Aversive/Asservissement Microb 2008

De Wikidroids.

Documentation de l'asservissement Microb Technology 2008, Olivier MATZ, Fabrice DESCLAUX (obstacle avoidance)

- *Cette documentation est encore en cours de rédaction, elle évolue régulièrement. N'hésitez pas à me signaler toute incohérence, ou tout point qui ne serait pas abordé avec suffisamment de clarté. Mon mail : zer0@droids-corp.org*
- *Les exemples présentés sont là dans un but didactique seulement, il est possible qu'ils ne compilent même pas (il est sinon préférable de se reporter au code de notre robot)*

Sommaire

Introduction

Principe de base de l'asservissement

Un asservissement est un système rebouclé permettant de contrôler un procédé comportant une entrée et une sortie. Le but de cette documentation est de présenter une solution d'implémentation d'un asservissement numérique modulaire en utilisant Aversive, appliqué au contrôle d'un robot pour Eurobot, mais qui pourrait être utilisé pour d'autres applications.

Cet asservissement est implémenté sur un microcontrôleur Atmel AVR, et a été utilisé en 2007 (avec quelques variantes) par les équipes Eirbot, Microb Technology, Esial robotik et Maggie. Il est relativement bien adaptable pour un autre robot et tout le code est disponible sous licence GPL.

Après avoir présenté la mécanique et l'électronique du robot, nous pourrons aborder le code d'un point de vue global. Puis à travers plusieurs exemples dont la difficulté s'accroît au fur et à mesure, nous ajouterons une à une les briques de notre asservissement.

Un peu de documentation sur l'asservissement numérique

- Asservissement de RCVA (http://www.robot-cva.com/?page=tech) (IUT de Ville d'Avray), documentation 2006, un peu court mais décrit les principes d'un des meilleurs asservissement de la coupe de robotique
- Description de l'asservissement du robot du Clubelek de l'INSA Lyon 2007 (http://clubelek.insa-lyon.fr/joomla/fr/base_de_connaissances/informatique /asservissement_et_pilotage_de_robot_autonome_introduc_2.php) , article bien écrit et bien illustré
- L'asservissement de l'UTC (http://wwwassos.utc.fr/coupem6/2007/fichiers /RapportTX.pdf) , rapport de 2006.
- Asservissement Microb Technology 2005 (http://microb.technology.free.fr /fr/asservissement2005/index.html) , l'ancêtre du document que vous lisez
- Asservissement du robot d'Eirbot (ENSEIRB) en 2003 (http://f4eru.free.fr/robot /asserv03.doc) , l'ancêtre de l'ancêtre
- L'asservissement 2007 de nos amis d'Eirbot (http://eirbot.enseirb.fr/documents /DocAsserv2k7.pdf) qui utilisent Aversive aussi.
- Un thread du forum de Planète-Sciences (http://www.planete-sciences.org/forums /viewtopic.php?t=8571&start=0&postdays=0&postorder=asc&highlight=) dans lequel de nombreux points sont discutés, notamment par Gargamel, un ancien professeur de l'IUT de VA, et expert dans le domaine.
- Asservissement sur wikipedia (http://fr.wikipedia.org/wiki/Asservissement)
- Implantation d'un correcteur numerique
- Ajouter des liens ici

Cahier des charges de notre assservissement

L'implémentation devra se conformer aux contraintes suivantes :

Multi-usage : possibilité d'asservir différents systèmes simples avec un PID, ou un autre filtre. Les fonctions d'entrées et de sortie ne sont pas forcément des codeurs incrémentaux et un PWM comme sur un robot (ex: chauffage, moteur brushless, ...)

Modulaire : permet une gestion entièrement autonome réalisée sous interruption ou une gestion manuelle dans une boucle.

Portable : sur la plupart des AVR (mais en conservant la possibilité de faire tourner le maximum de choses sur un PC).

Dynamique : on doit pouvoir 'instancier' un asservissement à tout moment, et le retirer lorsqu'on n'en a plus besoin. De plus, il doit être possible de gérer plusieurs asservissements à la fois, dans les limites de mémoire et de puissance du uC bien sûr.

Spécificités pour un robot d'Eurobot

- Asservissement de position
- Asservissement différentiel des roues, contrôlant l'angle et la distance du robot plutôt que chacune des roues indépendamment.

- Possibilité de générer des trajectoires dont la courbe de vitesse est trapézoïdales (limite de vitesse et d'accélération, plus anticipation sur le freinage pour s'arrêter à la bonne position).
- Localisation du robot sur le terrain : coordonnées x,y,a.
- Roues codeuses non motorisées en plus des roues motorisées.
- Générateur de trajectoire avec évitemment d'obstacles et points de passage.

Mécanique de notre robot

Deux roues motorisées à l'arrière, billes à l'avant

Notre robot est proche mécaniquement de la plupart de beaucoup de robots de la coupe. Chacun des 2 moteurs situés à l'arrière commande une roue.

Les roues du robot 2006 et leurs codeurs séparés

Le bloc moteur de 2006

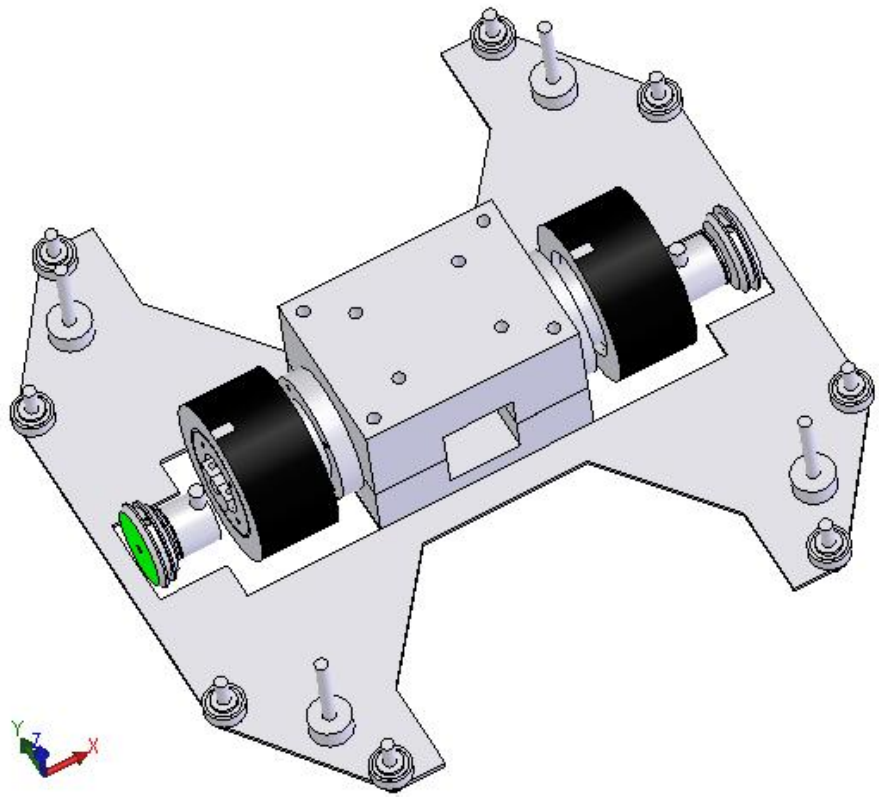
Le robot de Microb Technology 2006 vu de dessous

Précision de la mécanique

Il est nécessaire que la mécanique soit réalisée avec le plus de précision possible. En effet, un simple jeu dans la fixation des odomètres non-motorisés entraîne une dérive de la mesure de la position rendant toute localisation du robot très difficile. De même, la répartition du poids sur les différents points d'appui est très importante : le maximum de poids doit être sur les roues motorisées. Avec un code équivalent en 2006 et 2007, le comportement du robot était largement meilleur en 2007 grâce à une mécanique bien mieux réalisée.

Roues codeuses séparées

Comme indiqué plus haut, le robot dispose, en plus des roues de propulsion du robot, de roues non motorisées effectuant la mesure de la position. En effet, les roues motorisées, pour être efficaces en adhérence, doivent être plutôt larges et molles. Cela est malheureusement incompatible avec une mesure fiable de la position : diamètre de la roue et distance entre les 2 roues imprécis et tassage du pneu en virage. De plus, ces roues peuvent aussi avoir tendance à patiner lors de blocage contre un obstacle ou d'une forte accélération.



L'utilisation des roues codeuses externes permet de gagner en précision sur la mesure de position, et aussi de détecter les patinages (la valeur des codeurs moteurs est alors décorellée de celle des codeurs externes).

Il est à noter que contrairement à notre choix effectué en 2006, il semble préférable de placer les roues codeuses non motorisées à l'extérieur plutôt qu'à l'intérieur, comme nous l'avons fait en 2007.

Electronique du robot

Carte mère 2006

Carte de puissance 2006

Vous pouvez vous reporter aux pages suivantes pour plus de détails :

- Microb_Technology/2006/electronics
- Microb_Technology/2007/electronics

En résumé, le microcontrôleur central génère un PWM et un bit de sens pour chaque moteur à commander. Nous utilisons des moteurs à courant continu, commandés par des ponts en H (LMD18200T). La valeur de chaque odomètre qui arrive sous la forme de compteurs gray 2 bits (les fameux canaux A et B des roues codeuses) est décodée par des

microcontrôleurs plus petits (AT90s2313) qui la transforme en un compteur 8 bit. Tous les calculs sont effectués dans le microcontrôleur central (ATMega128), nous n'utilisons pas de composant spécialisé comme le LM629.

A noter que d'autres équipes comme Eirbot ou Esial Robotik utilisent une carte différente, mais seul la configuration et le module de lecture des codeurs doivent être changés (chez Eirbot, le décodage est effectué par un CPLD qui gère 4 codeurs).

Spécificités de notre asservissement

Asservissement en theta-alpha

Un autre choix est celui de l'asservissement en Theta-alpha. Nous l'avons expérimenté à Eirbot en 2004, puis depuis 2005 par Microb Technology et Eirbot. Elle consiste à ne pas asservir, comme généralement sur les robots de la coupe, chacune des roues avec un PID, mais plutôt d'utiliser un PID pour asservir la différence des positions de roues (l'angle Alpha), et un autre PID pour asservir la distance parcourue (Theta).

Un exemple simple d'utilisation est le calage bordure : dans ce cas, on n'asservit que la distance et non l'angle, ce qui permet au robot de ne pas opposer de résistance en angle lors de son recalage. De plus, intuitivement, onsent bien qu'asservir un angle et une distance correspond plus à asservir le robot plutôt que lorsque l'on asservit chacune des roues indépendamment. Pour couronner le tout, nous verrons que ce type d'asservissement facilitera notre travail pour la génération de trajectoires.

