Projet AMSE.

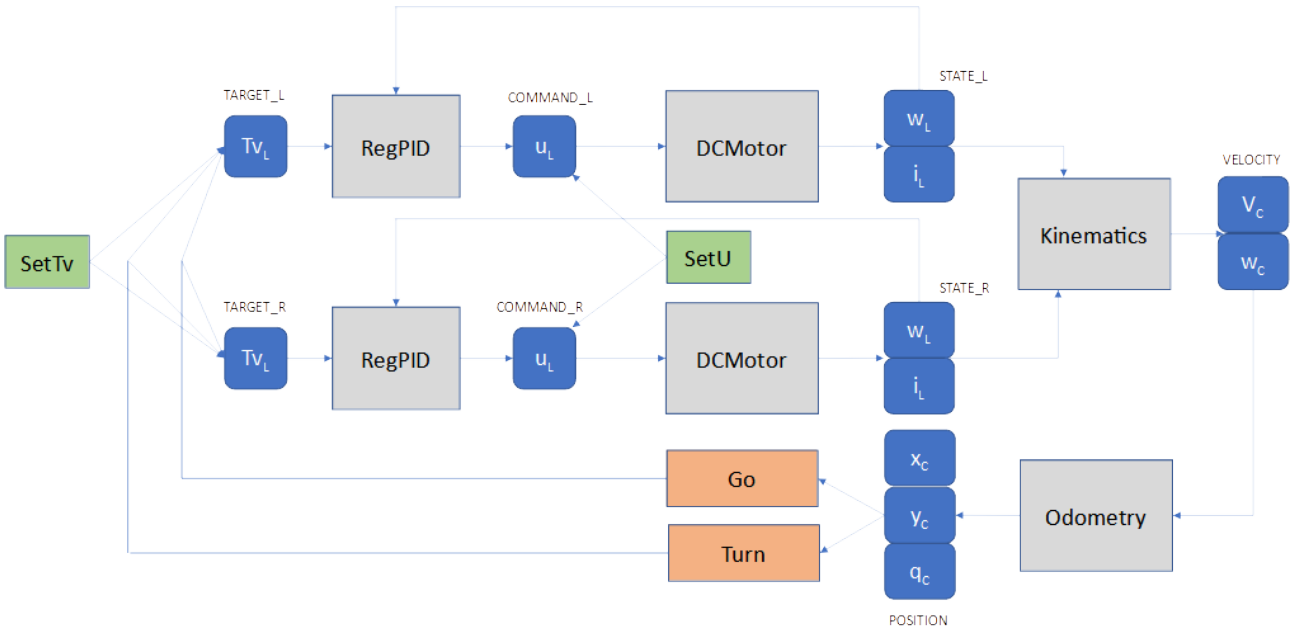
On se propose de simuler à l'aide d'une application multiprocessus le fonctionnement d'un robot mobile de type « *turttlebot* », ceci en temps-réel. A cet effet, la structure de contrôle du robot repose sur l'asservissement en vitesse de deux axes (axe **gauche**, « **LEFT** » et **droit**, « **RIGHT** »). Chaque axe se voit associer des processus exécutant des instances de programmes identiques, à savoir :

- ①un processus exploitant l'image **`DCMotor`**, correspondant au fonctionnement d'un moteur à courant commandé par l'induit. Celui-ci est alimenté par la tension `uL` (pour le moteur gauche) ou `uR` (pour le moteur droit). Le processus crée deux grandeurs de sortie, correspondant à la vitesse angulaire du rotor et au courant d'induit, soient le couple (`wL`, `iL`) pour le moteur gauche et (`wR`,`iR`) pour le moteur droit.
- ②un processus exploitant l'image **`RegPID`**, mettant en place une régulation de type « ***P**roportionnelle*, ***I**ntégrale*, ***D**érivée* » de la vitesse du moteur à courant continu auquel il est associé. Le régulateur de la roue gauche, à partir d'une consigne de vitesse `TvL`, va générer la commande `uL`, tandis que le régulateur de la roue droite va générer la commande `uR` à partir d'une consigne `TvR`.
- ③un processus utilisant l'image **`Kinematics`**, dont le rôle est de combiner les vitesses angulaires `wR` et `wR` pour générer la vitesse linéaire `vC` et la vitesse angulaire `wC` du robot mobile proprement dit.
- ④un processus faisant tourner l'image **`Odometry`**, afin de produire la position (abscisse et ordonnée dans le plan) ainsi que l'orientation du robot (par rapport à l'axe vertical).

