globales (pour l'efficacité).
- Algorithmes de déplacements (chemin) : A*, MCl, centres d'attraction / de répulsion, numérotation inverse, ...
- Sonar et positionnement du Robot, Télémètres : voir enseignant.
- Auttes projets Robots : voir enseignant.