Aperçu du programme final

Aversive

Notre code va se baser sur le projet Aversive que je développe, un framework de compilation GPL pour microcontrôleurs AVR utilisant GCC et la libc-avr. Le but de ce projet est de fournir des bibliothèques permettant d'utiliser les spécificités des AVR de manière simple. Aversive introduit la notion de module, il s'agit d'un programme compilé sous la forme d'une bibliothèque et fournissant une fonctionnalité. Par exemple, le module PWM permet de configurer les registres internes pour générer un signal PWM et de fournir à l'utilisateur une interface plus facile à appréhender.

Le code de notre asservissement s'intègre donc à Aversive sous la forme de plusieurs modules, ayant chacun un but spécifique. On peut d'ores et déjà considérer 4 groupes fonctionnels : d'une part l'asservissement a pour objectif par exemple la régulation de vitesse d'un moteur. Deuxièmement, nous avons besoin de calculer notre position courante en x,y,a sur le terrain ; troisièmement, nous devons fournir une interface pour générer des trajectoires de haut niveau pour notre robot. Pour finir, le dernier groupe fonctionnel va être charger de surveiller le bon fonctionnement de l'ensemble et de détecter des éventuels blocages, patinages, ...

Control system

- devices/control_system/control_system_manager : C'est le module d'asservissement principal, c'est lui qui gère l'appel des autres modules.
- devices/control_system/filters/pid : Un filtre de type PID (proportionnel, dérivé, intégral).
- devices/control_system/filters/ramp : Filtre permettant de limiter la dérivée de l'entrée, non utilisé dans notre asservissement de position, mais peut l'être dans un

asservissement de vitesse.

- devices/control_system/filters/quadramp : Filtre qui permettant de limiter la dérivée et la dérivée seconde de l'entrée. C'est ce filtre qui génère les courbes de vitesse trapézoïdales.

Position

- devices/robot/position_manager : calcul de la position du robot en fonction des codeurs.
- devices/robot/robot_system : Définition d'un procédé décrivant le robot (angle et distance). Le module effectue les transformations nécessaires pour le passage du repère (roueG, roueD) vers (angle, dist), et gère également le fait que le robot possède 4 roues codeuses.

Trajectory

- devices/robot/trajectory_manager : Définition de trajectoires de haut niveau pour l'utilisateur
- devices/robot/obstacle_avoidance : Module permettant de générer des trajectoires avec points de passage et évitement d'obstacles.

Blocking Detection

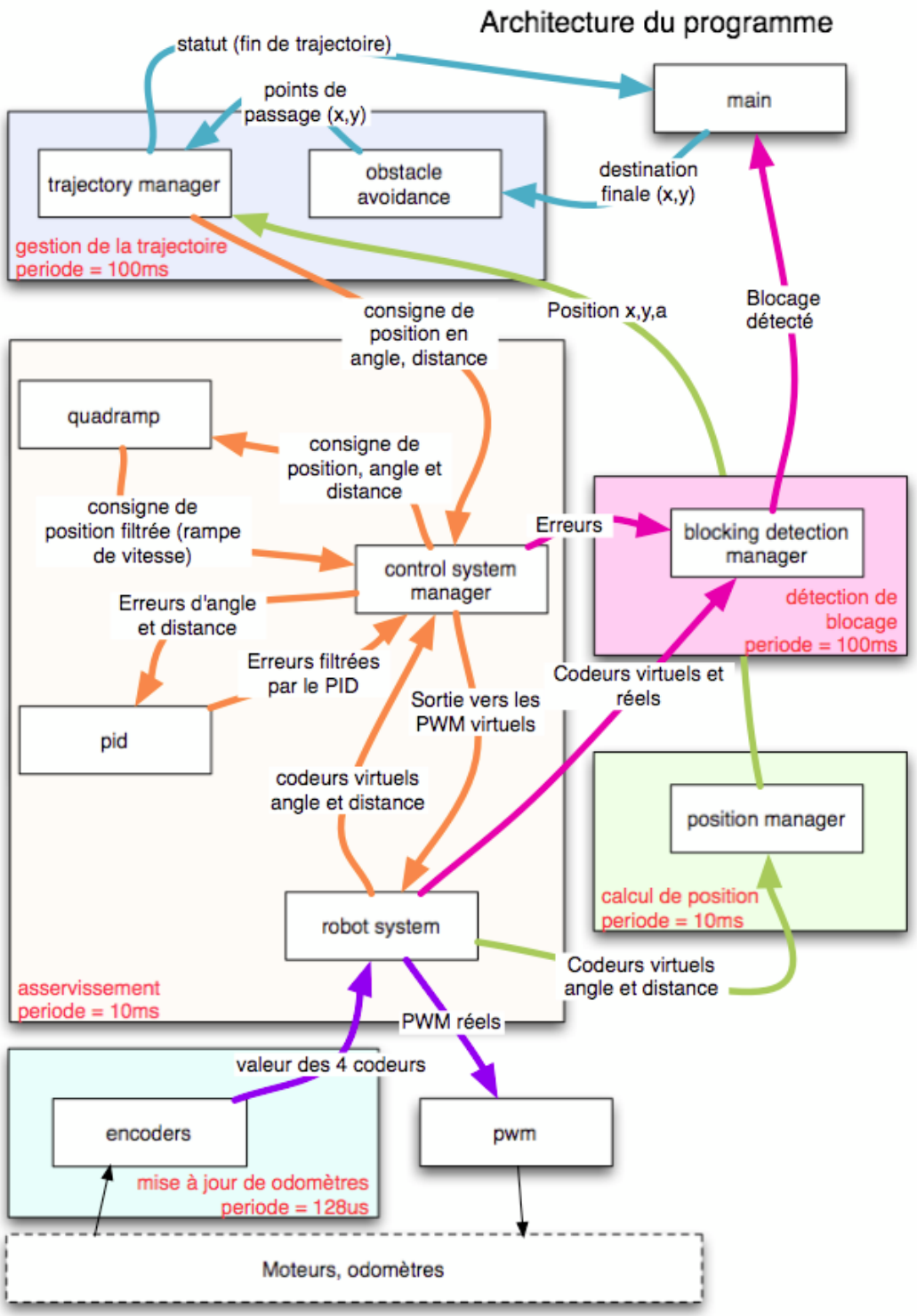
- devices/robot/blocking_detection_manager : Détection de blocage dans l'asservissement

Autres modules, non spécifiques à l'assservissement

- base/scheduler : permet de planifier l'appel de fonctions, avec gestion de priorités.
- base/math/vect2 : opérations sur les vecteurs en 2D.
- devices/encoders/encoders_microb : gestion des codeur, propre à notre hardware.
- hardware/pwm : emission de PWM pour le contrôle des moteurs.

Schéma de l'asservissement final

Au final, l'asservissement s'organise de la manière suivante :



- 0/ Le programme principal envoie une commande au module trajectory pour aller à un point (x,y,a) sur le terrain
- 1/ En fonction de la position courante, le module trajectory va donner des nouvelles consignes de position aux asservissement d'angle et de distance. Cette consigne est réactualisé périodiquement par un évènement (période de l'ordre de 100ms).
- 2/ Le module robot_system est exécuté sur interruption périodiquement(période de l'ordre de 5 à 10ms), il lit la valeur des codeurs réels et met à jour les valeurs des capteurs d'angle et de distance (virtuels).
- 3/ Pour chaque asservissement (angle et distance), le module CSM (control_system_manager) envoie la nouvelle position au filtre quadramp

- 4/ Le filtre retourne la consigne filtrée limitant la dérivée et la dérivée seconde du signal
- 5/ Puis le CSM demande la valeur des codeurs virtuels pour déterminer l'erreur d'angle et de distance
- 6/ Ces erreurs sont envoyées aux filtres PID
- 7/ Chaque filtre retourne la valeur filtrée
- 8/ Les sortie des PID (angle et distance) sont envoyées au module robot_system qui détermine les commandes PWM pour les roues
- 9/ Ces commandes sont envoyées au module PWM qui modifie les signaux générés par le uC
- 10/ Un autre évènement est exécuté de manière périodique (periode de l'ordre de 10ms), il s'agit du position_manager qui va déterminer la position du robot sur l'aire de jeu en fonction de la valeur des codeurs et de la position x,y,a précédente
- 11/ Cette position est bien entendu utile pour la prochaine exécuition du module trajectory

Où trouver notre code

Nous utilisons une branche de développement d'Aversive en avance de phase sur la branche principale. Le code source est ici: http://cvsweb.droids-corp.org/cgi-bin/viewcvs.cgi/aversive/?pathrev=b_zer0

Le code de 2006: http://cvsweb.droids-corp.org/cgi-bin/viewcvs.cgi/aversive_projects/microb2006/motherboard_atm128/

Le code (en cours d'évolution) de 2008: http://cvsweb.droids-corp.org/cgi-bin/viewcvs.cgi/aversive_projects/microb2008/main

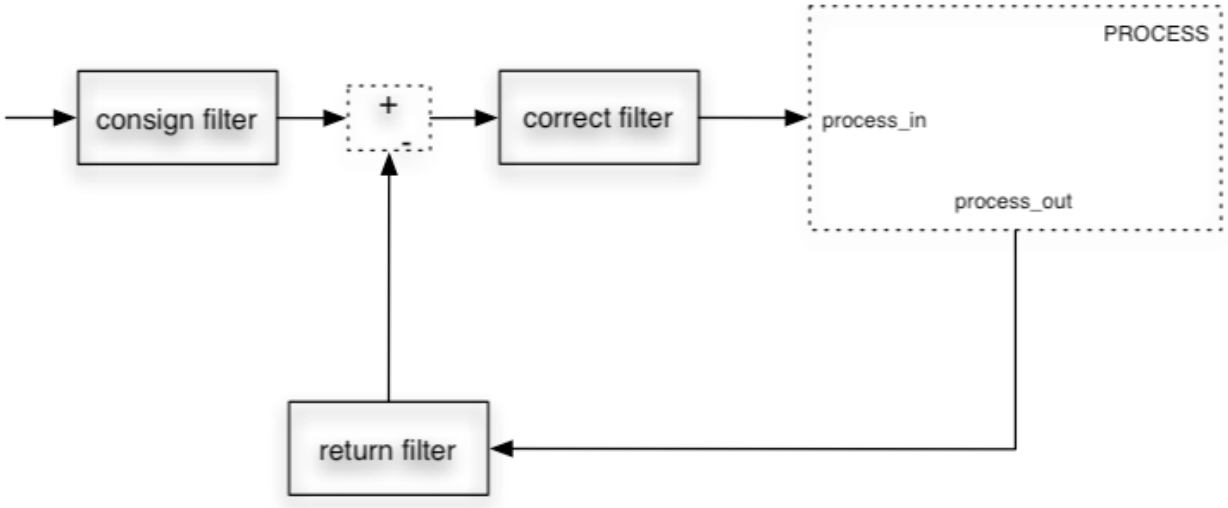
Asservissement minimaliste avec PID

Notre objectif dans cette section est de présenter les 2 modules principaux control_system et pid, qui à eux deux permettent déjà d'asservir simplement un système. Nous verrons que cette approche a beaucoup de défauts, qui seront corrigés dans les sections suivantes.

devices/control_system/control_system_manager

Description

Ce module se charge d'effectuer l'asservissement. Tout seul, il a peu d'utilité car il nécessite d'autres modules qui fournissent les différents filtres nécessaires à son fonctionnement, par exemple, le module pid en tant que le filtre de correction. La structure de donnée cs ne contient quasiment que des pointeurs de fonctions et des pointeurs vers des métadonnées propres aux filtres.



