

Éléments de robotique avec Arduino



Pascal MASSON

(pascal.masson@unice.fr)

*Version projection
Edition 2019-2020-V13*

P roportionnel I ntégral D érivé

Introduction

1. **Fonctionnement**
2. **Application : vitesse d'un moteur**
3. **Application : self balancing robot** EN COURS D'ECRITURE

□ Qu'est qu'un PID ?

- Le régulateur PID, appelé aussi correcteur PID (proportionnel, intégral, dérivé) est un système de contrôle permettant d'améliorer les performances d'un asservissement, c'est-à-dire un système ou procédé en boucle fermée. C'est le régulateur le plus utilisé dans l'industrie où ses qualités de correction s'appliquent à de multiples grandeurs physiques (c.f. Wikipedia).
- Le rôle d'un algorithme de correction PID est d'améliorer 3 des principales caractéristiques d'un système : la rapidité, la précision et la stabilité.

□ Dans quoi est-ce utilisé ?

- Pour un moteur par exemple, le PID assure une montée en régime rapide, une vitesse réelle très proche de celle demandée et un fonctionnement à la vitesse de consigne sans oscillations (accélérations, décélération...).



Drone, fusée



Stabilisateur



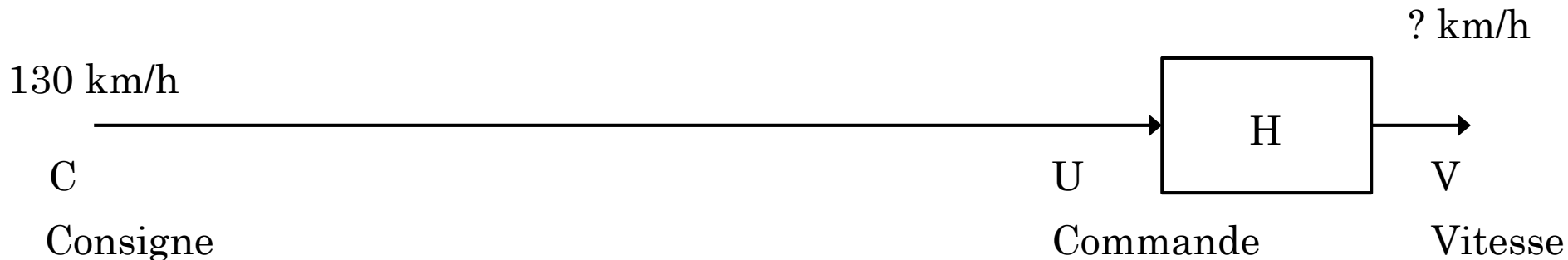
Self Balancing robot



Véhicule autonome

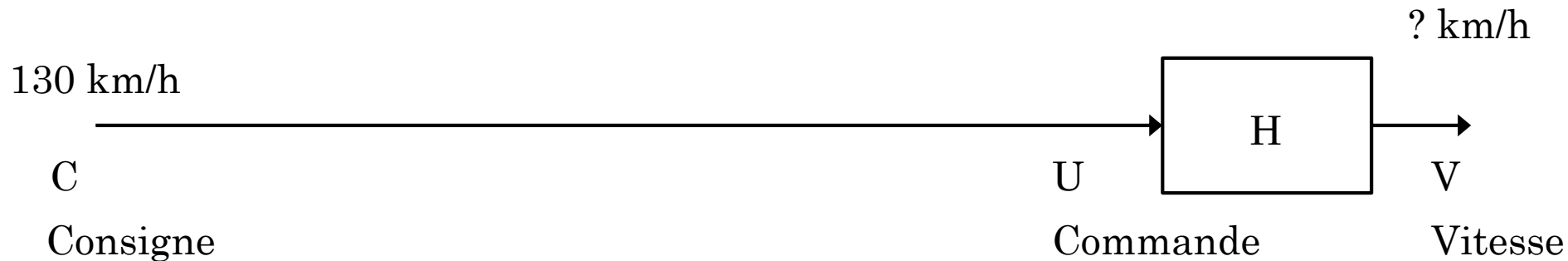
1.1. Sans asservissement

- Dans ce chapitre, nous allons construire le fonctionnement d'un PID en nous plaçant dans le cas de la voiture autonome
- Nous choisissons de rouler sur autoroute à la vitesse de 130 km/h avec une vitesse nulle au démarrage (prise du ticket au début de l'autoroute)
- Le régulateur de vitesse applique un certain régime sur le moteur (fonction H) qui correspond à la vitesse 130 km/h



