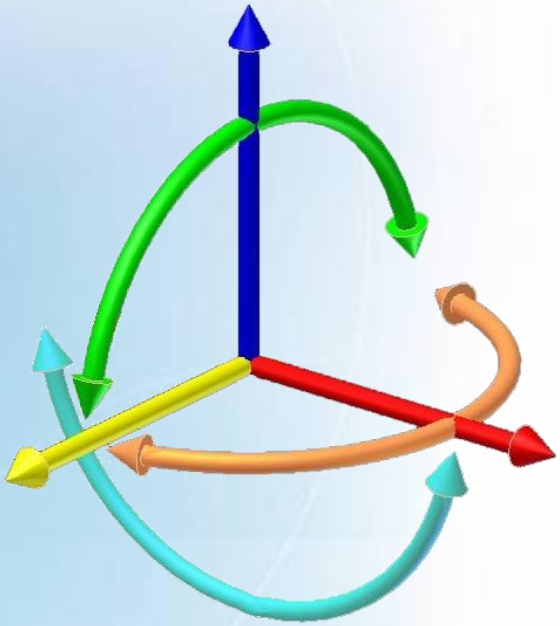
CAPTEURS

REGULATION
PID

MOTEURS
BRUSHLESS

CAPTEURS

- Les capteurs présents sur le système sont :
  - 1 **Accéléromètre** 3 axes (X, Y et Z)
  - 1 **Gyromètre** 3 axes (X, Y et Z)
- Ces capteurs sont identiques à ceux que l'on peut retrouver dans les smartphones.
- En réalité, ces capteurs sont (dons notre cas) présents dans un même composant nommé « **centrale inertielle ».**
- La centrale inertielle utilisée porte la référence MPU6050 de chez IvenSense communiquant sur un **bus I2C**.
- Afin d'obtenir une position angulaire, il est nécessaire d'effectuer des calculs à partir des informations capteurs.



BUS I2C

- Le protocole I²C est basé sur un bus série synchrone half duplex à 2 fils :
  - une ligne de données SDA (Serial DAta) bidirectionnelle fait transiter les bits entre le maître du bus et les esclaves (max 128)
  - une ligne SCK (Serial ClocK) qui cadence la transmission série.
- Procédure du protocole I2C :

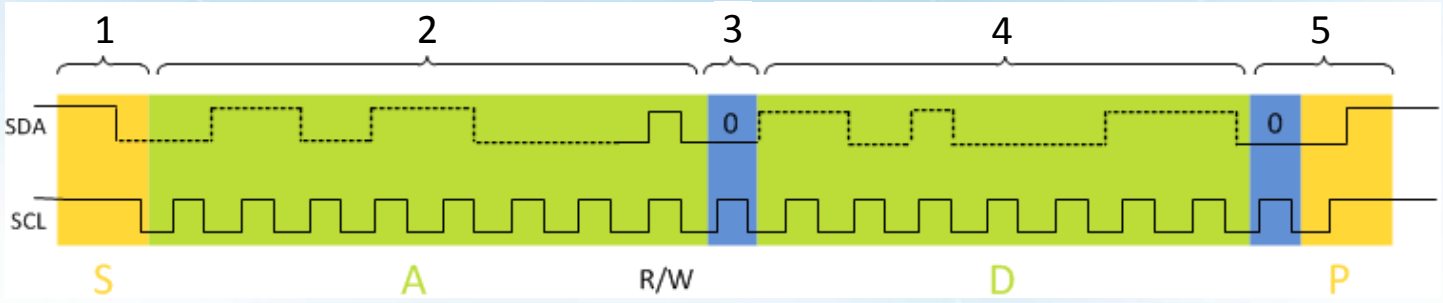
1 – Bit de Start en appliquant un passage de 1 à 0 de SDA alors que SCL est à l'état 1

2 – Adresse du composant sur 7 bits et un bit de lecture /écriture (0 pour écrire 1 pour lire)

3 – Bit ACK permet de signaler que la transmission du 1er octet (adresse) s'est bien effectuée

