Asservissement PID

Sommaire

Introduction:

J'espère avec ce tutoriel expliquer en langage simple ce qu'est l'asservissement avec un régulateur PID. Nous pourrons ensuite rentrer dans les détails du codage d'un tel régulateur.

Enfin, nous estimerons sa consommation en mémoire, en temps, et en puissance de calcul, afin de parvenir à un résultat pour choisir un micro-contrôleur pour implémenter cette technique.

De quoi s'agit il ?

L'histoire commence avec une action à réaliser, par exemple déplacer un robot de 0.5 m en avant.

L'asservissement par PID s'applique à de nombreux domaines, mais je préfère vous présenter directement l'utilisation concrète que nous en avons dans le robot.

Pour avancer droit, on peut intuitivement penser qu'il faut appliquer une tension constante sur les deux moteurs pendant le délai nécessaire.

On obtient ce délai par un petit calcul prenant en compte le diamètre et la vitesse de rotation des roues, sans oublier le temps de montée (accélération) et le temps de descente (décélération)... le tour est joué... ou pas! Malheureusement, cela ne fonctionne pas.

Même avec des moteurs de bonne qualité, la vitesse dépend souvent de la charge des accumulateurs, du poids du robot et d'un tas d'autres paramètres qui varient selon les circonstances. Du coup les résultats obtenus sont vite erronés.

Dans notre cas, on souhaite que le robot avance, mais les deux moteurs ne répondent pas de la même façon, le robot dévie immédiatement de sa trajectoire.

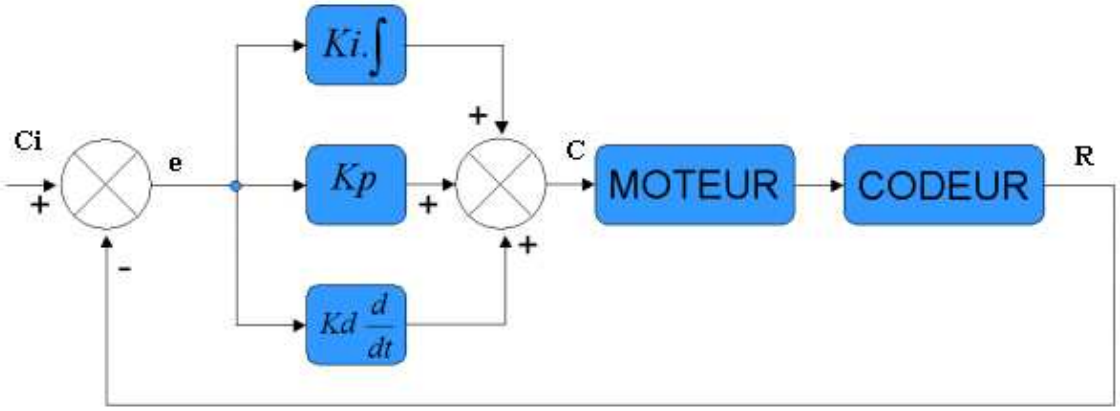
Donc, on ordonne au robot de se déplacer de façon précise (avancer de 0.5m) et il suit une trajectoire différente, et donc atteint une destination différente!! Ce qu'il fait n'est pas trop loin de la consigne, mais des imperfections apparaissent. Du coup, nous allons simplement dire qu'il a commis une **erreur**. Et l'idée ici c'est de rendre notre robot conscient de cette erreur qu'il a commis ou mieux est en train de commettre pour qu'il **corrige** automatiquement sa trajectoire et sa vitesse. On dit alors qu'on effectue un **asservissement**.

Principe d'asservissement

Après cette introduction, il est temps de passer aux choses sérieuses.

L'asservissement consiste tout simplement en la récupération d'une information sur la vitesse du moteur (ici grâce aux roues codeuses) puis en son utilisation pour ajuster la tension de commande. Il existe de nombreuses méthodes d'asservissement, nous présentons ici celle que nous utilisons dans notre robot (et la plus connue de toutes) : le PID (Proportionnel Intégrale Dérivée).