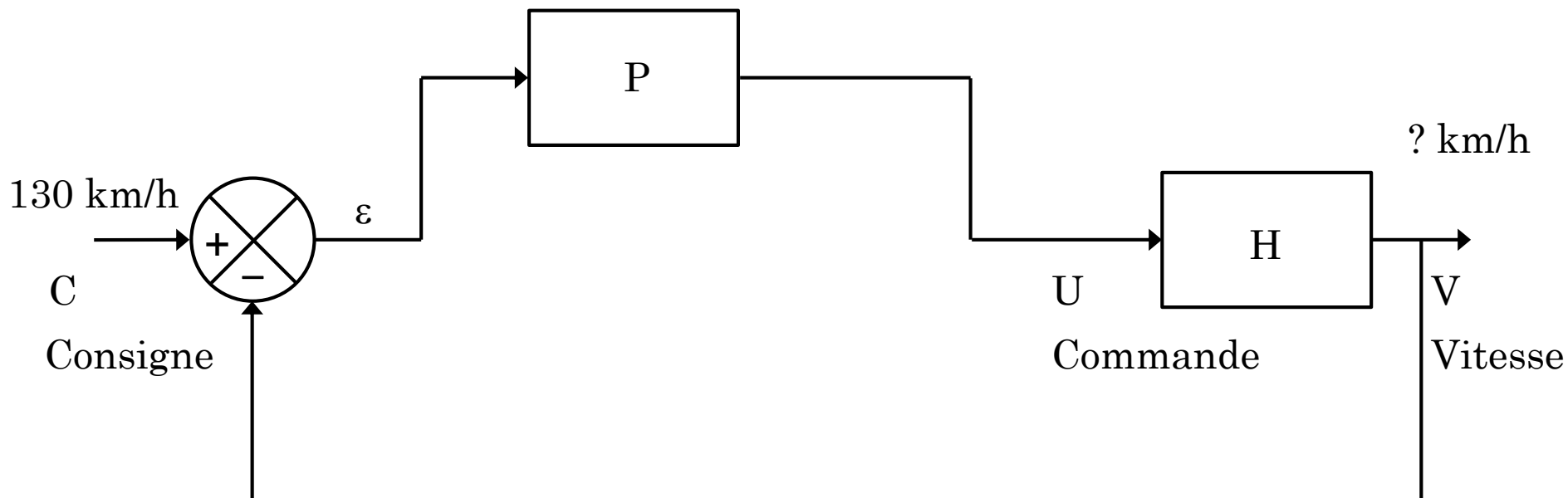
1.1. Sans asservissement

- La vitesse finale va dépendre de plusieurs facteurs comme la pente de la route et elle ne sera probablement pas 130 km/h



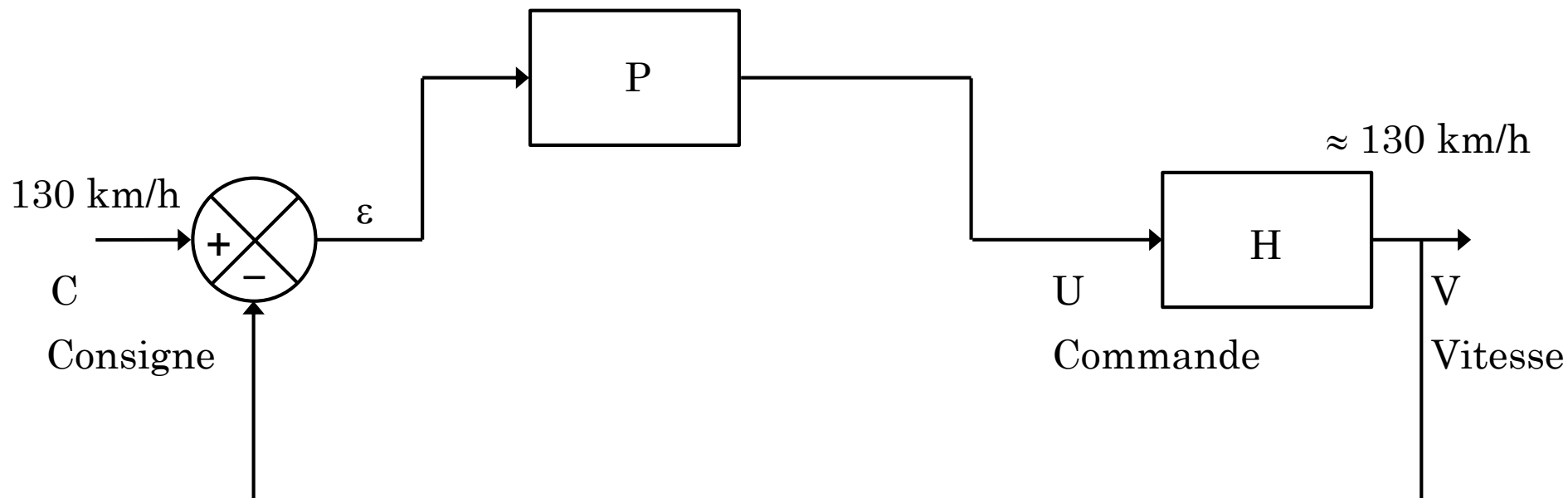
1.2. Asservissement Proportionnel

- On ajoute un capteur qui mesure la vitesse de la voiture et un élément qui change de façon Proportionnelle le régime du moteur en fonction de la différence entre la consigne et la vitesse mesurée
- Plus l'écart (ou erreur) entre la mesure et la consigne est grand plus le régulateur augmente le régime du moteur



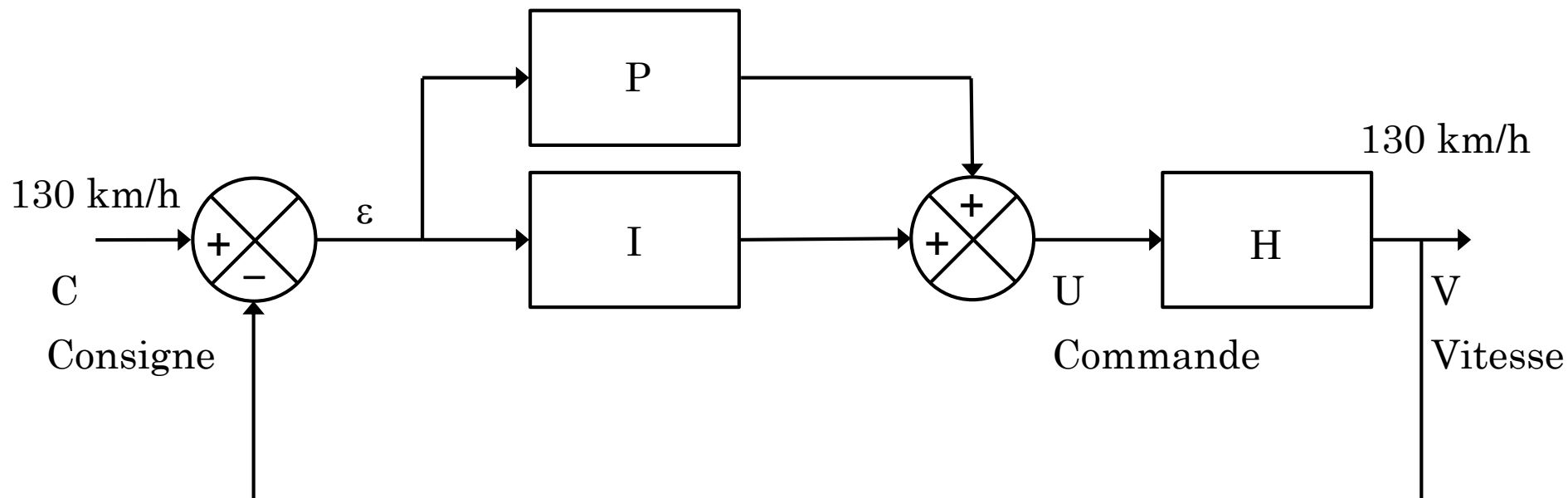
1.2. Asservissement Proportionnel

- Il existe toujours une erreur entre la vitesse de la voiture et la consigne
- Cette erreur est appelée « erreur statique »