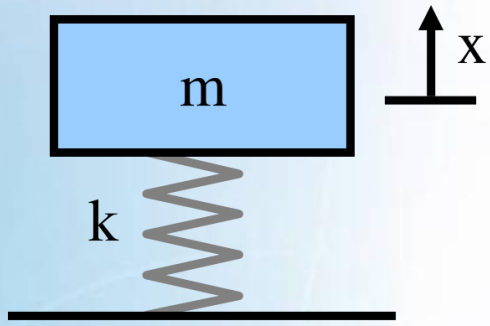
4 – Données à transmettre (avec un ACK pour valider la bonne transmission)

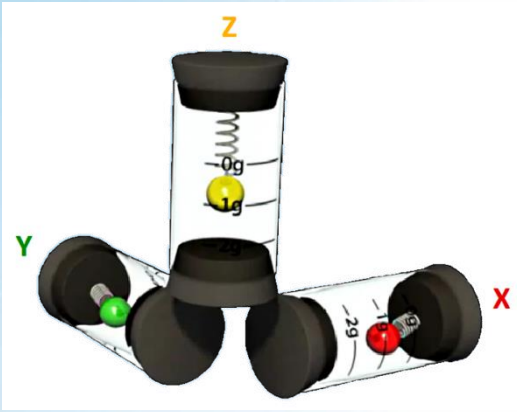
5 – Bit de Stop en appliquant un passage de 0 à 1 de SDA alors que SCL reste à l'état 1



ACCELEROMETRE

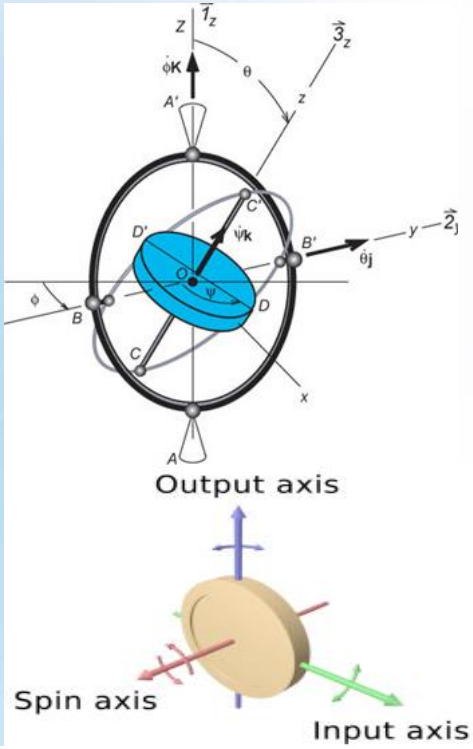
- Un accéléromètre mesure une accélération linéaire.
- Un accéléromètre peut être schématisé par un système masse + ressort. Considérons le schéma ci-contre : à l'équilibre, la position x de la masse m sera la référence, donc x=0. Si le support subit une accélération verticale, vers le haut, deux choses vont avoir lieu : ce support va se déplacer vers le haut d'une part et, à cause de l'inertie de la masse m, celle-ci va avoir tendance à rester à sa position de départ, forçant le ressort à se comprimer. La valeur x sera d'autant plus grande que l'accélération appliquée au support sera importante.