Principe d'asservissement PID



1. Ci : Consigne initiale (ce qu'on veut qu'il fasse)
2. e : erreur entre la consigne initiale et la réalité
3. C : Consigne appliquée au moteur
4. R : Grandeur réelle mesurée (réalité)

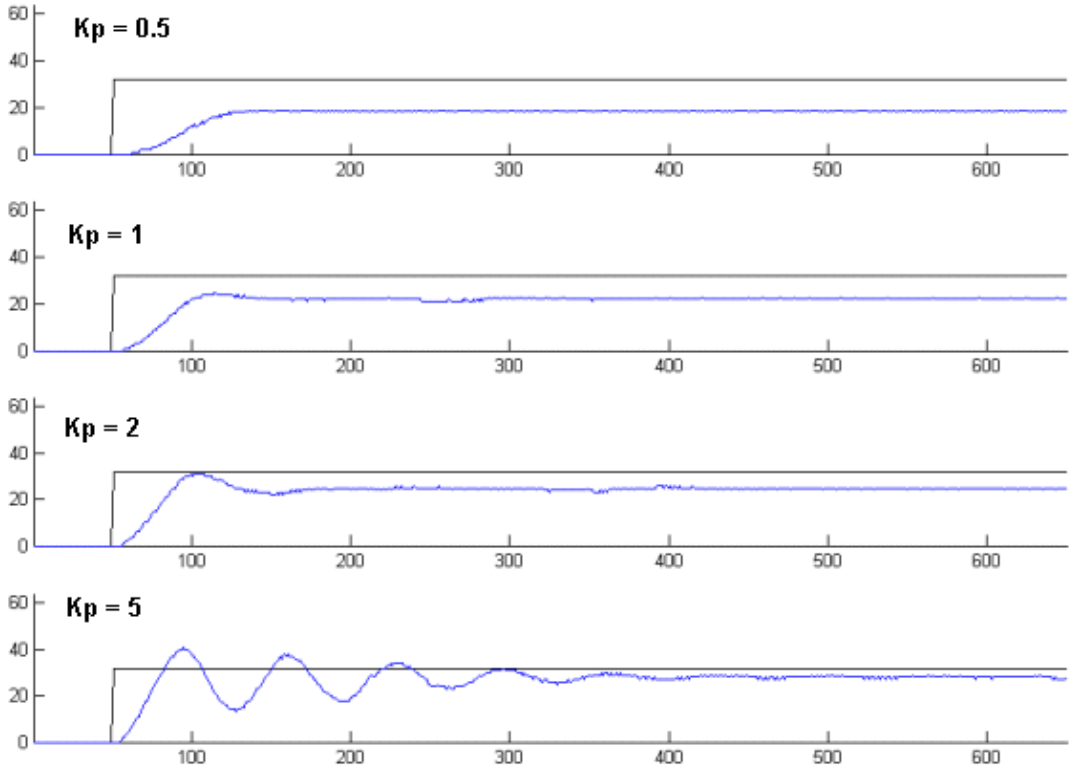
Le principe de base de tout asservissement est de prendre des mesures à la sortie, de les réinjecter à l'entrée pour comparer à la consigne afin d'obtenir l'erreur (ie la différence entre la mesure réelle et la consigne demandée). Quand elle est nulle, le robot a atteint sa destination et donc il n'y a plus rien à actionner. Ce système est dit en boucle fermée, puisque la sortie du système est réinjectée dans l'entrée.

Maintenant que vous connaissez le principe général, caractérisons le PID, c'est à dire le Proportionnel, Intégral, Dérivé. Ces trois blocs bleus correspondent chacun à une fonctionnalité : Ki est le coefficient intégral, Kp le coefficient proportionnel et Kd le coefficient dérivé. Commençons par le plus simple : l'asservissement Proportionnel.

Asservissement P

L'asservissement Proportionnel est le plus important du PID, car c'est principalement lui qui permet de donner de la puissance au moteur.

Pour voir son impact sur le déplacement du robot, il suffit d'annuler les autres coefficients Ki=0 et Kd=0. En faisant varier la valeur de Kp, on peut observer son impact. Voici quelques courbes que nous avons obtenues ainsi :