A ces différents programmes, qui constituent « l'ossature » de la simulation, il faut rajouter a minima quelques utilitaires supplémentaires afin de faciliter la mise en point, à savoir :

- ⑤**`SetU`** : permet d'imposer la commande au moteur droit ou gauche (à utiliser quand les régulateurs PID sont inhibés, sinon cette commande « imposée » sera écrasée par la sortie du PID à la période d'échantillonnage suivante…).
- ⑥**`SetTv`** : destiné à imposer la vitesse du rotor du moteur gauche ou du moteur droit.
- ⑦**`Go`** : commande permettant de faire avancer le robot en ligne droite à une distance donnée.
- ⑧**`Turn`** : commande déclenchant la rotation sur place du robot d'un angle déterminé.

L'ensemble des processus impliqués dans cette simulation échangera des données par l'intermédiaire de « *zones partagée* ». Il seront pilotés par l'intermédiaire de *signaux*. L'utilisation des « *alarmes cycliques* » permettra de mettre en place le mécanisme d'échantillonnage. La structure globale de cette application multiprocessus est résumée sur la figure ci-dessous :



Dans ce qui suit, on décrit avec davantage de détails les zones de mémoire partagées à gérer ainsi que les différents programmes à développer.

Zones de mémoire partagée utilisées par l'application :

`TARGET_L et TARGET_R.`
Elles correspondent respectivement aux vitesses des roues gauche et droite du robot mobile. Elle contiennent toutes deux une valeur de type « `double` » (notées `TvL` et `TvR` sur le schéma).
C'est sur ces valeurs que vont agir les processus exécutant les images `Go` (pour faire avancer ou reculer le robot) et `Turn` (pour le faire tourner sur place). On doit aussi pouvoir les imposer par l'entremise de la commande `SetTv`.

`COMMAND_L et COMMAND_R.`
Il s'agit des commandes appliquées aux moteurs gauche et droit respectivement. Les valeurs qui y sont stockées sont de type « `double` ». Normalement, les valeurs de ces commandes sont générées par les processus exécutant `RegPID`, mais elles peuvent aussi être imposées par `SetU`.

`STATE_L et STATE_R.`
Ces deux zones contiennent « l'état » (au sens de l'Automatique) des moteurs gauche et droit. Chacune regroupe deux valeurs « `double` », à savoir une pour la vitesse angulaire (`wL` et `wR`) et l'autre pour le courant moteur (`iL` et `iR`).

`VELOCITY.`
Contient deux valeurs double, représentant la vitesse linéaire (`vC`) et angulaire (`wC`) du robot.

`POSITION.`
Stocke la « position » du robot (plus exactement sa configuration, correspondant à la concaténation de l'abscisse `xC`, de l'ordonnée `yC` et de l'orientation `qC` du robot).

Programmes de base à développer.

DCMoteur.

Il s'agit du programme simulant le fonctionnement d'un moteur à courant continu. La valeur de la tension de commande est issue d'une zone partagée (`COMMAND_L` ou `COMMAND_R` selon qu'il s'agisse du moteur gauche ou du moteur droit). Le rôle de `DCMoteur` est de mettre à jour « l'état » de l'actionneur (stocké dans la zone `STATE_L` ou `STATE_R` suivant le cas) au rythme d'une alarme cyclique.
Au démarrage du processus exploitant ce programme, il faudra pouvoir indiquer dans la ligne de commande :

- **les paramètres « physiques »** du moteur, soit dans l'ordre sa résistance d'induit R, l'inductance L, la constante électrique K_e, la constante moteur K_m, le coefficient de frottement f et l'inertie totale ramenée sur le rotor J.
- La valeur T_e de la **période d'échantillonnage** que doit « cadencer » l'alarme cyclique, exprimée en s.
- Le moteur à prendre en charge, indiqué par le caractère `L` (pour le gauche) ou `R` (pour le droit).