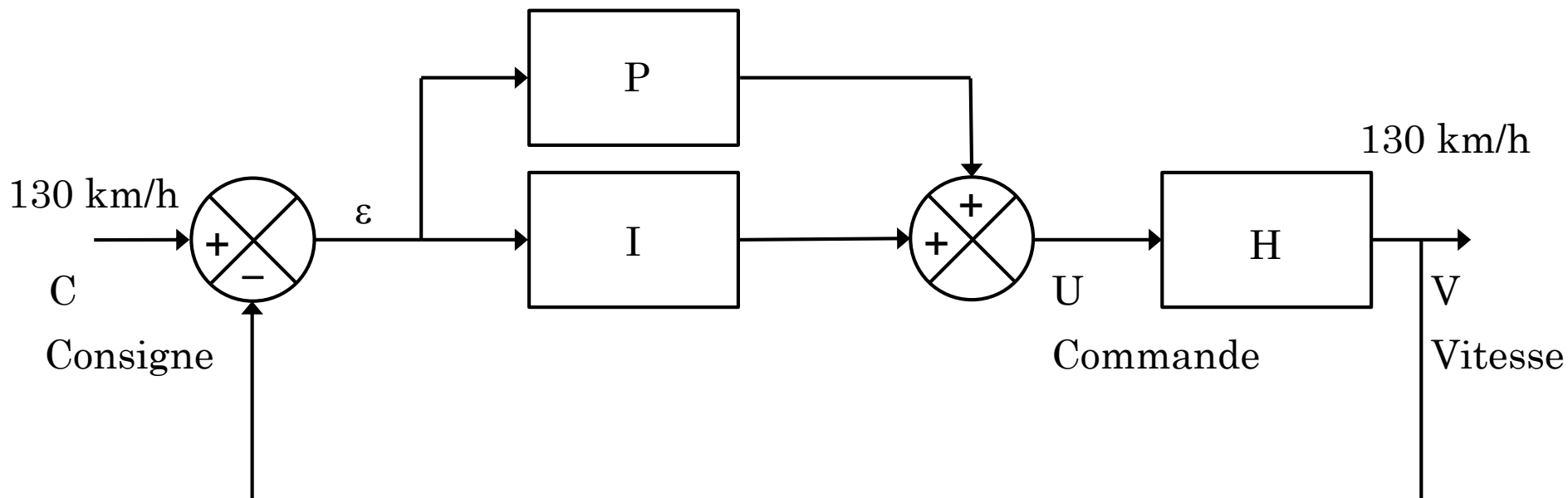
1.3. Asservissement Proportionnel Intégral

- A présent, on ajoute un système qui regarde l'erreur au cours du temps.
- Si cette erreur dure, on augmente le régime du moteur
- Si la vitesse dépasse la consigne, l'erreur devient négative et la voiture freine.



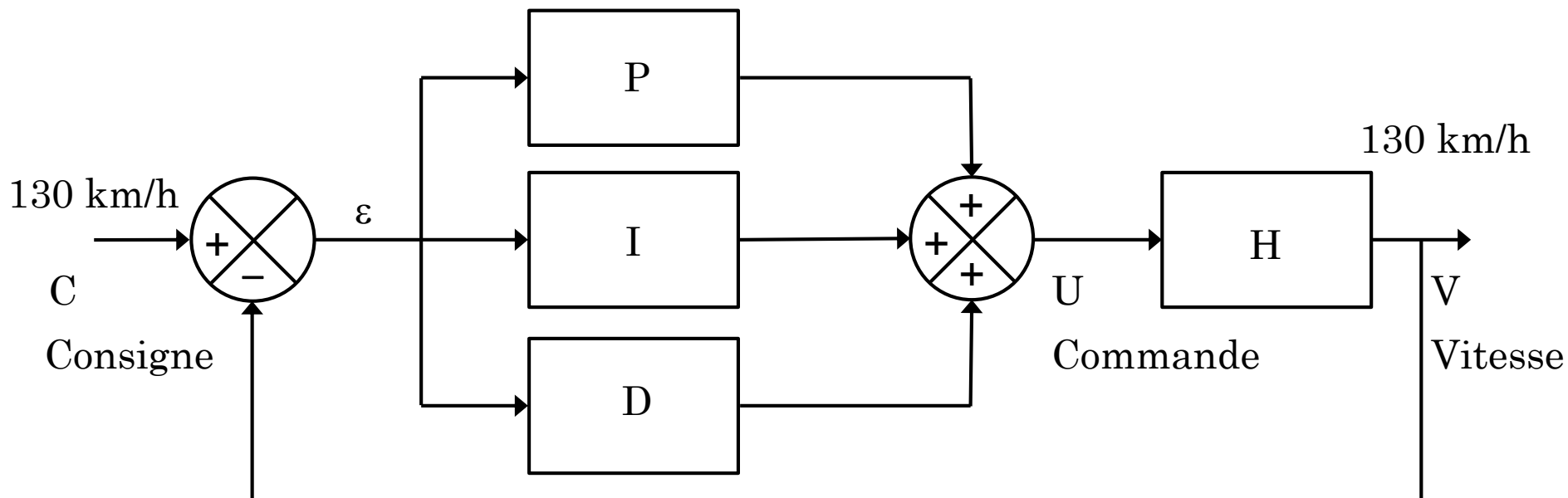
1.3. Asservissement Proportionnel Intégral

- Lorsque la voiture passe en dessous des 130 km/h, la voiture réaccélère
- Il y a des oscillations de la vitesse jusqu'à atteindre la consigne
- L'asservissement Intégral tient compte du temps qui s'écoule au fur et à mesure que la correction progresse, c'est à dire durant la progression de la réduction de l'écart $\varepsilon = C - V$