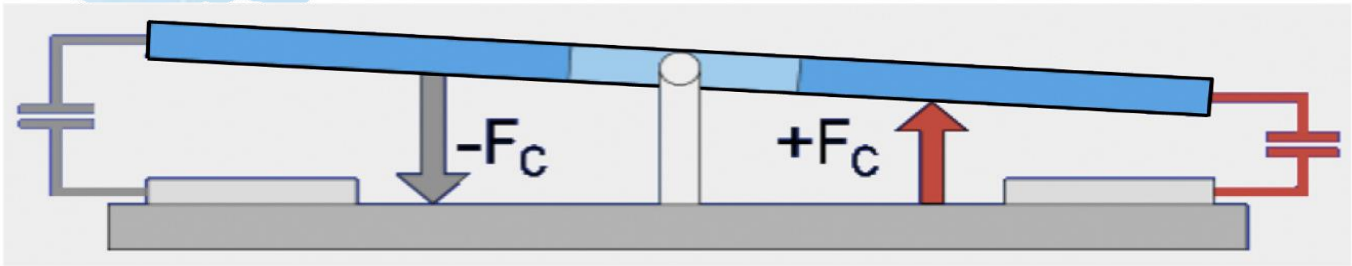




GYROMETRE

- Un gyromètre mesure une vitesse angulaire.
- Son principe de fonctionnement est basé sur le phénomène de Coriolis
- Le composant sous l'effet de la rotation va se déformer suivant ses axes respectifs comme indiqués ci-contre. L'élasticité joue un rôle important puisqu'elle fait apparaître des espaces utilisés en tant que condensateurs différentiels.