Le lancement du processus pourrait par exemple ressembler à ceci :
`$ DCMotor 1.8 0.02 0.004 0.02 3.2e-5 6.5e-6 0.01 L`
pour le lancement du processus avec $R=1{,}8$ $L=0{,}02$ $K_e=0{,}004$ $K_m=0{,}02$ $f=3{,}2\cdot10^{-5}$ $J=6{,}5\cdot10^{-6}$ et $T_e=0{,}01$ (valeurs fantaisistes juste données ici pour l'illustration), gérant le moteur gauche.
Comme indiqué sur le schéma général de l'application, au démarrage ce processus doit se « lier » aux zones dont il a besoin (`COMMAND_x` et `STATE_x` où `x` vaudra `L` ou `R` suivant le cas), ou les créer si elles n'existent pas. Il doit ensuite mettre en place l'alarme cyclique à la bonne période. *A priori*, c'est au sein de la routine d'interception du signal `SIG_ALRM` que sera réalisée la mise à jour de l'état du moteur, suivant les équations rappelées ci-dessous :

$$i_{k+1}=z_0\cdot i_k-K_e\cdot b_0\cdot\omega_k+b_0\cdot u_k$$ (nouvelle valeur du courant moteur)

$$\omega_{k+1}=z_1\cdot\omega_k+b_1\cdot i_k$$ (nouvelle valeur de la vitesse angulaire)

avec $z_0=e^{-\frac{T_e\cdot R}{L}}$, $b_0=\frac{1}{R}\cdot(1-z_0)$, $z_1=e^{-\frac{T_e\cdot f}{J}}$, $b_1=\frac{K_m}{f}\cdot(1-z_1)$. Dans ces expressions, i représente le courant moteur, ω la vitesse angulaire du rotor et u la tension de commande.
En plus de son fonctionnement normal, on souhaite que le processus exécutant `DCMotor` s'arrête de manière « propre » quand il reçoit le signal `SIG_USR1`.

SetU.

Il s'agit d'un programme utilitaire destiné essentiellement à la mise au point. Comme suggéré plus haut, sont rôle est de venir écrire une valeur dans une des zones partagées associées à la commande d'un moteur. Par exemple, l'appel :
`$ SetU 0.5 R`
est supposé écrire la valeur 0,5 dans la zone associée à la commande de moteur droit (`R`).

ResetState.

Il pourra arriver qu'il soit nécessaire de remettre à zéro le contenu d'une zone partagée correspondant à l'état d'un des moteurs, avec une invocation telle que :

```
$ ResetState L
```

qui doit alors remettre à zéro le courant moteur et la vitesse angulaire du moteur gauche (`L`).

RegPID.

Il s'agit d'un processus de régulation dont le but est d'imposer à un moteur une vitesse de rotation donnée, exprimée en $rad \cdot s^{-1}$. La vitesse de rotation à atteindre est indiquée dans la zone partagée `TARGET_x` (avec `x` valant `L` ou `R` suivant qu'on gère la roue gauche ou la roue droite). Lors du lancement du processus correspondant depuis l'invite de commande, il faudra pouvoir préciser les paramètres du régulateur, c'est à dire les valeurs de l'action proportionnelle K , celle de l'action intégrale I , celle de l'action dérivée D et finalement la période de rafraîchissement de la T_0 (c'est à dire en pratique la période de l'alarme cyclique rythmant la mise à jour de la commande du moteur) . Pour mémoire, l'algorithme que met en œuvre un tel régulateur est le suivant :

- on calcule « l'**erreur courante** » e_k , qui correspond ici à la différence entre la vitesse angulaire « cible » (la consigne à atteindre, lue depuis la zone partagée `TARGET_x`) et la vitesse angulaire réelle du moteur (lue depuis la zone de mémoire partagée `STATE_x`).
- On met à jour l'**intégrale de l'erreur** suivant $E_k = E_{k-1} + T_0 \cdot e_k$. Attention, on prendra $T_0 \approx T_e$.
- On estime la dérivée de l'erreur $\Delta E_k = \frac{e_k - e_{k-1}}{T_0}$
- La commande à appliquer au moteur est alors (cas d'un PID « idéal » à structure parallèle) : $u_k = K \cdot \left(e_k + I \cdot E_k + D \cdot \Delta E_k \right)$. Il faut alors l'écrire dans la zone `COMMAND_x`.