Structures

```
struct  cs {
        int32_t (*consign_filter)(void *, int32_t);
        void * consign_filter_params;

        int32_t (*correct_filter)(void *, int32_t);
        void * correct_filter_params;

        int32_t (*feedback_filter)(void *, int32_t);
        void * feedback_filter_params;

        int32_t (*process_out)(void *);
        void * process_out_params;

        void (*process_in)(void *, int32_t);
        void * process_in_params;

        int32_t consign_value;
        int32_t filtered_consign_value;
        int32_t error_value;
        int32_t out_value;
}
```

- Chaque filtre est défini par un pointeur vers la fonction à exécuter et par un pointeur vers les données du filtre.
- consign_value contient la consigne courante.
- filter_consign_value contient la consigne courante après application du filtre de consigne.
- error_value contient l'erreur courante, c'est à dire la différence entre la consigne filtrée et la sortie du procedé à asservir (résultat de la fonction process_out()).
- out_value contient la sortie courante, c'est à dire l'erreur après passage dans le filtre de correction. C'est cette valeur qui est envoyée à l'entrée du procédé par le biais de process_in().

Interface

`void cs_init(struct cs * cs)`

Initialise la structure cs : tous les champs sont mis à NULL. C'est important de ne pas oublier cette étape, car quand on joue avec les pointeurs de fonctions, il ne faut pas qu'ils pointent n'importe où.

On considère qu'une fonction de filtrage est inactive lorsque le pointeur vers celle ci

est égal à NULL.

```
void cs_set_consign_filter(struct cs * cs, int32_t (*consign_filter)(void *, int32_t), void *
consign_filter_params)

void cs_set_correct_filter(struct cs * cs, int32_t (*correct_filter)(void *, int32_t), void *
correct_filter_params)

void cs_set_feedback_filter(struct cs * cs, int32_t (*feedback_filter)(void *, int32_t), void *
feedback_filter_params)

void cs_set_process_in(struct cs * cs, void (*process_in)(void *, int32_t), void * process_in_params)

void cs_set_process_out(struct cs * cs, int32_t (*process_out)(void *), void * process_out_params)
```

Fonctions d'initialisation des filtres et fonctions d'entrée/sortie de l'asservissement.

```
int32_t cs_do_process(struct cs * cs, int32_t consign)
```

L'appel de la fonction int32_t cs_do_process(struct cs * cs, int32_t consign) effectue les opérations suivantes, dans l'ordre :

- sauvegarde de la consigne dans la structure
- application du filtre a la consigne s'il est défini
- lit la sortie du process (vaut 0 si la sortie du process n'est pas définie -> boucle ouverte)
- filtre avec feedback_filter si défini
- soustraction de la consigne filtrée avec la sortie du process filtrée
- sauvegarde dans error_value
- application du filtre de correction s'il est défini
- sauvegarde la valeur de sortie
- envoie de la valeur de sortie vers process_in() si défini.
- retourne la valeur de sortie (la même que celle envoyée au process)

```
void cs_manage(struct cs * cs)
```

Cette fonction ne fait qu'appeler la fonction cs_do_process() avec la consigne préalablement stockée dans la structure (par un cs_set_consign(struct cs * cs, int32_t v) par exemple.

```
int32_t cs_get_out(struct cs * cs)
```

Retourne la dernière sortie envoyée au procédé

```
int32_t cs_get_error(struct cs * cs)
```

Retourne la dernière erreur calculée

```
int32_t cs_get_consign(struct cs * cs)
```

Retourne la consigne courante

```
int32_t cs_get_filtered_consign(struct cs * cs)
```

Retourne la consigne courante filtrée

```
void cs_set_consign(struct cs * cs, int32_t v)
```

Modifie la consigne sans appliquer le calcul de l'asservissement, contrairement a `cs_do_process()`.

devices/control_system/filters/pid

Description

Ce module est un filtre, c'est à dire qu'il implémente l'interface définie dans la doc des filtres.

Il permet d'appliquer un PID, c'est à dire la somme pondérée de l'intégrale, de la dérivée et de l'entrée elle-même. Notre filtre possède quelques extensions par rapport à un PID classique, car il permet aussi de saturer certains signaux (entrée, terme intégral, sortie).

Structures

```
struct pid_filter
{
   int16_t gain_P; /**< Gain of Proportionnal module */
   int16_t gain_I; /**< Gain of Integral module */
   int16_t gain_D; /**< Gain of Derivate module */

   uint8_t out_shift;

   int32_t max_in; /**<  In saturation levels */
   int32_t max_I; /**<   Integral saturation levels */
   int32_t max_out; /**< Out saturation levels */

   int32_t integral; /**< previous integral parameter */
   int32_t prev_in;  /**< previous in */
   int32_t prev_D;   /**< previous derivate parameter */
   int32_t prev_out; /**< previous out command (for debug only) */
};
```

Interface

`void pid_init(struct pid_filter * p);`

> Initialise les valeurs de la structure : gain à 1 pour le proportionnel, 0 pour les autres. Les saturations sont désactivées, et le diviseur global est à 1. Les champs integral et last_in sont mis à 0. Le filtre se comporte donc comme le filtre identité (aussi appelé filtre-qui-sert-à-rien) lorsque qu'il vient d'être initialisé.

`int32_t pid_do_filter(struct pid_filter * p, int32_t in);`

> Applique le filtre sur la valeur in et retourne le résultat.

`void pid_set_gains(struct pid_filter * p, int16_t gp, int16_t gi, int16_t gd);`

`void pid_set_maximums(struct pid_filter * p, int32_t max_in, int32_t max_I, int32_t max_out);`

`void pid_set_out_shift(struct pid_filter * p, int16_t out_shift);`

> Les accesseurs qui vont bien pour configurer le filtre.

`int16_t pid_get_gain_P(struct pid_filter *p);`

`int16_t pid_get_gain_I(struct pid_filter *p);`

`int16_t pid_get_gain_D(struct pid_filter *p);`

`int16_t pid_get_max_in(struct pid_filter *p);`

`int16_t pid_get_max_I(struct pid_filter *p);`

```
int16_t pid_get_max_out(struct pid_filter *p);

uint8_t pid_get_out_shift(struct pid_filter *p);

int32_t pid_get_value_I(struct pid_filter *p);

int32_t pid_get_value_in(struct pid_filter *p);

int32_t pid_get_value_D(struct pid_filter *p);

int32_t pid_get_value_out(struct pid_filter *p);
```

Les accesseurs en lecture.

Exemple

Cet exemple d'une vingtaine de lignes est là pour bien comprendre le fonctionnement des modules décrit. L'idée est d'asservir une roue en vitesse avec un PID. Il n'est bien sûr pas souhaitable d'implémenter un asservissement comme ceci, nous verrons des exemples plus évolués dans la suite de ce document.

```
struct cs cs;
struct pid_filter p;

/* fonctions à définir ailleurs dans votre programme */
extern void motor_set(void *, int32_t); /* applique une tension au moteur */
extern int32_t motor_get_speed(void *); /* retourne la vitesse du moteur */

int main(void)
{
  pid_init(&p);
  /* configuration du PID, dépend du moteur */
  pid_set_gains(&p, ...);
  pid_set_maximums(&p, ...);
  pid_set_out_shift(&p, ...);

  cs_init(&r);
  cs_set_correct_filter(&cs, pid_do_filter, &p);
  cs_set_process_in(&cs, motor_set, NULL);
  cs_set_process_out(&cs, motor_get_speed, NULL);

  while(true) {
        wait_ms(10);
        cs_manage(&cs);
  }
}
```

Rendre l'asservissement autonome (sous interruption)

L'étape suivante consiste à faire tourner l'asservissement en tâche de fond. En effet, le microcontrôleur aura sûrement d'autres choses à faire que de faire des boucles d'attentes actives. Pour cela, on utilise le module scheduler.

base/scheduler

Description

Gestionnaire d'appel de fonction sous interruption.

L'idée est de pouvoir planifier l'appel de fonctions de manière périodique ou unique. Il permet en outre de gérer la priorité d'évenements et empêche l'appel récursif d'une même fonction.

Son rôle dans l'asservissement est simple, il permet de le rendre autonome, c'est à dire qu'il sera exécuté sous interruption, laissant la main au programme principal lorsque le calcul est terminé.

Interface

```
void scheduler_init(void);
```

Initialisation du module

```
int8_t scheduler_add_single_event_priority(void (*f)(void *), void * data, uint16_t period, uint8_t priority);

int8_t scheduler_add_periodical_event_priority(void (*f)(void *), void * data, uint16_t period, uint8_t priority);

int8_t scheduler_add_single_event(void (*f)(void *), void * data, uint16_t period);

int8_t scheduler_add_periodical_event(void (*f)(void *), void * data, uint16_t period);
```

Ajoute un évènement à la table. Il peut être périodique ou unique. La priorité peut être précisée, entre 0 et 255 (la plus forte valeur correspond à la priorité la plus haute). La fonction planifiée sera appelée toutes les *period* avec *data* en paramètre.

Chacune des fonctions d'ajout d'évènement retourne un identifiant unique de l'évènement ajouté.

```
int8_t scheduler_del_event(int8_t num);
```

Supprime l'élèment identifié par *num*.

Exemple

```
#define PERIOD 10000/SCHEDULER_UNIT /* 10 ms */
#define PRIORITY 100

struct cs cs;
struct pid_filter p;

/* fonctions définies ailleurs */
extern void motor_set(void *, int32_t);
extern int32_t motor_get_speed(void *);

int main(void)
{
  pid_init(&p);
  pid_set_gains(&p, ...);
  pid_set_maximums(&p, ...);
  pid_set_out_shift(&p, ...);

  cs_init(&cs);
  cs_set_correct_filter(&cs, pid_do_filter, &p);
  cs_set_process_in(&cs, motor_set, NULL);
  cs_set_process_out(&cs, motor_get_speed, NULL);

  scheduler_add_periodical_event_priority(cs_manage, (void *)&cs, PERIOD, PRIORITY);
  sei();
  while(1);
}
```

Utilisation d'une rampe de vitesse sur un asservissement de vitesse

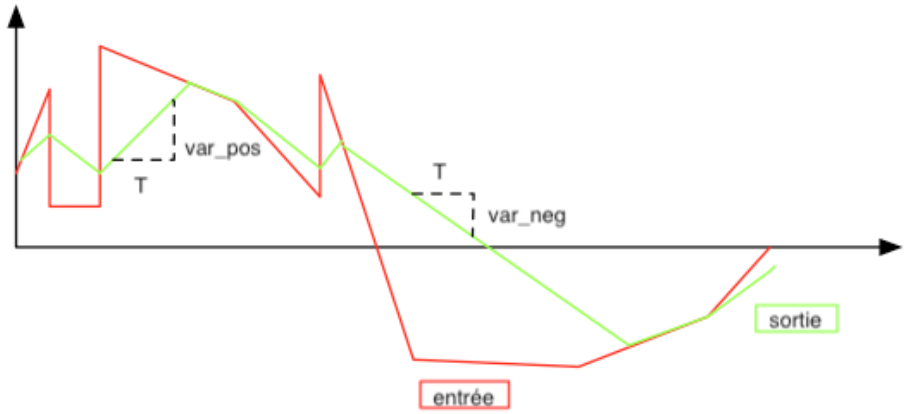
Maintenant que nous avons vu deux utilisations de base de notre module d'asservissement, nous allons tenter de le faire évoluer en appliquant un filtre en entrée de celui-ci (sur la consigne). Notre premier exemple est un asservissement de vitesse. Si nous nous amusons à changer brusquement la consigne de vitesse, alors il y a fort à parier que le PWM maximum sera envoyé jusqu'à atteindre la vitesse désirée (avec un léger dépassement ou non, selon les reglages du PID).