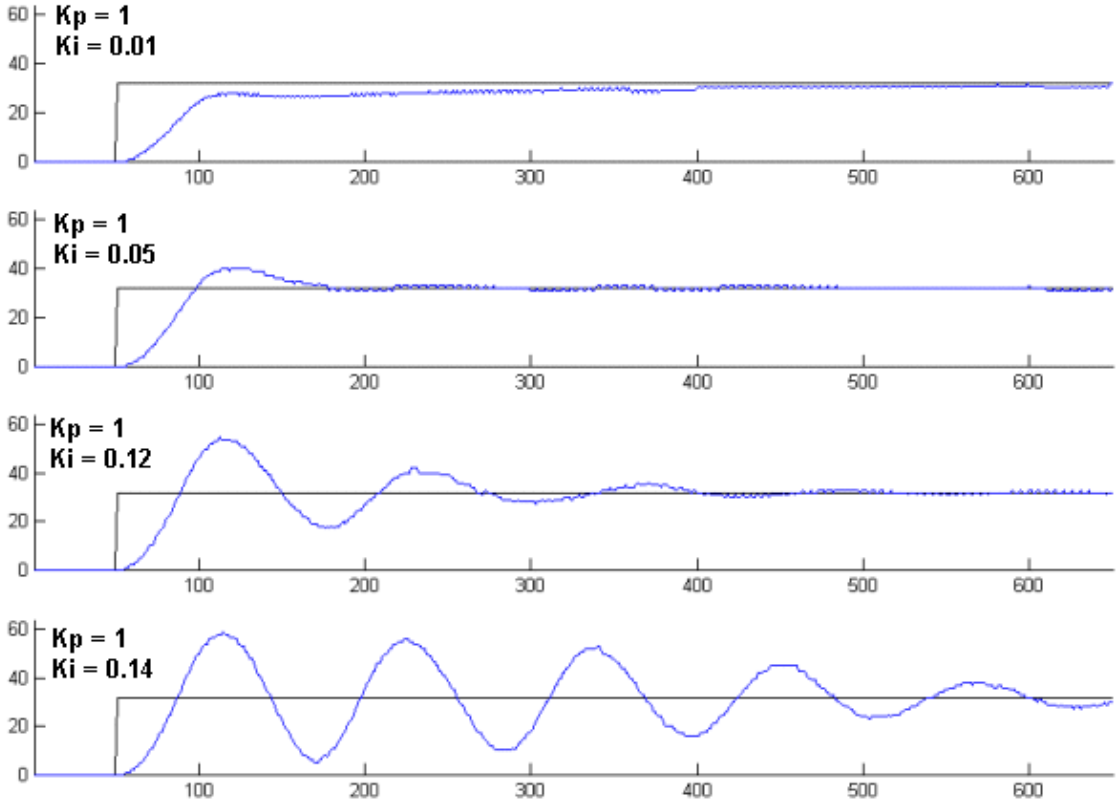
La courbe noire représente la consigne initiale (Ci) et la courbe bleue le résultat réel (R).

Les résultats sont simples à analyser, plus Kp est grand plus on atteint la consigne rapidement. Mais lorsque Kp augmente, des oscillations autour de la consigne apparaissent et surtout, la consigne n'est pas exactement atteinte! Cette petite différence entre la grandeur réelle et la grandeur désirée une fois que le système est stabilisé s'appelle l'erreur statique. Pour compenser cette erreur statique, nous allons rajouter le terme Intégral.

Asservissement PI

Pour éliminer l'erreur statique,l'idée est ici d'intégrer l'erreur depuis le début et d'ajouter cette nouvelle erreur à la consigne jusqu'à ce qu'elle s'annule. Lorsqu'elle est nulle, le terme intégral se stabilise et compense parfaitement l'erreur entre la consigne et la valeur réelle.

Pour voir ceci de près, il suffit demettre Kp=1 puis de faire varier le coefficient de l'intégrale Ki afin de voir son impact sur le système:

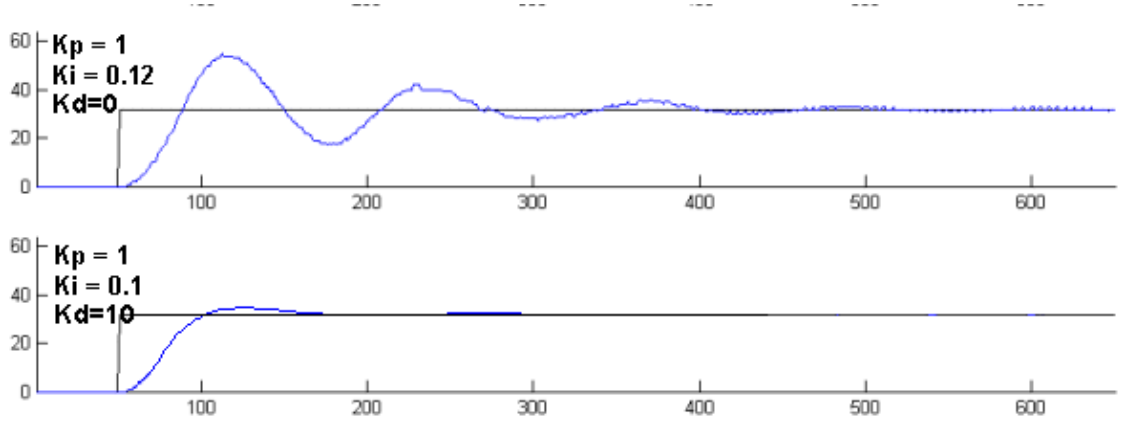


Les choses s'améliorent, on arrive à éliminer l'erreur statique. On voit comme pour l'asservissement P qu'en augmentant le coefficient Ki on atteint plus rapidement la consigne mais que notre système oscille plus violemment et dépasse la consigne de nombreuses fois.

C'est pour remédier à ce second problème que l'on ajoute le terme Dérivé.

Asservissement PID

L'idée est à présent de dériver l'erreur et d'ajouter cette valeur à celle obtenue précédemment afin de limiter les dépassements. En faisant varier le terme dérivé, on obtient les résultats suivants :



Comme vous pouvez le voir sur les courbes, en ajoutant le terme dérivé, on diminue le dépassement (en anglais overshoot).

Résumé

Pour résumer, le régulateur PID est un régulateur à boucle fermée qui utilise la différence entre l'entrée et la sortie pour modifier la consigne dans le but d'atteindre une réponse égale à la valeur d'entrée. Pour ceci il utilise 3 termes:

- Le terme Proportionnel qui permet d'augmenter la vitesse de montée (atteint la consigne le plus rapidement

possible).

- Le terme Intégral qui réduit l'erreur statique.
- Le terme Dérivé qui réduit le dépassement.

Attention

Le terme dérivé peut causer des instabilités dans le système si on l'augmente de manière inconsidérée ou si on effectue un asservissement de la vitesse. En effet, la dérivée d'une vitesse est une accélération, donc on amplifie les bruits d'accélération. Du coup, si on travaille sur une régulation de vitesse, il vaut mieux diminuer ce coefficient ou même l'annuler et se contenter de jouer les 2 autres termes.

Implémentation du PID

Après la théorie, passons à la pratique. Dans un premier temps, on désire effectuer un asservissement **NUMERIQUE** sur un micro-contrôleur. Il faut prévoir sa consommation en temps de calcul et en mémoire, adapter sa précision (nombre de bits pour représenter les nombres)... On souhaite bien sûr en même temps optimiser notre programme pour que ça ne soit pas trop compliqué à mettre en oeuvre.

Pour commencer, on va supposer qu'on a déjà optimisé les paramètres Kp, Ki et Kd.

Nous avons deux entrées, l'une qui donne la consigne initiale (Ci) provenant du reste du système et l'autre qui donne la valeur réelle mesurée par les roues codeuses (R). Notre but est de calculer la consigne C à appliquer aux moteurs. Nous effectuons d'abord une soustraction pour déterminer l'erreur, puis nous calculons les 3 termes proportionnel, Intégral et dérivé en fonction de cette erreur :

Terme Proportionnel

Ce terme est le plus simple à calculer, il s'agit d'une simple multiplication de l'erreur par le coefficient Kp. Il n'y a rien de bien compliqué, sauf que la multiplication nous amène à nous poser des questions à propos du format des données dans notre système. Cette multiplication sera plus ou moins rapide suivant la précision qu'on veut garantir pour le résultat. Ce problème étant commun à tous les termes, on le traitera plus tard.