1.4. Asservissement Proportionnel Intégral Dérivé

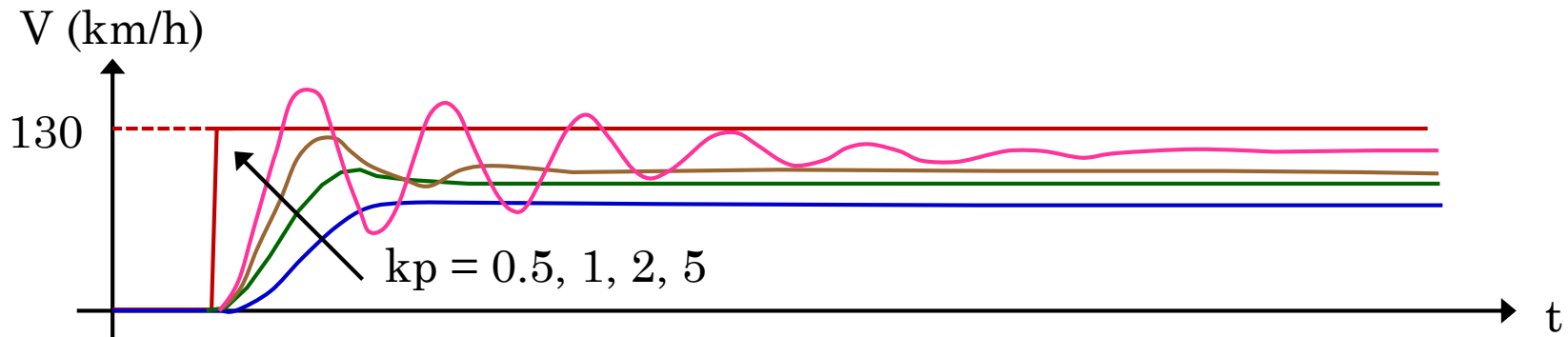
- La vitesse de la consigne est atteinte mais ça n'a pas été très efficace et il faut ajouter une troisième règle afin d'être plus performant.
- On joute un asservissement Dérivé qui anticipe : plus la vitesse se rapproche de la consigne moins la voiture accélère et moins elle se rapproche de la consigne, plus la voiture accélère



1.5. Mise en équation

□ Proportionnel

- L'asservissement Proportionnel est le plus important du PID
- Sa mise en équation est très simple : $U(t) = k_p \cdot \varepsilon(t) = k_p \cdot [C(t) - V(t)]$
- Où k_p est un gain sans dimension.
- Plus k_p est grand plus on se rapproche rapidement de la consigne sans jamais l'atteindre. On remarque aussi que lorsque k_p augmente, des oscillations autour de la consigne apparaissent.



1.5. Mise en équation

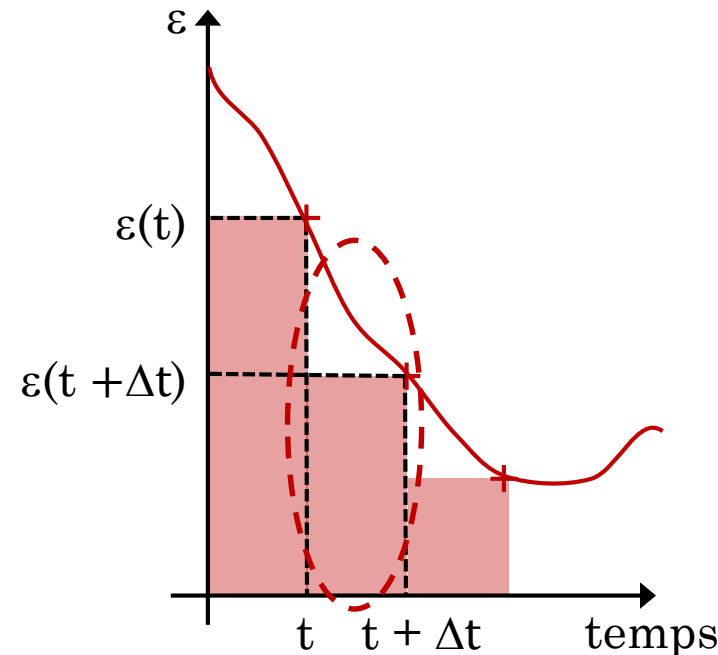
□ Intégral

- Le terme intégral revient au calcul d'une surface et en numérique nous pouvons considérer 2 approches (il faut en plus ajouter l'asservissement Proportionnel à U) :

- ✓ Méthode des rectangles

$$U(t + \Delta t) = k_i. [\Sigma(t) + \varepsilon(t + \Delta t). \Delta t]$$

$$\underbrace{\hspace{10em}}_{= \Sigma(t + \Delta t)}$$



1.5. Mise en équation

□ Intégral

- Le terme intégral revient au calcul d'une surface et en numérique nous pouvons considérer 2 approches (il faut en plus ajouter l'asservissement Proportionnel à U) :

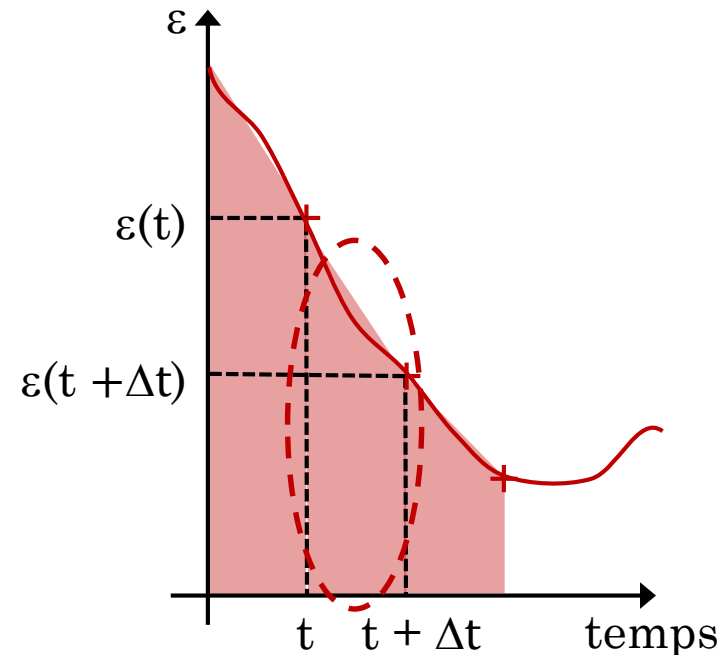
- ✓ Méthode des rectangles

$$U(t + \Delta t) = k_i \cdot \underbrace{[\Sigma(t) + \varepsilon(t + \Delta t) \cdot \Delta t]}_{= \Sigma(t + \Delta t)}$$