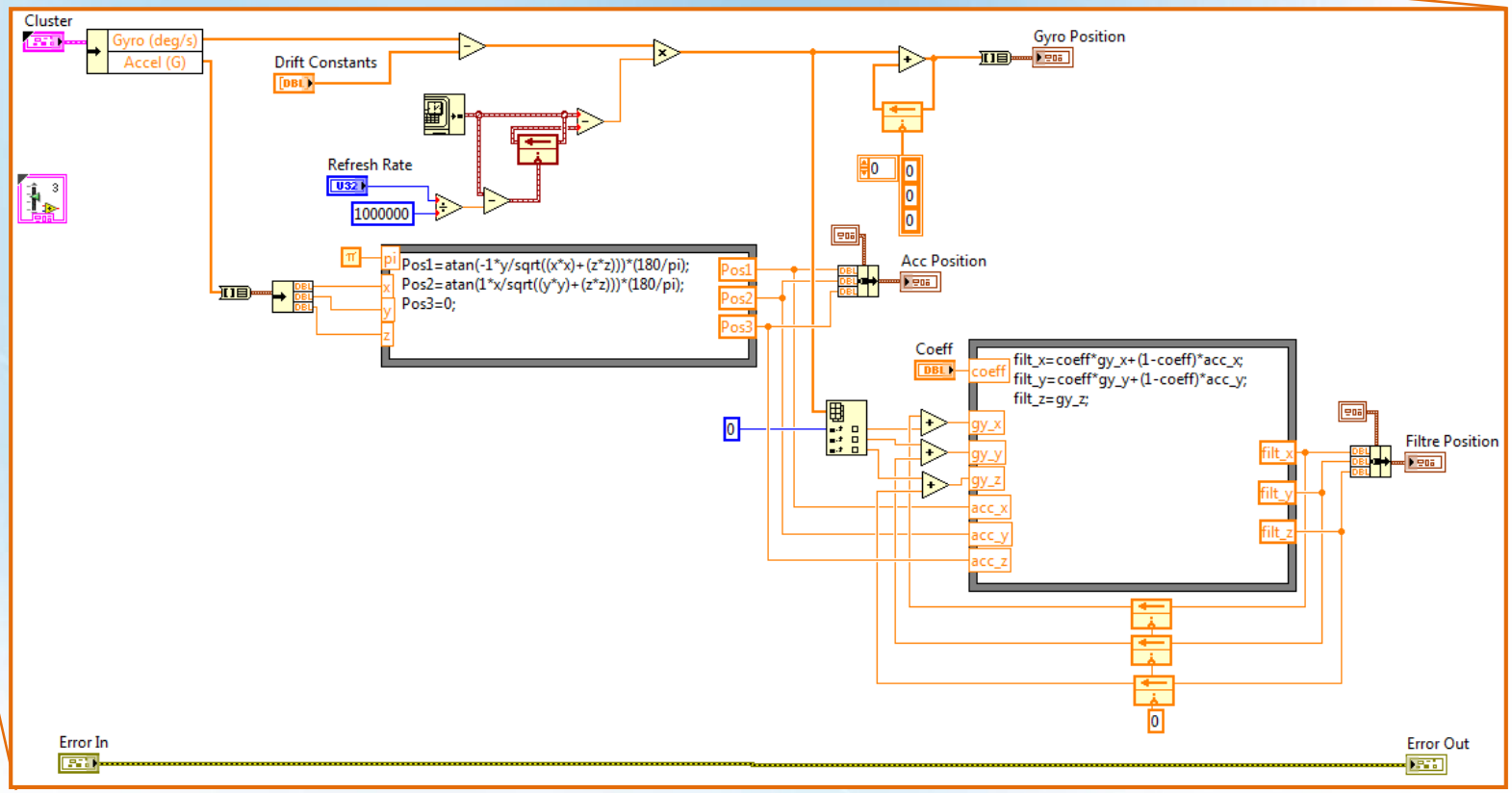




CALCULS

- La centrale inertielle est le capteur qui nous permettra d'en déduire une postions angulaire or :
  - L'accéléromètre mesure une accélération linéaire
  - Le gyromètre mesure une vitesse angulaire
- Il faut donc croiser les informations de ces deux capteurs afin d'obtenir une information de position angulaire (exemple avec le filtre complémentaire utilisé dans notre application).
  - **angle = (coeff)*(angle + gyro*dt) +(1-coeff)*(x_acc)**
  - La variable coeff permet d'adapter la confiance à donner à l'accéléromètre / au gyromètre car l'acceleromètre donne une information vrai dans en période stable alors que le gyro nous donne une mesure valide pendant le mouvement.
- Il existe aussi d'autre types de filtres ou de calculs plus complexes permettant d'obtenir une mesure encore plus précise, comme par exemples les Quaternions.

CALCULS


Cluster
Gyro (deg/s)
Accel (G)
Drift Constants
Refresh Rate
1000000
Gyro Position
Acc Position
Pos1=atan(-1*y/sqrt((x*x)+(z*z)))*(180/pi);
Pos2=atan(1*x/sqrt((y*y)+(z*z)))*(180/pi);
Pos3=0;
Coeff
filt_x=coeff*gy_x+(1-coeff)*acc_x;
filt_y=coeff*gy_y+(1-coeff)*acc_y;
filt_z=gy_z;
Filtre Position
Error In
Error Out

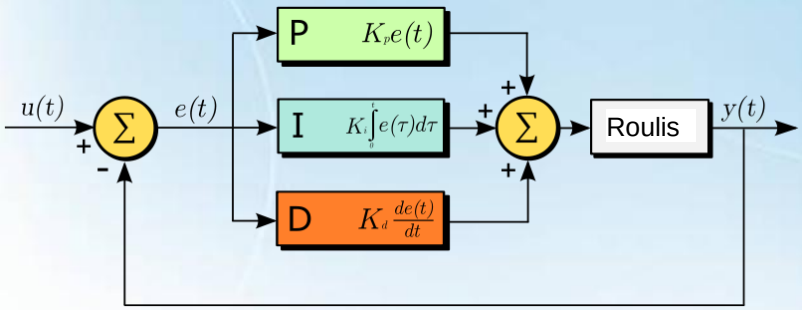
CALCULS

nacelle_arduino | BLcontroller.h | EEPROMAnything.h | I2Cdev.cpp | I2Cdev.h | IMU | MPU6050.cpp | MPU6050.h | SerialCom.h | SerialCommand.cpp | SerialCommand.h | Timer1.h

```
void updateACCAttitude(){
  uint8_t axis;

  // 80 us
  // Apply complimentary filter (Gyro drift correction)
  // If accel magnitude >1.4G or <0.6G and ACC vector outside of the limit range => we neutralize the effect of accelerometers in the angle estimation.
  // To do that, we just skip filter, as EstV already rotated by Gyro
  if (( 36 < accMag && accMag < 196 ) || disableAccGtest) {
    for (axis = 0; axis < 3; axis++) {
      //utilLP_float(&EstG.A[axis], accLPF[axis], AccComplFilterConst);
      EstG.A[axis] = EstG.A[axis] * (1.0 - AccComplFilterConst) + accLPF[axis] * AccComplFilterConst; // note: this is different from MultiWii (wrong brackets postion in
    }
  }
}


void getAttiduteAngles() {

  // attitude of the estimated vector
  // 200us
  angle[ROLL]  = angleOffsetRoll +  Rajan_FastArcTan2_deg1000(EstG.V.X , sqrt(EstG.V.Z*EstG.V.Z+EstG.V.Y*EstG.V.Y));
  // 142 us
  angle[PITCH] = angleOffsetPitch + Rajan_FastArcTan2_deg1000(EstG.V.Y , EstG.V.Z);

}
```