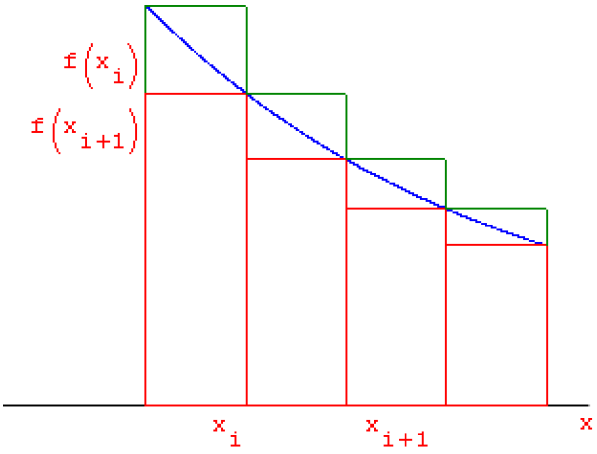
Terme Intégral

Après le terme Proportionnel, nous allons calculer le terme Intégral. L'intégration n'est pas une opération facile, mais en la voyant comme un simple calcul de surface, nous pouvons utiliser l'un des algorithmes d'estimation d'intégrale puis sommer ces erreurs. En général l'erreur diminue au cours du temps donc notre intégrale ne diverge pas (condition nécessaire bien que non suffisante).

Je vous présente ici deux méthodes possibles pour le calcul de l'approximation d'une intégrale, vous pourrez en trouver bien d'autres sur internet.

Méthode des rectangles:

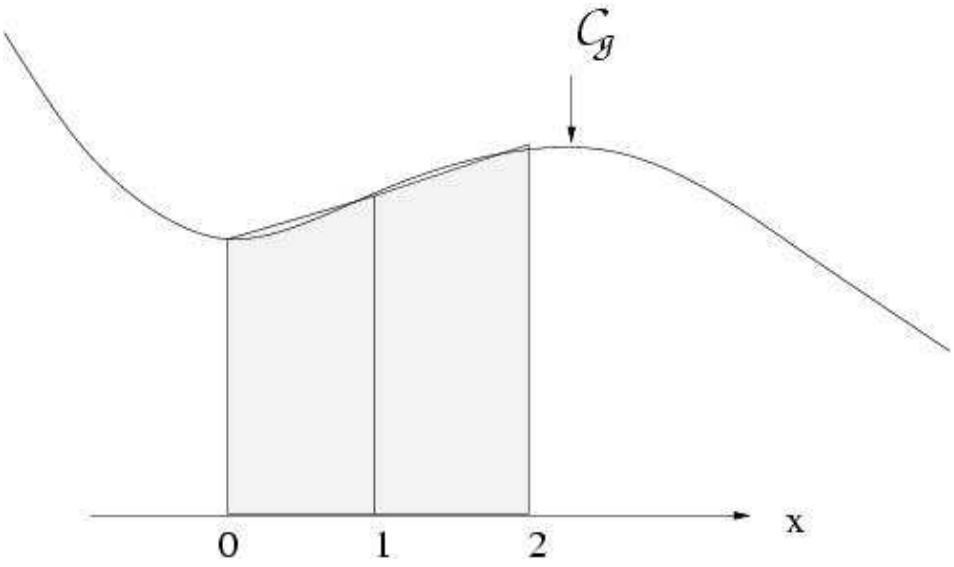
Elle consiste à diviser la région limitée par la courbe en de petits rectangles, et faire la somme.de leurs surfaces :



Comme vous pouvez le constater sur l'exemple ci-dessus, la précision n'est pas excellente, et elle empire si on augmente la longueur des intervalles d'échantillonnage. On obtient soit une approximation supérieure (rectangles verts) soit une approximation inférieure (rectangles rouges).

Méthode des Trapèzes:

Cette seconde méthode est beaucoup plus précise puisque, au lieu de calculer la surface d'un rectangle touchant la courbe en un point, on va calculer la surface d'un trapèze dont les deux angles supérieurs appartiennent à la courbe:



Si on gagne en précision avec la deuxième méthode, on perd en rapidité de calcul, puisque calculer la surface d'un rectangle est plus simple que de calculer celle d'un trapèze. Il n'y a donc pas de "meilleure" méthode en soi, il va falloir choisir. Dans notre cas, on pourrait se satisfaire de la méthode des rectangles qui a l'avantage de consister en de simples additions qui ne consomment pas grande chose en terme de temps. En essayant d'échantillonner à une fréquence bien adaptée, et si on a bien optimisé nos paramètres, l'erreur oscille moins et du coup on peut se permettre d'approximer la valeur sur un petit intervalle à une valeur constante.

Truc pour gagner du temps

Bon, pour calculer la surface du rectangle on fait dt*e avec dt une valeur temporelle qui sépale 2 prises d'échantillons. ceci veut dire qu'on va faire encore une multiplication de plus pour chaque prise d'échantillon!! Bon c trop!!!