- ✓ Méthode des trapèzes

$$U(t + \Delta t) = k_i \cdot \underbrace{\left[\Sigma(t) + \frac{\varepsilon(t) + \varepsilon(t + \Delta t)}{2} \cdot \Delta t \right]}_{= \Sigma(t + \Delta t)}$$

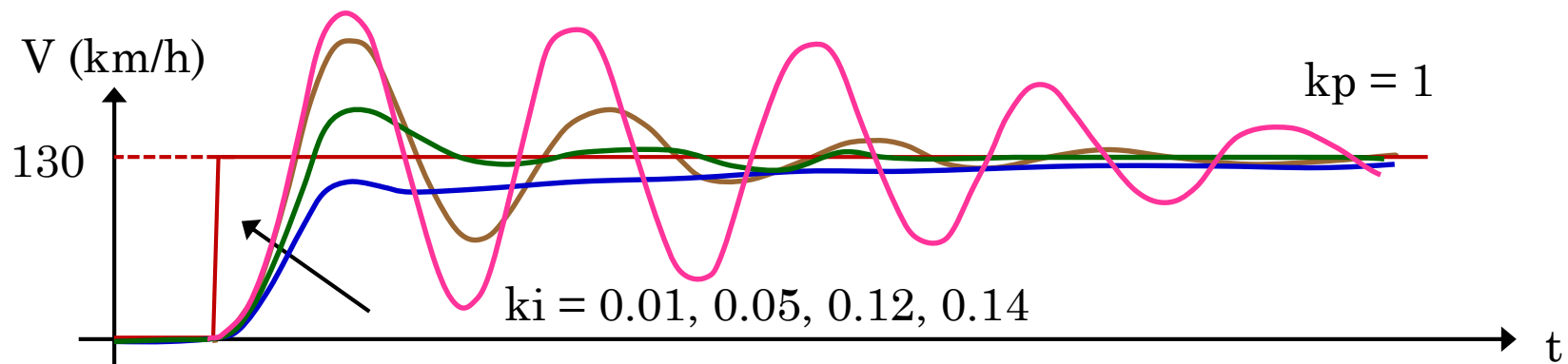
- ki (en s⁻¹) est le gain



1.5. Mise en équation

□ Intégral

- La méthode des trapèzes est plus précise mais elle demande plus de calcul et la mémorisation de l'erreur de la boucle précédente
- Pour réduire les calculs, on peut mesurer l'erreur de façon régulière. Dans ce cas Δt peut être intégré au gain k_i qui devient sans dimension
- Avec le PI, l'erreur statique est presque éliminée. Si on augmente k_i , la consigne est atteinte plus rapidement mais les oscillations sont de plus en plus grandes.



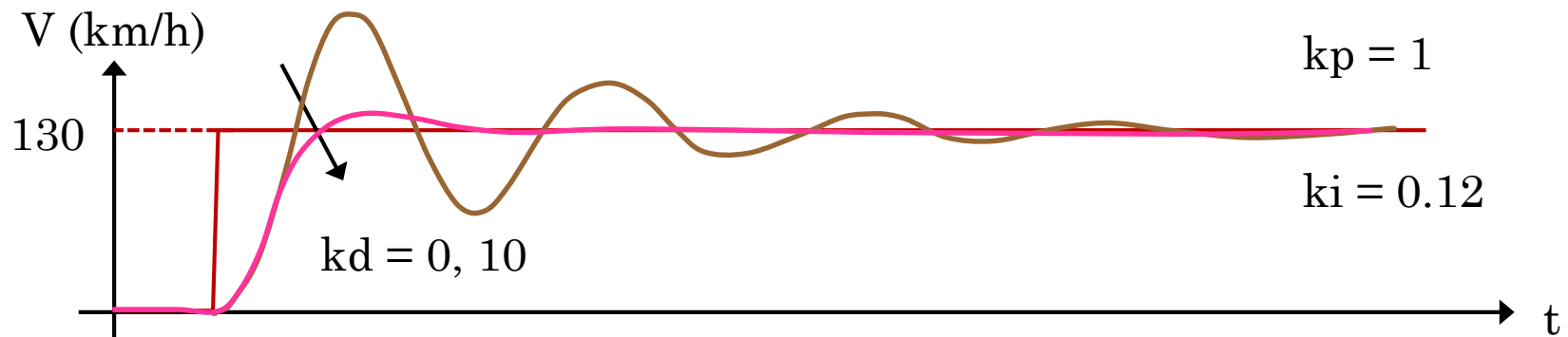
1.5. Mise en équation

□ Dérivé

- On réalise une dérivée numérique avec l'erreur ε (il faut en plus ajouter les asservissements Proportionnel et Intégral à U) :

$$U(t + \Delta t) = k_d \cdot \left[\frac{\varepsilon(t + \Delta t) - \varepsilon(t)}{\Delta t} \right]$$