Un exemple de lancement d'un régulateur PID pourrait alors être le suivant :

```
$ RegPID 1.0 0.25 0 0.01 L
```

qui démarre un processus de régulation en vitesse sur le moteur gauche (`L`) on prenant pour paramètre de régulation $K = 1{,}0$ $I = 0{,}25$ $D = 0$ et $T_0 = 0{,}01$

On souhaite qu'à la réception du signal `SIGUSR2`, le régulateur soit désactivé (c'est à dire qu'il n'écrive plus la valeur de la commande dans la zone partagée adéquate). S'il reçoit une nouvelle fois ce signal `SIGUSR2`, il est ré-activé (attention de bien veiller à remettre à zéro la valeur de E_k avant de « rendre la main » à ce régulateur...), est ainsi de suite.

SetTv.

Ce programme est destiné à imposer la valeur de consigne au moteur gauche ou au moteur droit. La syntaxe de l'appel sera identique à celle de `SetU`.

Kinematics.

Ce programme exploite les zones de données partagées issues des moteurs gauche et droit pour déterminer les vitesses angulaire ω_{Ck} et linéaire v_{Ck} du robot mobile suivant :

$$\begin{pmatrix} v_{Ck} \\ \omega_{Ck} \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ -\frac{1}{W} & \frac{1}{W} \end{pmatrix} \cdot \begin{pmatrix} R_0 \cdot \omega_{Lk} \\ R_0 \cdot \omega_{Rk} \end{pmatrix}$$

, où W désigne l'entraxe entre les roues et R_0 leur rayon commun. ω_{Lk} et ω_{Lk} sont les vitesses angulaires des rotors lues respectivement dans les zones `STATE_L` et `STATE_R`. Les valeurs calculées iront alimenter le contenu de la zone partagée `VELOCITY`. L'entraxe ainsi que le rayon des roues seront indiqués à `Kinematics` lors de son lancement depuis l'invite de commande, suivant (par exemple) :
`$ Kinematics 0.4 0.07 0.02` (le dernier paramètre correspondant à la période de mise à jour).

`Odometry`.
Ce programme met à jour la position et l'orientation du robot dans son plan d'évolution. Si on note x_k , y_k et θ_k respectivement son abscisse, son ordonnée et son orientation, nous aurons

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \\ \theta_{k+1} \end{pmatrix} \approx \begin{pmatrix} x_k - v_{Ck} \cdot T_1 \cdot \sin(\theta_k) \\ y_k + v_{Ck} \cdot T_1 \cdot \cos(\theta_k) \\ \theta_k + T_1 \cdot \omega_{Ck} \end{pmatrix}$$

.
Les données relatives à la position dans le plan et à l'orientation du robot, à savoir `xC`, `yC` et `qC`, seront extraites de la zone partagée nommée `POSITION`. Elle vont être mise à jour périodiquement à partir des informations issues de la zone partagée `VELOCITY`. `T1` est la période de mise à jour (elle sera choisie supérieure aux périodes de mise à jour associées aux tâches de plus bas niveau). Ce paramètre, exprimé en *s*, sera communiqué au processus au lancement en tant qu'argument de la ligne de commande.

`SetPosition.`
Il s'agit d'un utilitaire destiné à modifier ou initialiser la position et l'orientation du robot en cours de simulation. Il prend comme argument l'abscisse, l'ordonnée et l'orientation (en °) qu'on souhaite imposer au robot. Un exemple d'appel pourra ressembler à ceci :
`$ SetPosition 3.5 7.2 45`
pour positionner le robot à l'abscisse 3.5 *m*, l'ordonnée 7.2 *m* et avec une orientation de 45° par rapport à l'axe vertical.

`RobotStart.`
Il s'agit d'une des « pièces maîtresses » de la simulation car c'est lui qui sera en charge de créer / détruire les différentes zones de mémoire partagées ainsi que de lancer les différents processus impliqués dans le fonctionnement du robot (c'est à dire ceux exploitant les images de `DCMotor`, `RegPID`, `Kinematics` et `Odometry`). En cours de simulation, si le processus exécutant `RobotStart` reçoit le signal `SIGUSR1`, il devra mettre fin « proprement » à l'ensemble de l'application (ce qui inclut la destruction des différentes zones partagées).

`PositionServer.`
Le processus exécutant cette image est un serveur itératif, écoutant sur le port 7777 et qui renverra la position et l'orientation du robot sous forme de 3 `double` à réception du caractère `@`. Il devra mettre en œuvre des « *threads* » de service pour satisfaire les différents clients connectés. Le but de

cette application est de pouvoir suivre en parallèle la trajectoire de différents robot sur une machine centralisant ces dernières, par exemple, ou d'autoriser plusieurs robot à partager leurs positions.