Si on regarde de près, cette valeur est la même tout le temps (on échantillonne à des instants réguliers) donc le truc c'est d'intégrer cette valeur dans le Ki et de cette manière uniquement calculer à chaque cycle I(i)=I(i-1)+e (avec I(i) le terme intégrale à l'instant i) puis on termine par une seule multiplication avec Ki quand nécessaire et on gagne une multiplication dans chaque cycle...

;)

En gros :

I(i)=I(i-1)*e

Terme Dérivé

Comme on nous a appris à l'école au tout début, le calcul de la dérivée en un point revient à trouver la pente de la courbe à cet endroit et du coup on a de(t)/dt=(e(t)-e(t-1))/delta t . Avec delta t la période d'échantillonnage.

Truc pour gagner du temps

Comme pour l'integrale, on peut optimiser nos calculs en intégrant le terme delta t dans le coefficient Kd et du coup on gagne une **DIVISION** qui fait horreur dans le monde embarqué.

En Gros :

D=e(i)-e(i-1)

Pondération des termes

Une fois calculés, on va faire une pondération sur les termes en les multipliant successivement par Kp, Ki et Kd et en sommant tout le monde. Ainsi on obtient notre Consigne à appliquer au moteur..

Un peu de Code

Pour finir, nous allons examiner comment coder un PID simple en C :

Il nous faut comme entrées Ci, et R, et on retourne C.

Dans notre programme, nous avons besoin des variables:

- e=Ci-R (variable globale)
- P=e (on va utiliser e directement mais ainsi l'algorithme est plus clair),
- I(i)=I(i-1)+e, (on utilise une variable globale I qui represent l'accumulateur jusqu'à l'etat i-1)
- D=e(i)-e(i-1). (on crée une variable globale Old_e qui va contenir l'ancienne valeur de e)
- des constantes pré-calculées Kp, Ki et Kd.

Voilà un exemple de code C pour la fonction PID, il faut soit le mettre dans une interruption qui se produit quand on reçoit l'échantillon suivant, soit l'intégrer dans une boucle dans le programme principal synchronisé sur la fin du timer et qui sature entre 2 prises d'échantillons.

```
/**
*PID : fonction qui effectue un asservissement PID
*Remarque:
* Il faut commencer par d&eacute;finir les constantes Kp, Ki, Kd dans le programme principal
* exemple:
* #define Kp 1
* #define Ki 0.05
* #define Kd 0.01
*/

double PID(long Ci,long R)
{
 long P,D;
 double C;
 Old_e = e;
 e = Ci-R;
```

```
 P=e;//Terme Proportionnel
 I = I+e;//Terme Integral
 D = e-Old_e;//Terme D&eacute;riv&eacute;
 C = Kp*P+Ki*I+Kd*D;// oops on a fini

 return C;
}
```

Ce code n'est pas idéal, mais il sert de démonstration du principe. Je vais expliquer ici quels problèmes il pose :

Le problème le plus important intervient au moment du calcul de la dérivée : quand la consigne passe brusquement de 0 à une valeur non nulle, la dérivée est infinie. L'effet du terme dérivé est alors bien supérieur à celui des autres termes, du coup le robot reçoit une consigne exagérément grande et fait un bond brusque en avant. Notre robot se met à osciller en vitesse, et avec un peu de malchance, la consigne diverge, et on obtient un comportement erratique.

Donc il faut filter l'erreur avant de la dériver de façon à ce qu'elle monte en douceur et ne diverge pas. Du coup, notre code va changer un peu :

```
/**
*PID : fonction qui effectue un asservissement PID
*Remarque:
* Il faut commencer par definir les constantes Kp, Ki, Kd dans le programme principal
* exemple:
* #define Kp 1
* #define Ki 0.05
* #define Kd 0.01
*/

double PID(float Ci,float R)
{
 long P,D;
 double C;
 e = Ci-R;
 FR=filter(R);//On filtre le retour
 P=e;//Terme Proportionnel
 I = I+e;//Terme Integral
 D = FR-Old_R;//Terme D&eacute;riv&eacute;
 C = Kp*P+Ki*I+Kd*D;// oops on a fini
 Old_R = FR;//On sauvegarde

 return C;
}
```