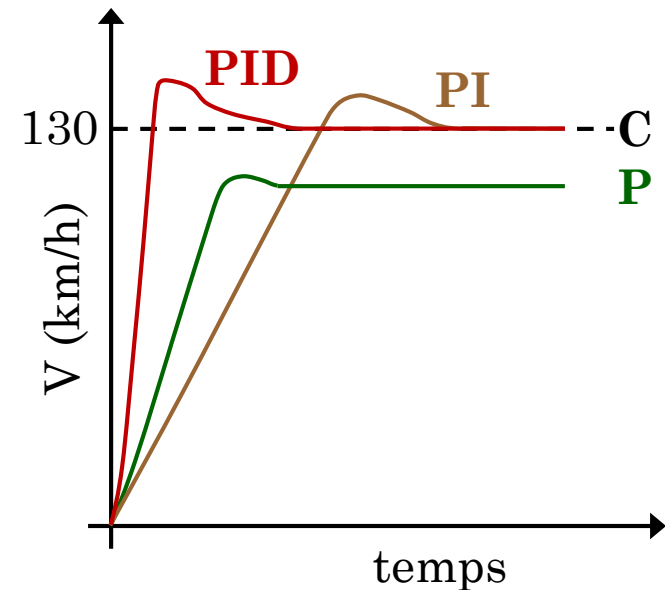
- k_d est le gain (en s). Si Δt est constant, on peut l'intégrer dans k_d
- L'ajout de l'asservissement Dérivé réduit les oscillations et le dépassement



1.6. En résumé

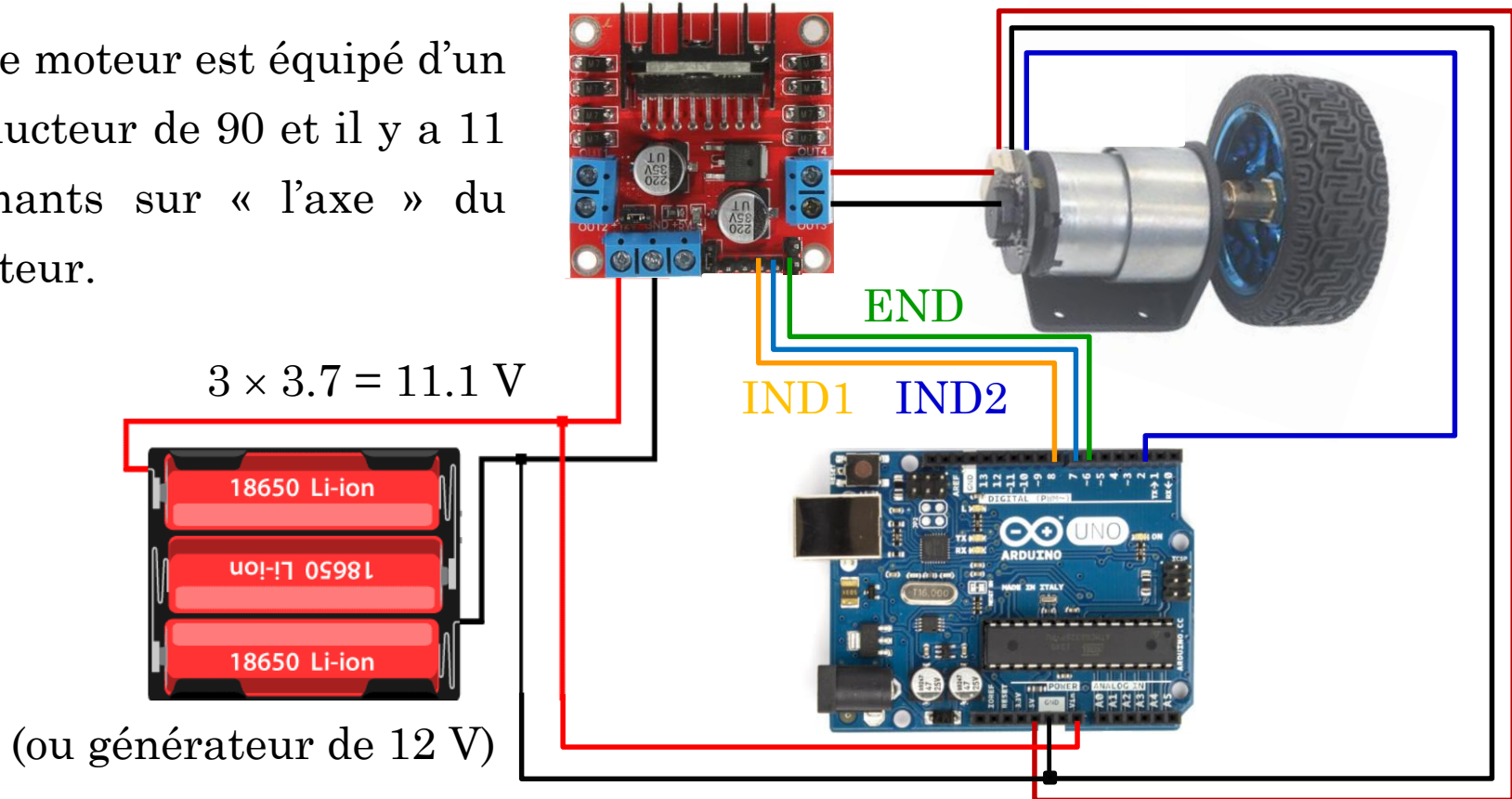
- Proportionnel
 - ✓ Corrige les effets d'une perturbation,
 - ✓ Déstabilise le système quand on augmente le gain,
 - ✓ Mais elle n'annule pas l'erreur.
- Intégral
 - ✓ Corrige les effets d'une perturbation,
 - ✓ Annule l'erreur statique,
 - ✓ Introduit un dépassement,
 - ✓ Mais elle n'est pas très rapide.
- Dérivé
 - ✓ Accélère la correction,
 - ✓ Stabilise plus rapidement le système,
 - ✓ Mais n'annule pas l'erreur statique et est sensible aux vibrations.

	Précision	Stabilité	Rapidité
P	↗	↘	↗
I	↗	↘	↘
D	↘	↗	↗



2.1. Description du montage

- Le montage est tiré du cours « Eléments de robotique avec arduino : Moteurs »
- Le moteur est équipé d'une roue codeuse avec capteur à effet Hall.
- Le moteur est équipé d'un réducteur de 90 et il y a 11 aimants sur « l'axe » du moteur.