Comme il est physiquement impossible de modifier instantanément la vitesse d'un moteur, cela pose un problème. En effet, lorsque le PWM est maxium, l'accélération est fixée par les caractéristiques du moteur. C'est tolérable s'il n'y a qu'un moteur, mais s'il y en a 2 (un pour la roue gauche et l'autre pour la droite), et qu'ils n'ont pas exactement la même réponse, alors le robot n'ira pas droit pendant la phase d'accélération. La solution à ce problème est de limiter les variations de consignes, et donc l'accélération.

control_system/filters/ramp

Description

Le filtre impose une contrainte sur son entrée. Ce module va donc être chargé de limiter la variation de consigne (sa dérivée).



Structures

```
struct ramp_filter {
        uint32_t var_neg;
        uint32_t var_pos;
        int32_t prev_out;
}
```

var_neg et var_pos sont respectivement les variations maximales négatives et positives (en valeur absolue).

Interface

`void ramp_init(struct ramp_filter * r);`

Initialise les valeurs de la structure : var_pos et var_min à 0xFFFFFFFF. Dans ce cas, le filtre agit comme un filtre qui recopie l'entrée.

`int32_t ramp_do_filter(struct ramp_filter * r, int32_t in);`

Applique le filtre sur la valeur in et retourne le résultat.

```
void ramp_set_vars(struct ramp_filter * r, uint32_t neg, uint32_t pos);
```

Les accesseurs qui vont bien pour configurer le filtre.

Exemple

```
struct cs cs;
struct pid_filter p;
struct ramp_filter ra;

#define PERIOD 10000/SCHEDULER_UNIT /* 10 ms */
#define PRIORITY 100

/* fonctions définies ailleurs */
extern void motor_set(void *, int32_t);
extern int32_t motor_get_speed(void *);

int main(void)
{
  pid_init(&p);
  pid_set_gains(&p, ...);
  pid_set_maximums(&p, ...);
  pid_set_out_shift(&p, ...);

  ramp_init(&ra);
  ramp_set_vars(&ra, 2, 1);

  cs_init(&cs);
  cs_set_consign_filter(&ra, ramp_do_filter, &ra);
  cs_set_correct_filter(&cs, pid_do_filter, &p);
  cs_set_process_in(&cs, motor_set, NULL);
  cs_set_process_out(&cs, motor_get_speed, NULL);
  cs_set_consign(&cs, 0);

  scheduler_add_periodical_event_priority(cs_manage, (void *)&cs, PERIOD, PRIORITY);
  sei();

  wait_ms(1000);
  cs_set_consign(10);

  while(1);
}
```

A chaque fois qu'on appellera cs_manage(&cs), la consigne sera filtrée par rapport à l'entrée définie par cs_set_consign().

Pour l'exemple ci-dessus, les consignes après filtrage vaudront :

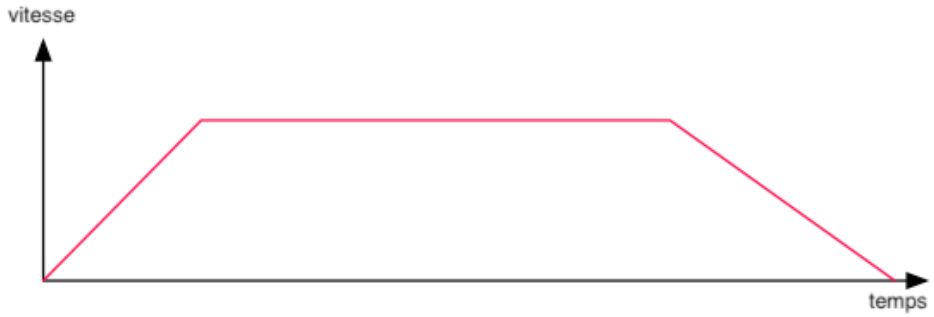
```
0, 0, 0, ... pdt une seconde ... 0, 2, 4, 6, 8, 10, 10, 10, ...
```

Si plus tard dans le programme, on appellait cs_set_consign(&r, -2), les consignes réelles redescendraient linéairement avec une pente de -1 :

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -1, -2, -2, -2, ...
```

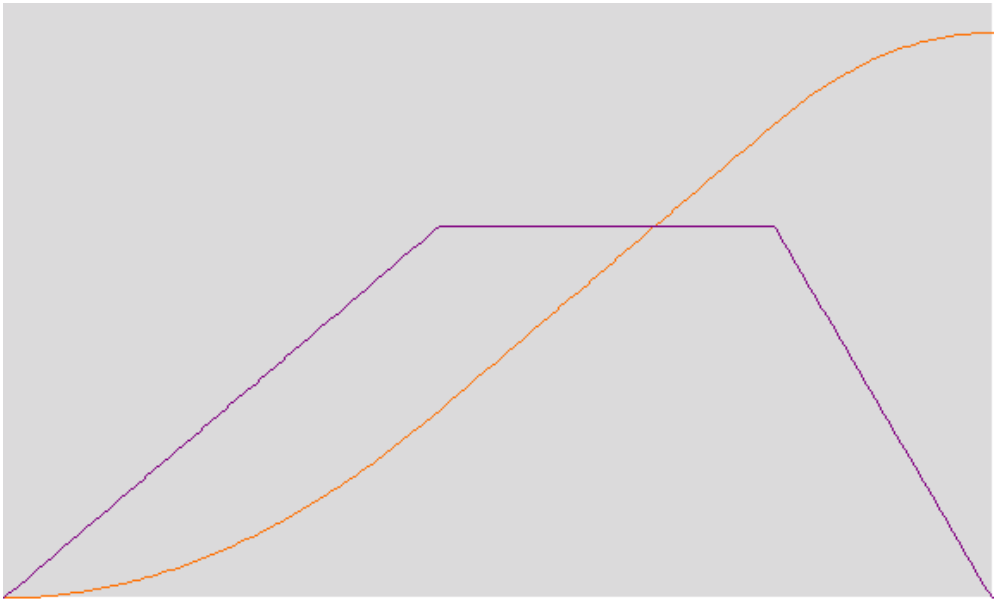
Une rampe de vitesse sur un asservissement de position

Rapprochons nous de notre objectif, c'est à dire un asservissement de position où l'on peut fixer une limite d'accélération/décélération, et aaussi de vitesse. Nous souhaitons obtenir une courbe de vitesse trapézoïdale :



La différence avec l'exemple précédent est que l'on n'asservit pas la vitesse mais la position. Cela signifie que la consigne est la distance totale en pas que le moteur doit parcourir depuis son zéro. Si nous démarrons une trajectoire trapézoïdale, l'aire du trapèze doit être égale à la consigne de position.

Par exemple, sur la courbe ci-dessous, l'entrée du filtre est un échelon (non représenté). Cette entrée correspond à une consigne de position demandée par l'utilisateur. La sortie est la consigne de position filtrée, lissée par les contraintes limitant l'accélération et la vitesse. Elle a une forme de 'S' (en orange), et c'est l'intégrale de la courbe trapézoïdale :



En orange, la consigne de position que l'on souhaite obtenir. En violet, la vitesse obtenue si l'on suit cette consigne.

Toute la difficulté de ce filtre est d'anticiper le moment ou il faut commencer à décélérer, de manière à ce que la vitesse soit nulle au moment où l'on atteint la position de consigne.

control_system/filters/quadramp

Description

Le traitement est beaucoup plus compliqué que dans le cas du filtre ramp, puisqu'il faut anticiper la décélération pour pouvoir arriver à la consigne de position avec une vitesse nulle.

Structures

```
struct quadramp_filter {
   uint32_t var_2nd_ord_pos;
   uint32_t var_2nd_ord_neg;
   uint32_t var_1st_ord_pos;
   uint32_t var_1st_ord_neg;

   int32_t previous_var;
   int32_t previous_out;
}
```

var_1st_ord_pos et var_1st_ord_neg sont les limites de la dérivée première (*vitesse* dans le cas d'un asservissement de position). var_2nd_ord_neg et var_2nd_ordpos sont les limites de la dérivée seconde (*accélération* dans le cas d'un asservissement de position).

Interface

`void quadramp_init(struct quadramp_filter * r);`

> Initialise les valeurs de la structure : var_pos, var_min, derivate_var_pos, derivate_var_neg à 0xFFFFFFFF. Dans ce cas, le filtre agit comme le filtre qui recopie l'entrée.

`int32_t quadramp_do_filter(struct quadramp_filter * r, int32_t in);`

> Applique le filtre sur la valeur in et retourne le résultat.

`void quadramp_set_2nd_order_vars(struct quadramp_filter * r, uint32_t var_2nd_ord_pos, uint32_t var_2nd_ord_neg);`

`void quadramp_set_1st_order_vars(struct quadramp_filter * r, uint32_t var_1st_ord_pos, uint32_t var_1st_ord_neg);`

> Les accesseurs pour configurer le filtre.

Exemple

Accélération max = 1 Accélération max = -1 Vitesse max = 10 Vitesse min = -10 Position = 150

La sortie du filtre quadramp est suite de consignes de position P (on note sa dérivée V). Les valeurs seront les suivantes (si V=0 et P=0 à t=0):