2.2. Comptage des tours de la roue

- Il faut se référer à la partie « encodeurs » du cours « Eléments de robotique avec arduino : Moteurs »
- Une des deux sorties de l'encodeur est connectée à l'IO n° 2 de la carte ce qui permet d'utiliser l'interruption INT0.
- La période du Timer1 est réglée à 20 ms : temps ou bout duquel on détermine le nombre de tours effectués par la roue
- Pour 1 tour de roue, le compteur donne le chiffre $90 \times 11 = 990$
- Ici, on ne s'intéressera pas à la vitesse de la roue (m/s) mais au nombre de tour (ou rotation) par second (tr/s).
- Pour un PWM de 200 sur le END (entrée enable du L298), le compteur indique 1664 ce qui signifie que la vitesse de la roue est :

$$(28 / 990) / 0.02 = 1.41 \text{ tr/s}$$

- A noter qu'on obtient au maximum 1.67 tr/s pour un PWM de 255.

2.2. Comptage des tours de la roue

- Le programme avant la mise en place de l'asservissement :

```
#include <TimerOne.h>
const int END = 6, IND1 = 8, IND2 = 7;
int PWMG = 0, PWMD = 0;
unsigned int compteurD = 0;
unsigned long int deltatemps = 20;
float const Mult= (1/990.0)/(deltatemps*0.001);
float RPMD;
void setup(){
  Timer1.initialize(deltatemps*1000);
  Timer1.attachInterrupt(CalculD);
  Serial.begin(115200);
  pinMode(END, OUTPUT);
  pinMode(IND1, OUTPUT);
  pinMode(IND2, OUTPUT);
  analogWrite(END, 0);
  attachInterrupt(0, changeD, RISING);
  delay(500); }
void loop(){ }

//*****Determination de la vitesse en tr/s *****
void CalculD(){
  RPMD = Mult*compteurD;
  Serial.print(millis());
  Serial.print(" ");
  Serial.println(RPMD,3);
  compteurD = 0; }
//*****Interruptions*****
void changeD() {
  compteurD++; }
//****Fonctions pour le moteur droit*****
void moteurD(int vit, byte sens) {
  if (sens == 0){
    digitalWrite(IND1,LOW);
    digitalWrite(IND2,HIGH);
    analogWrite(END, vit);}
  else {
    digitalWrite(IND1,HIGH);
    digitalWrite(IND2,LOW);
    analogWrite(END, vit);} }
```

2.3. Asservissement Proportionnel

- On commence par définir plusieurs variables :

```
const float consigneD = 1.4; // Vitesse souhaitée pour la roue en tr/s
```

```
float erreurD = 0.0; // difference entre la consigne et la mesure
```

```
const int kp = 100; // coefficient de proportionnalité
```

- La partie du code qui réalise l'asservissement proportionnel est :

```
erreurD = consigneD - RPMD; // calcul de l'erreur
```

```
PWMD = kp*erreurD; // determination de la valeur du PWM
```

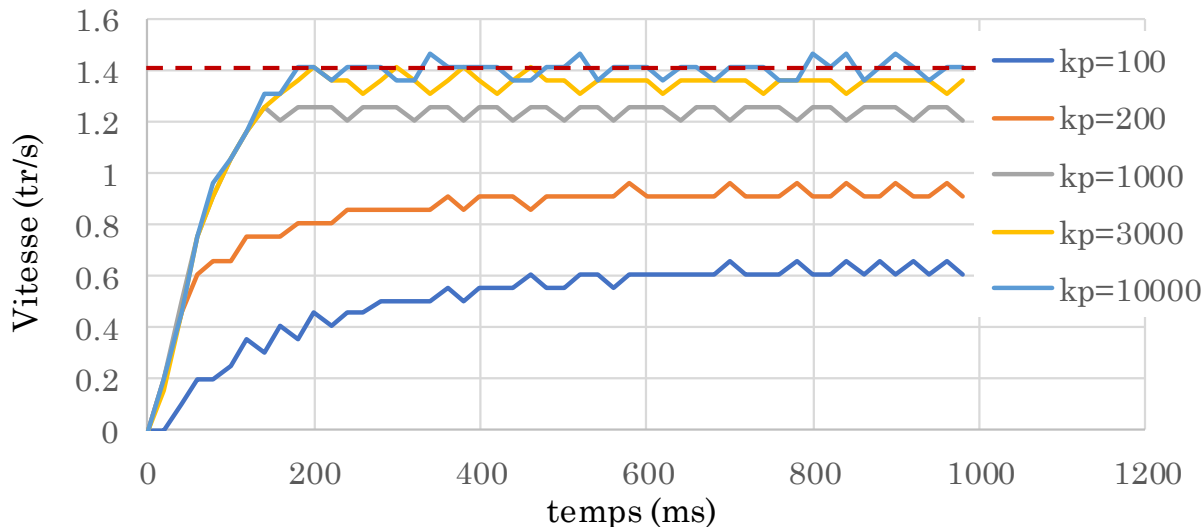
```
if (PWMD < 0) {PWMD = 0;} // bornage du PWM dans l'intervalle [0, 255]
```

```
else if (PWMD > 255) {PWMD = 255;}
```

```
moteurD(PWMD, 1); // application du nouveau signal PWM
```

2.3. Asservissement Proportionnel

- On peut alors tester l'impact de la valeur de k_p sur le respect de la consigne.
- Plus k_p est grand, plus on se rapproche de la consigne de 1.4 tr/s.
- On remarquera des variations de la vitesse sur les plateaux.