```
V = 1,2,3,4, 5, 6, 7, 8, 9, 10,10,10,10,10, 10,  9,
P = 1,3,6,10,15,21,28,36,45,55,65,75,85,95,105,114,
```

```
V =   8,  7,  6,  5,  4,  3,  2,  1
P = 122,129,135,140,144,147,148,149,150
```

La dérivée de la sortie du filtre correspond à la vitesse de consigne, et est bien de forme trapézoïdale. La position, quant à elle, est en forme de S: une parabole à l'accélération et à la décélération, et une droite lorsque la est vitesse stable.

L'implémentation doit être faite entièrement avec des nombres entiers: même si l'utilisation de *double* (= flottant de 32 bits avec avr-gcc) simplifierait effectivement beaucoup le code grâce à la lib math, ce n'est pas possible car l'entrée est un entier sur 32 bits et la conversion en double ferait perdre quelques bits de précisions sur les grands nombres.

De plus, la mise à jour des paramètres de la rampe est une opération qui arrivera régulièrement (voir le module trajectory pour bien comprendre ça). Il faut être sûr que la

modification des paramètres de la rampe fonctionne à tout moment.

Test

Voici un apperçu de la sortie du filtre. La courbe rouge correspond à l'entrée et la courbe verte correspond à la sortie. A certains moments, les caractéristiques du filtres sont modifiés dynamiquement :

à t=0

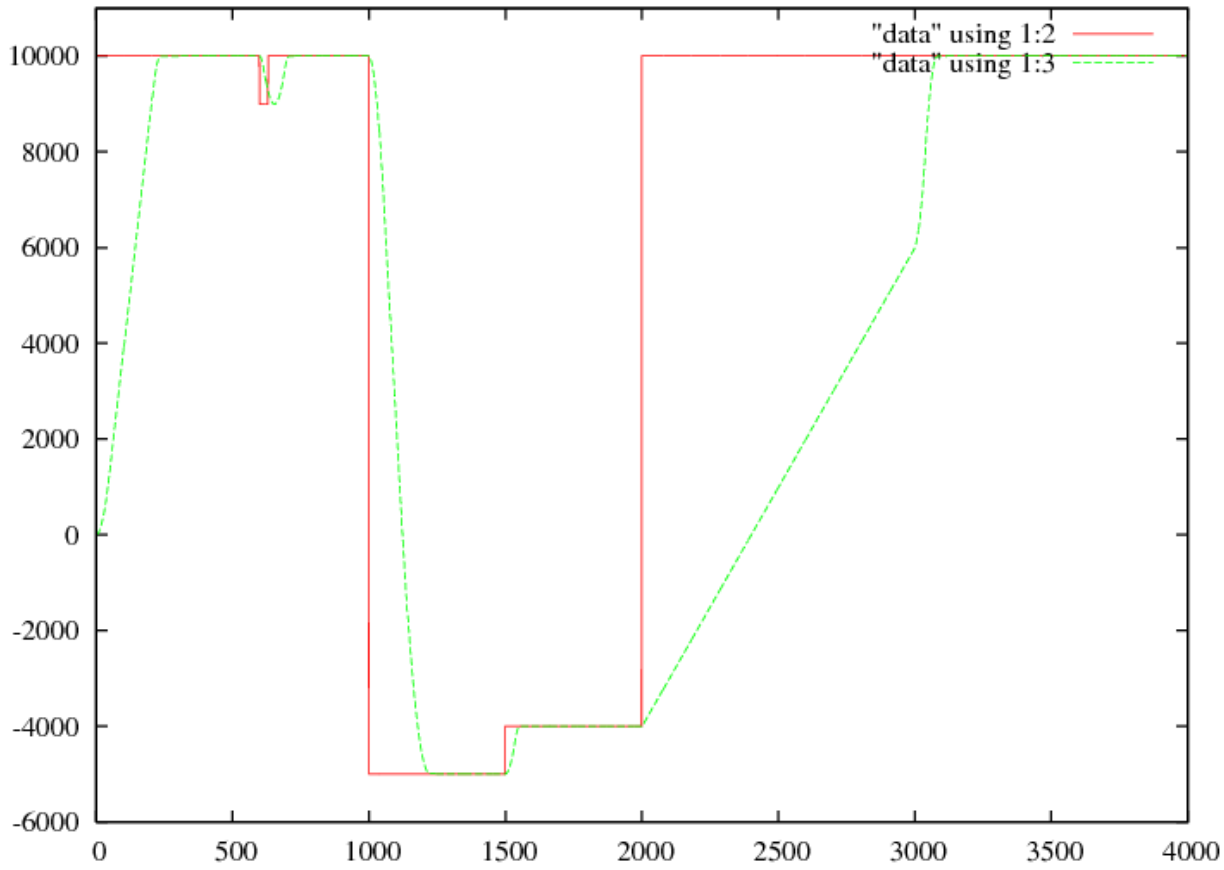
```
quadramp_set_1st_order_vars(&q, 50, 100);
quadramp_set_2nd_order_vars(&q, 1, 2);
```

à t=2000:

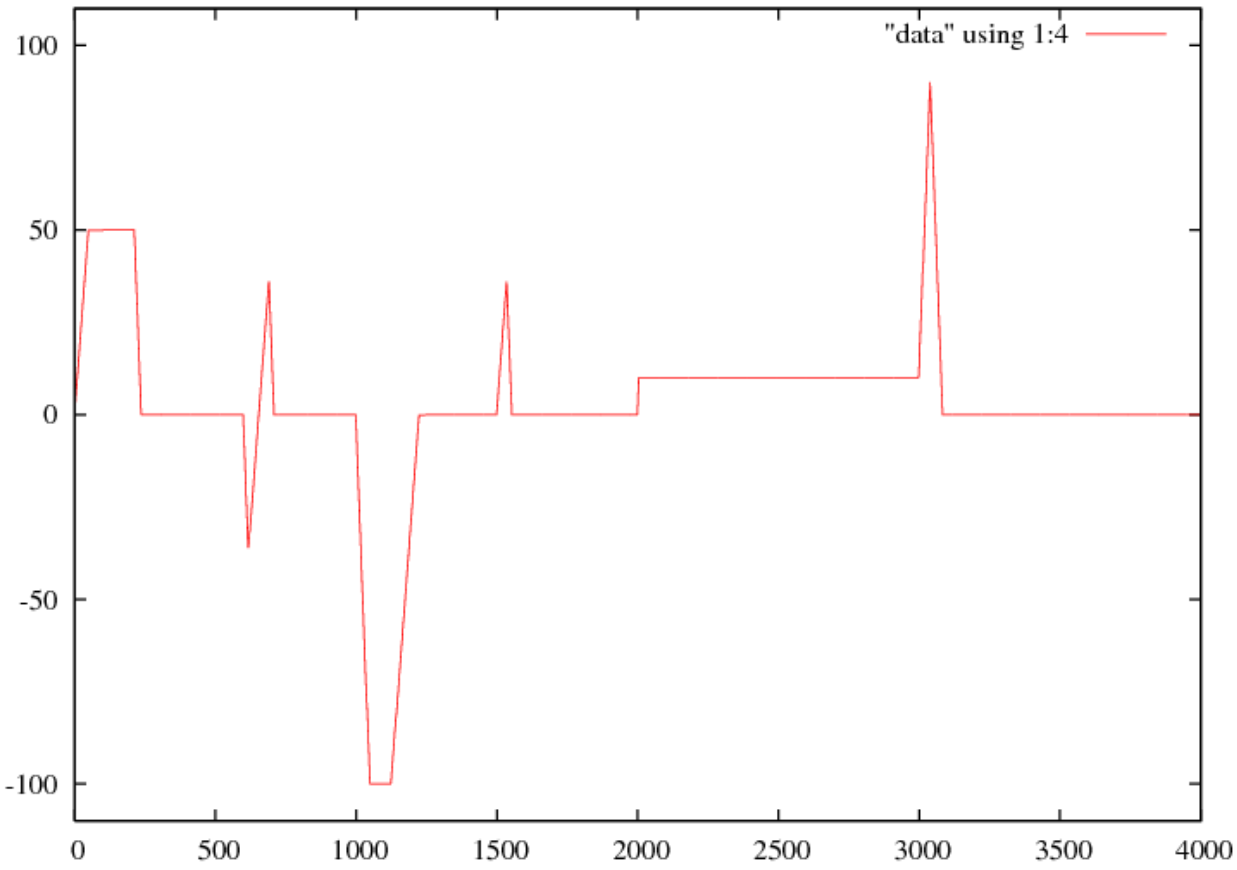
```
quadramp_set_1st_order_vars(&q, 10, 10);
quadramp_set_2nd_order_vars(&q, 2, 2);
```

à t=3000:

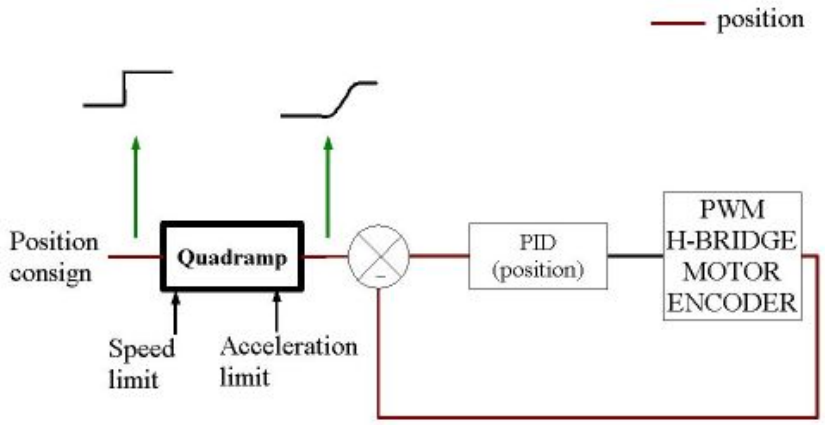
```
quadramp_set_1st_order_vars(&q, 100, 100);
```



La seconde courbe représente le même exemple et montre bien que la courbe de vitesse est trapézoïdale (dans le cas où la vitesse max est atteinte, sinon elle est triangulaire).



Voici un schéma de principe de l'asservissement complet :



Exemple de programme

```
struct cs cs;
struct pid_filter p;
struct quadramp_filter qr;