2.3. Asservissement Proportionnel + Intégral

- On définit à présent le paramètre k_i et la somme des erreurs :

```
const float ki = 0.2; // coefficient de l'intégrale
```

```
float somme_erreurD = 0.0; // somme des erreurs
```

- La partie du code qui réalise l'asservissement PI est :

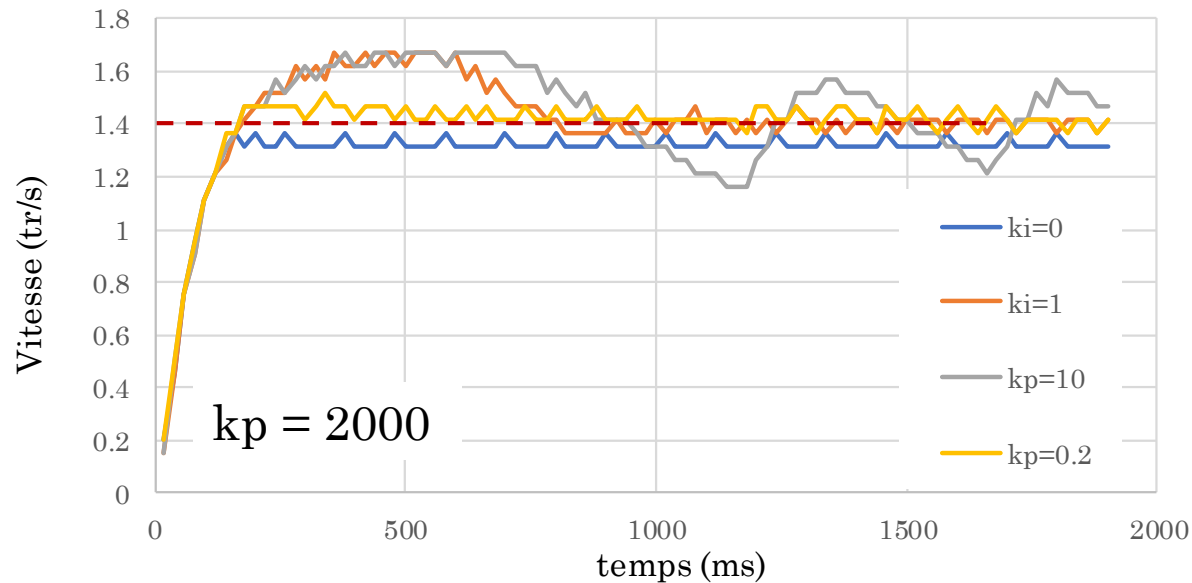
```
erreurD = consigneD - RPMD;
```

```
somme_erreurD = somme_erreurD + erreur;
```

```
PWMD = kp*erreurD + ki* somme_erreurD;
```

2.3. Asservissement Proportionnel + Intégral

- L'asservissement Intégral permet de rejoindre la consigne.
- Une valeur de k_i trop importante induit un « grand » dépassement et même des oscillations.



2.3. Asservissement Proportionnel + Intégral + Dérivé

- On définit à présent le paramètre kd et l'erreur précédente :

```
const float kd = 10; // coefficient de la dérivée
```

```
float erreurD_avant = 0.0; // erreur précédente
```

```
float delta_erreurD = 0.0; // variation de l'erreur
```

- La partie du code qui réalise l'asservissement PID est :

```
erreurD = consigneD - RPMD;
```

```
somme_erreurD = somme_erreurD + erreur;
```

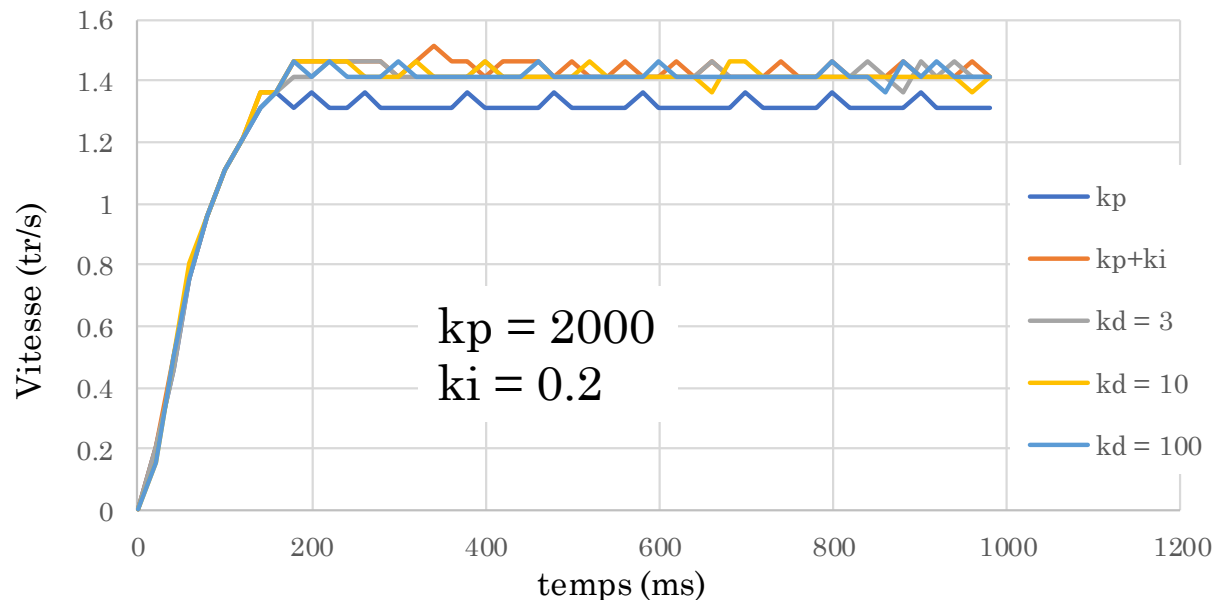
```
delta_erreurD = erreurD - erreurD_avant;
```

```
erreurD_avant = erreurD;
```

```
PWMD = (kp*erreurD) + (ki*somme_erreurD) + (kd*delta_erreurD);
```

2.3. Asservissement Proportionnel + Intégral + Dérivé

- Dans le cas de cet exemple, le gain apporté par la dérivée n'est pas flagrant
- La difficulté pour l'asservissement PID est de trouver les bons coefficients
- Une fois les coefficients réglés, on peut ralentir la roue (en posant la main dessus par exemple) et constater que le moteur va forcer pour conserver la vitesse de la consigne.



3.1. Présentation

- Un self balancing robot est un robot en équilibre sur 2 roues
- Un accéléromètre permet de connaître l'inclinaison du robot et de l'annuler en actionnant les roues
- On peut aussi modifier l'inclinaison finale du robot pour le faire avancer ou reculer
- Ce fonctionnement a été popularisé par le Segway



3.2. Le montage