#define PERIOD 10000/SCHEDULER_UNIT /* 10 ms */
#define PRIORITY 100

/* fonctions définies ailleurs */
extern void motor_set(void *, int32_t);
extern int32_t motor_get_speed(void *);

int main(void)
```

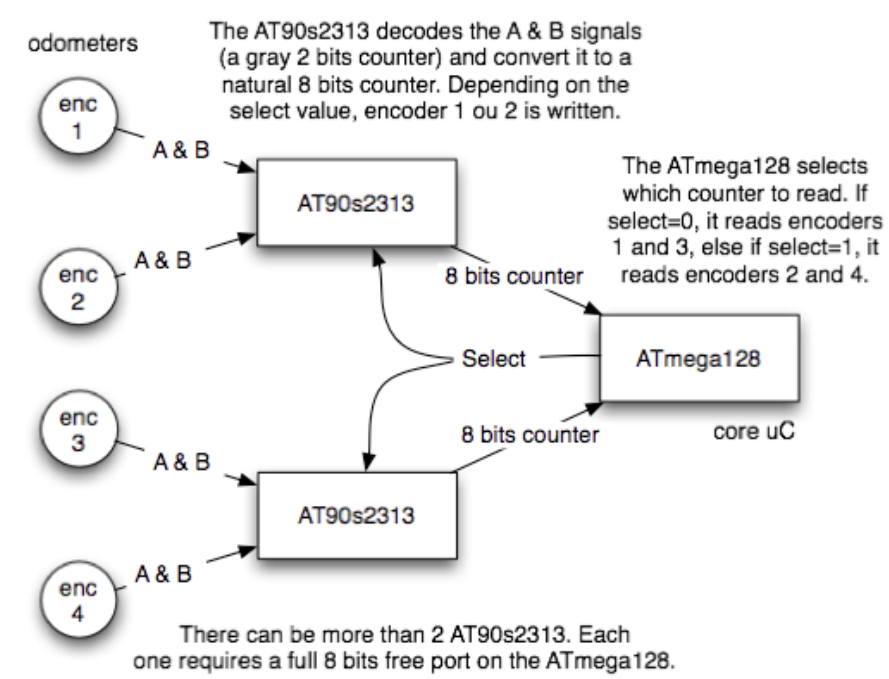
```
{
  pid_init(&p);
  pid_set_gains(&p, ...);
  pid_set_maximums(&p, ...);
  pid_set_out_shift(&p, ...);

  quadramp_init(&ra);
  quadramp_set_2nd_order_vars(&ra, 2, 1);
  quadramp_set_1st_order_vars(&ra, 10, 10);

  cs_init(&cs);
  cs_set_consign_filter(&qr, ramp_do_filter, &qr);
  cs_set_correct_filter(&cs, pid_do_filter, &p);
  cs_set_process_in(&cs, motor_set, NULL);
  cs_set_process_out(&cs, motor_get_speed, NULL);
  cs_set_consign(&cs, 0);

  scheduler_add_periodical_event_priority(cs_manage, (void *)&cs, PERIOD, PRIORITY);
  sei();

  wait_ms(1000);
  cs_set_consign(1000);

  while(1);
}
```

Interface avec les PWM et les odomètres

Dans tous les exemples précédents, nous supposions que les fonctions permettant de commander les moteurs et de lire les valeurs des odomètres étaient fournies. Or, la génération du signal PWM requiert la configuration de registres internes aux AVR, et la lecture des codeurs nécessite quant à elle un hardware spécifique pour le décodage des signaux.

devices/encoders/encoders_microb

Actuellement, il existe dans Aversive deux modules pour la lecture des odomètres adaptés à 2 hardware différents. Celui d'Eirbot et celui de Microb Technology que nous allons détailler. D'un point de vue électronique, et vu de l'ATmega128, le fonctionnement est relativement simple : la valeur des codeurs arrive sur un bus de 8 bits partagé par 2 codeurs. Le codeur sélectionné est le codeur pair ou impair selon l'état de la broche de sélection.



Interface

```
void encoders_microb_init(void);
```

Initialisation du module encoders_microb. Cette fonction met à 0 les compteurs internes au module.

```
void encoders_microb_manage(void);
```

Cette fonction lit l'état des bus pour récupérer la valeur des codeurs ayant un numéro pairs, puis modifie l'état de la broche de sélection pour préparer la prochaine lecture des codeurs impairs. L'appel suivant fait l'inverse. Cette fonction doit être appelée suffisamment souvent pour qu'entre 2 lectures d'un même codeur, sa valeur n'ait pas varié de plus de 127. En l'occurence, sur le robot de 2006, cette fonction est appelée toutes les 128us, permettant de lire un même codeur toutes les 256us.

```
int32_t encoders_microb_get(void * encoders_microb_data);
```

encoders_microb_get() permet de lire la valeur brute des compteurs. L'argument encoders_microb_data est un pointeur qui sera casté en (int) et représente le numéro du codeur. L'intérêt de mettre un void * est que le prototype reste compatible avec d'autres implémentations nécessitant le passage d'une structure complète.

hardware/pwm

Description

Le module pwm fournit quant à lui une fonction de sortie pour l'asservissement, c'est à dire la fonction void `process_in(void *, int32_t)`. Il permet de générer sur une broche un signal pwm (Pulse Width Modulation), c'est à dire un signal carré périodique dont le rapport cyclique est variable. C'est ce signal (en plus du bit de signe) qui sera amplifié par les LM18200T et qui servira à commander les moteurs. Ce module permet de générer ces deux signaux en une commande.

Interface

- `void pwm_init(void);`

Initialisation du module pwm.

- `void pwm_set(void *, int32_t);`

L'interface minimale pour fonctionner avec l'asservissement est de disposer d'une fonction comme ci-dessus. Le premier paramètre peur servir à passer par exemple le numéro du pwm.

Exemple

Nous pouvons reprendre l'exemple de la section précédente, en y ajoutant la gestion des pwm et des codeurs.

```
struct cs cs;
struct pid_filter p;
struct quadramp_filter qr;

#define PERIOD 10000/SCHEDULER_UNIT /* 10 ms */
#define PWM_MOTOR ((void *)PWM1A_NUM)
#define ENCODER_MOTOR ((void *) 0) /* numéro du codeur */

int main(void)
{
  pwm_init(); /* la configuration est dans un .h à part */

  encoders_microb_init();
  scheduler_add_periodical_event_priority(encoders_microb_manage, NULL, 1, 200);

  pid_init(&p);
  pid_set_gains(&p, ...);
  pid_set_maximums(&p, ...);
  pid_set_out_shift(&p, ...);

  quadramp_init(&ra);
  quadramp_set_2nd_order_vars(&ra, 2, 1);
  quadramp_set_1st_order_vars(&ra, 10, 10);

  cs_init(&cs);
  cs_set_consign_filter(&qr, ramp_do_filter, &qr);
  cs_set_correct_filter(&cs, pid_do_filter, &p);
  cs_set_process_in(&cs, pwm_set, PWM_MOTOR);
  cs_set_process_out(&cs, encoders_microb_get_value, ENCODER_MOTOR);
  cs_set_consign(&cs, 0);

  scheduler_add_periodical_event_priority(cs_manage, (void *)&cs, PERIOD, 100);
  sei();

  wait_ms(1000);
  cs_set_consign(1000);

  while(1);
}
```

Asservissement polaire (angle/distance)

Concentrons nous maintenant sur une application plus spécifique à un robot pour la coupe en essayant d'asservir son angle et sa distance avec 2 moteurs, tout en se repérant sur le terrain. Nous réutiliserons pour celà tout ce que nous avons vu plus haut.

devices/robot/robot_system

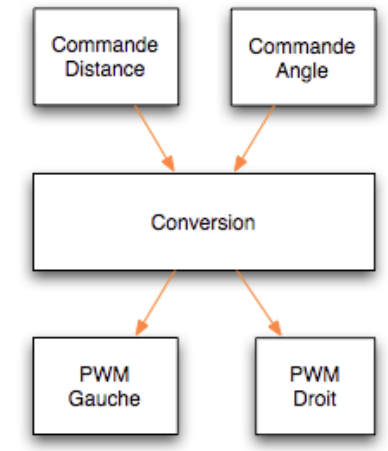
Description

Nous l'avons vu plus haut, nous ne souhaitons pas asservir les roues indépendamment l'une de l'autre. Nous avons fait le choix d'asservir 2 grandeurs qui en dépendent :

- la somme des valeurs retournées par les 2 codeurs des deux roues divisée par 2 (distance=theta)
- la différence des valeurs retournées par les 2 codeurs des deux roues divisée par 2 (angle=alpha)

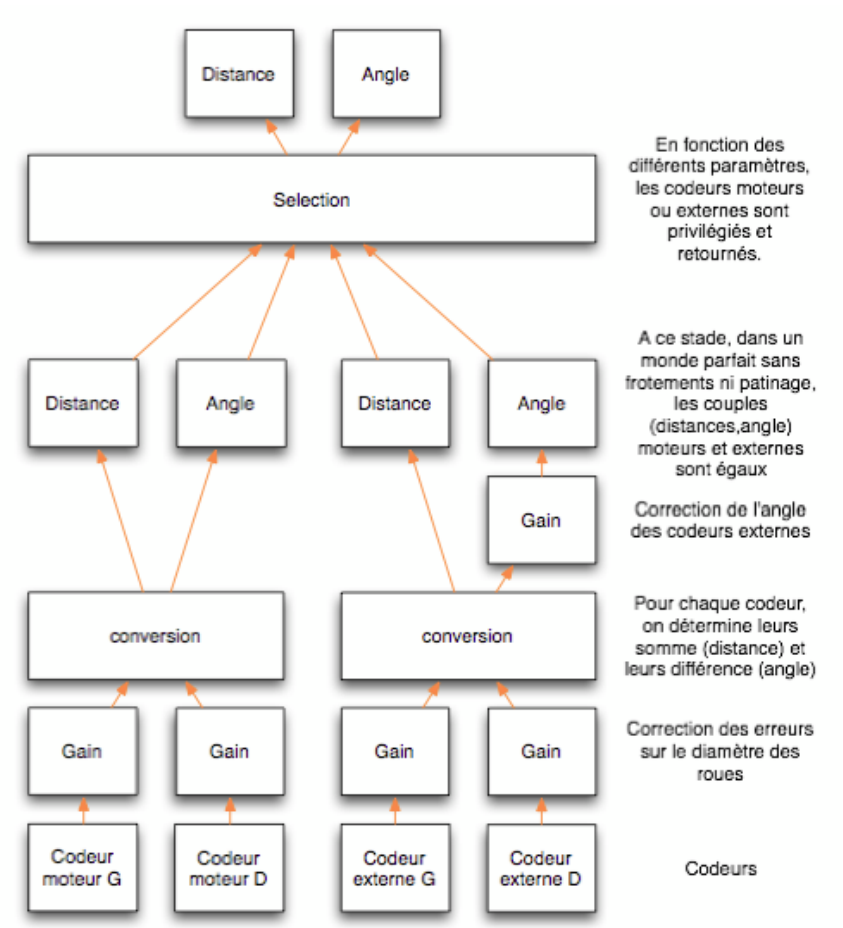
Ce module effectue cette opération simple pour la lecture des codeurs (`angle_get()` et `distance_get()`) et pour l'envoi des PWM (`angle_set()` et `distance_set()`).

Les PWM d'angle et de distance n'existent bien sûr pas physiquement. Leur interface est la même que celle des vraies PWM, ce qui permet de les utiliser comme une fonction `process_in` dans le control system manager.



Pour les codeurs incrémentaux, le travail est plus complexe car l'objectif est de tirer parti des deux paires de codeurs (moteurs et externes). Nous l'avons vu, notre robot nous dispose de roues codeuses sur chacun des moteurs, mais aussi des roues codeuses folles (non motorisées). Pour la propulsion du robot, nous devons donc gérer 4 odomètres. Le module robot_system tente de tirer partie de l'information des 4 codeurs pour fournir les codeurs virtuels d'angle et de distance.

L'idée est de pouvoir utiliser indépendamment les codeurs des moteurs ou les codeurs externes, et de pouvoir basculer de l'un à l'autre à tout moment. Cela implique quelques opérations mathématiques pour que les codeurs retournent la même chose, malgré qu'ils soient placés à des endroits différents. Une dernière fonctionnalité est que le module permet d'appliquer des coefficients correcteurs aux codeurs afin de compenser des défauts mécaniques (petite erreur sur le diamètre de la roue par exemple). De cette manière, il ne reste plus qu'à appliquer un coefficient multiplicateur à l'angle pour passer des codeurs externes aux codeurs moteurs.



Structures

- dans angle_distance.h :

```
struct rs_wheels {
	int32_t left;
	int32_t right;
};
```

Structure décrivant les coordonnées du robot en polaire.

```
struct rs_polar {
	int32_t distance;
	int32_t angle;
};
```

Structure décrivant les compteurs des roues.

- dans robot_system.h

```
struct robot_system
{
	uint8_t flags;
	struct rs_polar pmot_prev;
	struct rs_polar pext_prev;

	struct rs_wheels wmot_prev;
	struct rs_wheels wext_prev;

	struct rs_polar virtual_pwm;
```

```
	struct rs_polar virtual_encoders;
	f64 ratio_mot_ext;

	/* Motor encoders */
	int32_t (*left_mot_encoder)(void *);
	void* left_mot_encoder_param;
	f64 left_ext_gain;

	int32_t (*right_mot_encoder)(void *);
	void* right_mot_encoder_param;
	f64 right_ext_gain;

	/* External encoders */
	int32_t (*left_ext_encoder)(void *);
	void* left_ext_encoder_param;
	f64 left_mot_gain;

	int32_t (*right_ext_encoder)(void *);
	void* right_ext_encoder_param;
	f64 right_mot_gain;

	/* PWM */
	void (*left_pwm)(void *, int32_t);
	void *left_pwm_param;
	void (*right_pwm)(void *, int32_t);
	void *right_pwm_param;
};
```

Cette structure stocke toutes les informations nécessaires au bon fonctionnement de robot_system. Pointeurs vers les fonctions d'entrées/sorties (pwm et codeurs), les gains à appliquer sur chacun des codeurs, le rapport entre la voie-codeurs et la voie-moteurs, des flags et compteurs internes.

Interface de angle_distance.h

`void rs_get_polar_from_wheels(struct rs_polar * p_dst, struct rs_wheels * w_src);`

> Conversion d'une structure contenant les valeurs des codeurs vers une structure contenant la somme des codeurs sur 2, et leur différence sur 2.

`void rs_get_wheels_from_polar(struct rs_wheels * w_dst, struct rs_polar * p_src);`

> Conversion inverse de la fonction ci-dessus.

Interface de robot_system.h

`void rs_init(struct robot_system * rs)`

> Initialise la structure rs : met à 0 toute la structure, sauf le ratio à 1.0

`void rs_set_ratio(struct robot_system * rs, double ratio);`

> Définit le rapport entre la distance qui sépare les roues des moteurs et la distance qui sépare les roues externes.

`void rs_set_left_pwm(struct robot_system * rs, void (*left_pwm)(void *, int32_t), void *left_pwm_param);`

`void rs_set_right_pwm(struct robot_system * rs, void (*right_pwm)(void *, int32_t), void *right_pwm_param);`

`void rs_set_left_mot_encoder(struct robot_system * rs, int32_t (*left_mot_encoder)(void *), void *left_mot_encoder_param, double gain);`

`void rs_set_right_mot_encoder(struct robot_system * rs, int32_t (*right_mot_encoder)(void *), void *right_mot_encoder_param, double gain);`

```
void rs_set_left_ext_encoder(struct robot_system * rs, int32_t (*left_ext_encoder)(void *), void
*left_ext_encoder_param, double gain);

void rs_set_right_ext_encoder(struct robot_system * rs, int32_t (*right_ext_encoder)(void *), void
*right_ext_encoder_param, double gain);
```

Spécifie les fonctions d'entrée/sortie : pwm gauche et droit, et codeurs gauche et droit. Pour les codeurs, on spécifie aussi le gain qui sert à corriger la valeur de chaque codeur.

```
void rs_set_angle(void * rs, int32_t angle)
```

Met à jour les PWM pour prendre en compte la nouvelle commande d'angle, et sauvegarde dans last_command.

```
void rs_set_distance(void * rs, int32_t dist)
```

Met à jour les PWM pour prendre en compte la nouvelle commande de distance, et sauvegarde dans last_command.

```
int32_t rs_get_angle(void * rs)
```

Retourne l'angle courant

```
int32_t rs_get_distance(void * rs)
```

Retourne la distance courante

```
int32_t rs_get_ext_angle(void * rs)

int32_t rs_get_mot_angle(void * rs)

int32_t rs_get_ext_distance(void * rs)

int32_t rs_get_mot_distance(void * rs)
```

Même chose, mais en demandant spécifiquement la distance ou l'angle selon les codeurs moteurs ou les roues folles.

```
void rs_update(void * rs);
```

Lit les codeurs, et effectue les calculs pour mettre à jour les codeurs virtuels internes.

```
void rs_set_flags(struct robot_system * rs, uint8_t flags);
```

Pour le moment le seul flag utilisé est RS_USE_EXT qui permet de déterminer si on utilise les codeurs externes ou les codeurs moteurs pour faire la conversion en angle/distance. Cette fonction peut être appelée en cours d'utilisation sans perturber le fonctionnement.

Calcul de la position (x,y,a)

math/vect2

Ce module fournit des fonctions et structures de données pour utiliser des vecteurs 2D dans des repères cartésiens ou polaires. Les vecteurs utilisent le scalaire *Real* pour leurs coordonnées (par défaut c'est un *double*). Il est documenté ici.

devices/robot/position_manager

Description

Le module position_manager des fonctions de récupération de la position du robot. C'est un "manager" car il fournit une fonction position_manage() dont le rôle est de mettre à jour la position du robot sur le terrain. Il permet donc de maintenir à jour la position du robot sur le terrain (x,y,a) en utilisant robot_system.

Structures

```
struct robot_physical_params
{
  double track_cm;
  double distance_imp_per_cm;
};
```

Structure décrivant le nombre d'impulsions pour faire un cm, anisi que la voie des codeurs réels (ou virtuels) du robot.

```
struct xya_position
{
  double x;
  double y;
  double a;
};
```

Structure permettant de stocker une position cartésienne du robot sur le terrain. L'angle a est stocké en radian.

```
struct xya_position_int16_t
{
        int16_t x;
        int16_t y;
        int16_t a;
};
```

Même structure, mais les valeurs sont entieres. L'angle est stocké en degré.

```
struct robot_position
{
        struct robot_physical_params phys;
        struct xya_position pos_d;
        struct xya_position_int16_t pos_int16_t;
        struct rs_polar prev_encoders;
        struct robot_system * rs;
}
```

Structure stockant la position instantanée du robot, ainsi qu'un pointeur vers la structure robot_system qui décrit comment récupérer la valeur des codeurs angle/distance.

Interface

void posistion_init(struct robot_position * pos)

Initialise la structure position (tout à 0).

void posistion_set_related_robot_system(struct robot_position * pos, struct robot_system * rs)

Sauvegarde dans pos le pointeur vers la structure robot_system associée. La structure

rs est utilisée pour récupérer la valeur des codeurs virtuels en angle et distance.

```
void position_set(struct robot_position * pos, int16_t x, int16_t y, int16_t a);
```

Définit la position actuelle du robot.

```
void position_set_physical_params(struct robot_position * pos, double track_cm, double
distance_imp_per_cm);
```

Définit les paramètres physiques du robot (distance entre les roues, nombre d'impulsions par cm.

```
void position_manage(struct robot_position * pos)
```

Recalcule la position absolue en (x,y,a) en fonction du delta de position des roues et des paramètres physiques du robot.

```
inline int16_t position_get_x_s16(struct robot_position *pos);

inline int16_t position_get_y_s16(struct robot_position *pos);

inline int16_t position_get_a_deg_s16(struct robot_position *pos);

inline double position_get_x_double(struct robot_position *pos);

inline double position_get_y_double(struct robot_position *pos);

inline double position_get_a_rad_double(struct robot_position *pos);
```

Accesseurs.

Détection de blocage

robot_control/blocking_detection_manager

Description

La détection de blocage est utile dans le cas d'un asservissement d'un robot pour la coupe E=M6. Elle permet de repérer les obstacles et donc d'agir en conséquence en effectuant par exemple une manoeuvre de contournement.

Il y a au moins deux types de blocages :

- 1- Les roues motrices patinent. Dans ce cas, on peut détecter le blocage en remarquant que les valeurs retournées par les codeurs des moteurs et les codeurs des roues sont incohérentes.
- 2- Les roues motrices sont bloquées. L'erreur de l'asservissement augmente, jusqu'à atteindre un seuil critique. Le fait d'avoir un asservissement qui effectue des rampes de vitesse simplifie ce type de détection de blocage, car on considère que les paramètres de la rampe sont reglées de telle manière que moteur est physiquement capable d'appliquer cette rampe.

En pratique, et comme cette année il est probable que ne disposions pas de codeurs sur les roues motorisées, nous n'utiliseront qu'un seul algorithme pour détecter les blocages (il a cependant fait ses preuves). Le module récupérera les informations du module control_system de manière autonome. L'asservissement "surveillé" doit être un asservissement de postion. Celà nous permet d'utiliser ce module aussi bien sur les roues

motrices du robot que sur un autre actionneur motorisé. Le plus délicat est de régler les seuils pour trouver un bon compromis entre une réaction trop ou pas assez rapide.

La fonction suivante est appelée intervale régulier pour détecter les blocages.

```
Si la vitesse mesurée est supérieure à un seuil ET que l'erreur d'asservissement est supérieure à un autre seuil
       Si cpt < CPT_MAX
               Incrementer cpt
Sinon
       cpt = 0
```

Une autre fonction peu à tout moment être appelée, pour demander s'il y a blocage :

```
Si cpt == CPT_MAX
       etourner 1
retourner 0
```

Interface

`void bd_init(struct blocking_detection * bd, struct cs *cs)`

Initialisation du module

`void bd_set_thresholds(struct blocking_detection * bd, uint32_t speed_thres, uint32_t err_thres, uint16_t cpt_thres)`

Initialise les seuils

`void bd_reset(struct blocking_detection * bd)`

Remet le compteur à 0. Cette fonction devrait être appelée dès qu'un blocage a été pris en compte.

`void bd_manage(struct blocking_detection * bd)`

A appeler régulierement pour mettre à jour les informations de blocages.

`uint8_t bd_get(struct blocking_detection * bd)`

Retourne 1 en cas de blocage.

Génération de trajectoire

devices/robot/trajectory_manager

Description

On entre dans une partie difficile. Ce module sert a définir des consignes pour l'asservissement de manière a pouvoir controler le robot de manière simple, par exemple en lui précisant une destination (x,y). Le principe de base est d'ajouter une fonction événement périodique (environ toutes les 0.1 secondes) qui va mettre à jour les consignes de distance et d'angle de l'asservissement. Le module fournit aussi des fonctions plus simples qui permettent de faire l'interface entre l'asservissement (qui travaille en pas) et le développeur/stratège (qui lui se sent plus à l'aise avec des centimètres).

Interface

Encore en développement, mais on peut se faire une bonne idée de la version finale ici (http://cvsweb.droids-corp.org/cgi-bin/viewcvs.cgi/aversive/modules/devices/robot/trajectory_manager/trajectory_manager.h?revision=1.4.4.5&view=markup&pathrev=b_zer0)

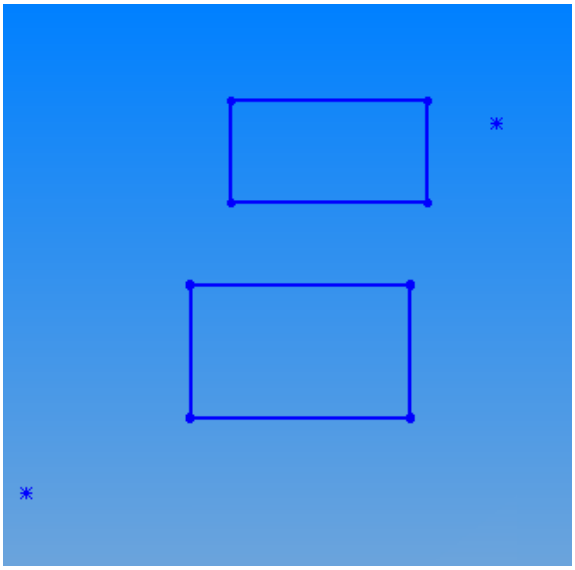
devices/robot/obstacle_avoidance

Description

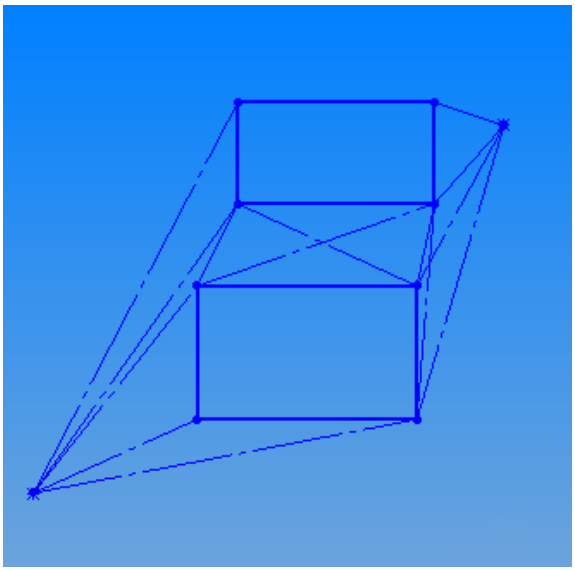
Ce module permet de générer des trajectoires avec des points de passage en contournant une liste d'obstacles.

L'algorithme utilisé est l'algorithme du "point visible" (ce n'est peut être pas le vrai nom). Il prend en entrée une liste de polygones (convexes pour simplifier), représentants les objets à éviter lors du calcul de trajectoire. Le point de départ et d'arrivée ainsi que les dimensions de l'aire de jeu sont également fournis.

Avec ces données, l'algorithme essaye de trouver le chemin le plus court pour relier le point de départ au point d'arrivée tout en évitant les objets de l'aire de jeu. Dans l'exemple ci-dessous, le robot part du point en bas a gauche et doit arriver au point en haut a droite en évitant les 2 rectangles (2 robots adverses, par exemple).



Dans un premier temps, l'algorithme va tracer tous les segments reliant tous les points ayants une vue "directe" l'un de l'autre ; autrement dit, tous les points que se "voient". Dans notre exemple, il s'agit des traits en pointillé.

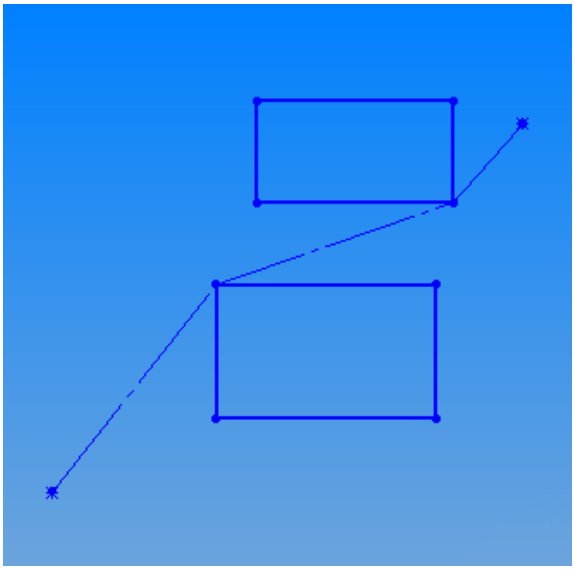


Il est à noter que les traits formant les côtés des polygones sont également générés (si les polygones ne s'entrecroisent pas). Nous obtenons un graphe dont les arrêtes sont les segments générés précédemment. Le but est ensuite de pondérer chaque segment par sa longueur. Ainsi un segment de longueur 2 aura un poids de 2, et ainsi de suite.

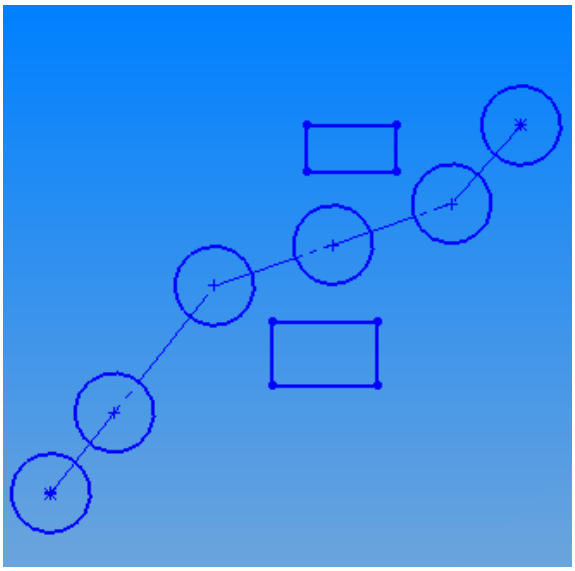
Nous terminons en calculant le chemin le plus court (en terme de poids) reliant le point de départ au point d'arrivée, ceci étant fait en utilisant l'algorithme de Dijkstra (http://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra)

Résultats

Le robot n'a alors plus qu'à suivre la suite de segment retourné.

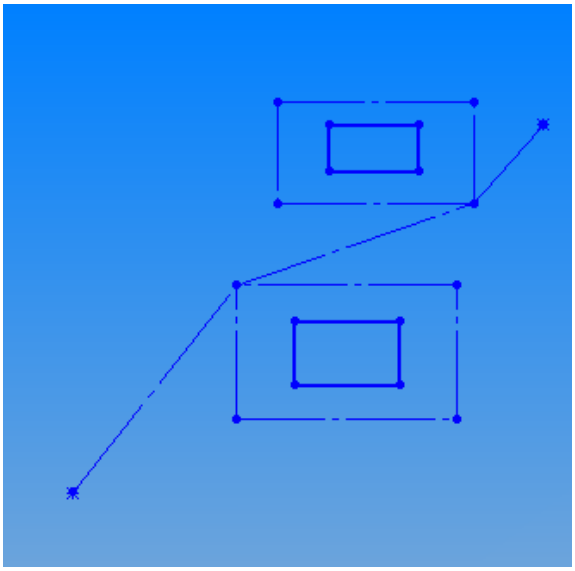


Voila le résultat final (notre robot est représenté par un rond). Le robot arrive au point final tout en évitant les obstacles.

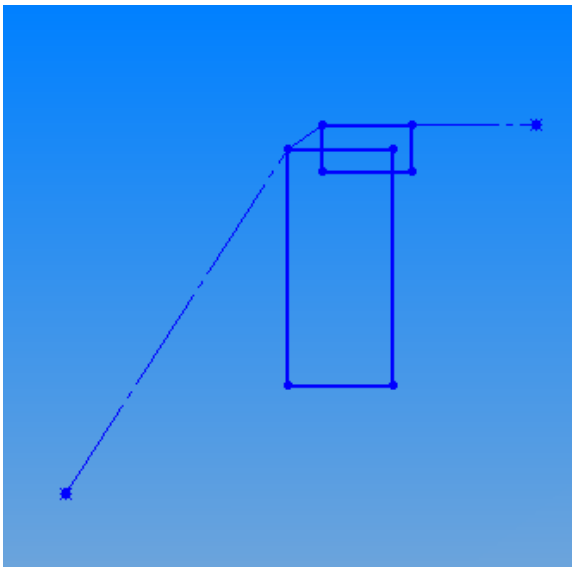


Notez que cet algorithme donne le chemin le plus court pour une configuration donnée au moment du calcul de la trajectoire. Si les objets à éviter ne sont pas fixe, d'autres calculs de trajectoires seront sûrement nécessaires.

Un autre point important est le fait que le chemin retourné frôle les sommets des polygones, ce qui veut dire que dans la réalité notre robot frôlera le robot adverse ou les objets du terrain. Pour palier à ce problème, une solution est de surévaluer la taille des objets à éviter. Ici les vrai objets (en traits pleins) ont été représenté par des polygones légèrement plus grands.



L'algorithme arrive également à calculer un chemin correct même si les polygones se croisent. En effet, si un sommet est inclu dans un autre polygone, aucun segment n'y sera relié (le sommet n'est pas "vu" par les autres sommets)



Vidéo

Dans Fichier:Traj direct.3gp, nous avons une première trajectoire sans obstacle : le passe par 4 points sur le terrain, le premier à quelques centimètres devant lui, le second assez loin à sa gauche, puis il pivote et recule vers le troisième, et enchaine vers le dernier point en bas à gauche sur la vidéo.

Dans la seconde vidéo Fichier:Traj avoid.3gp, on place un obstacle au milieu du terrain. La seule information supplémentaire dont dispose le robot est la position et la taille de l'obstacle. On lui demande d'effectuer la trajectoire à nouveau, mais certaines parties deviendraient alors impossible à faire sans toucher l'obstacle. Il ajoute alors automatiquement des points de passage supplémentaires.

Conclusion

Cette documentation décrit le fonctionnement de l'asservissement de Microb Technology pour 2008. Comme précisé au début de cette page, elle est encore en train d'évoluer et plusieurs points peuvent encore être incomplets ou même incohérents. Je suis ouvert à tout commentaire.

Nous avons vu que l'asservissement d'un robot à la coupe n'est pas une chose facile, mais en séparant proprement le programme en plusieurs briques que le l'on peut assembler, il est plus facile d'y voir clair. Ce document peut servir de base pour quelqu'un qui souhaiterait réimplémenter un asservissement pour Eurobot, ou bien de documentation de notre asservissement. Comme plusieurs équipes l'ont fait, il est tout à fait envisageable de reprendre une partie du code et de l'adapter sur un robot différent du notre.

- Dernière modification de cette page le 1 janvier 2008 à 16:29.