REGULATION PID

- Pour réagir correctement la nacelle doit être :
  - Correctement asservie (réponse du système à un changement de consigne)
  - Correctement régulée (réponse du système suite à une perturbation externe)
- Sur ce système nous utilisons 2 boucles de régulations car il y a 2 axes à piloter (tangage et roulis)
- La valeur de sortie permettant de piloter le moteur est calculée en fonction de l'erreur (écart entre la mesure et la consigne) et des paramètres P, I et D.

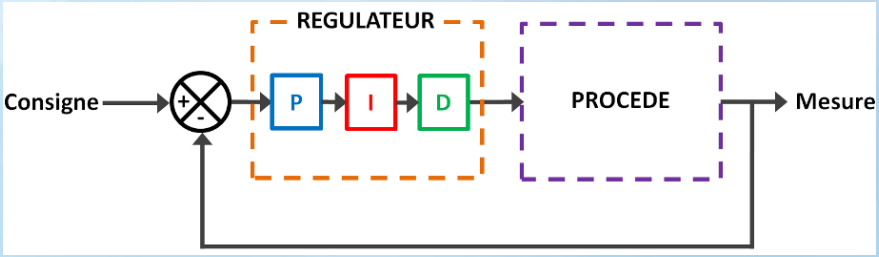


	Précision	Stabilité	Rapidité
P	↗	↘	↗
I	↗	↘	↘
D	↘	↗	↗

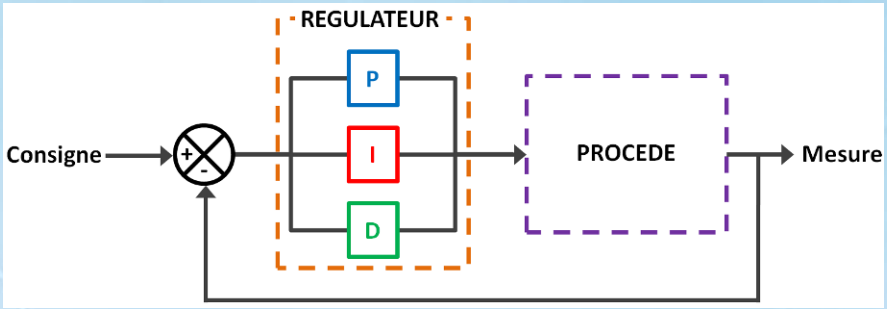
Type de régulation
P – Le gain
I – L'intégrale
D – La dérivé

TYPE DE REGULATION

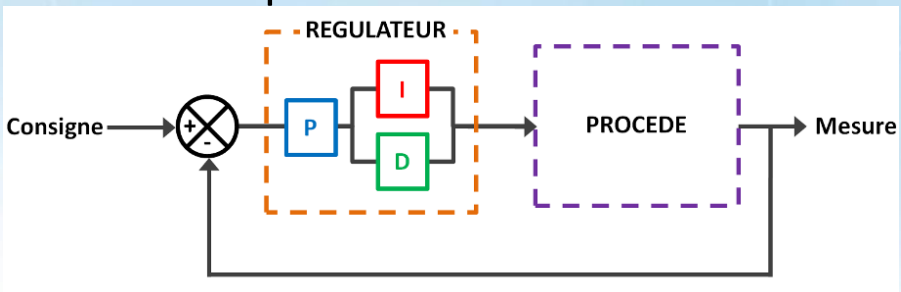
- PID série :



- PID parallèle :



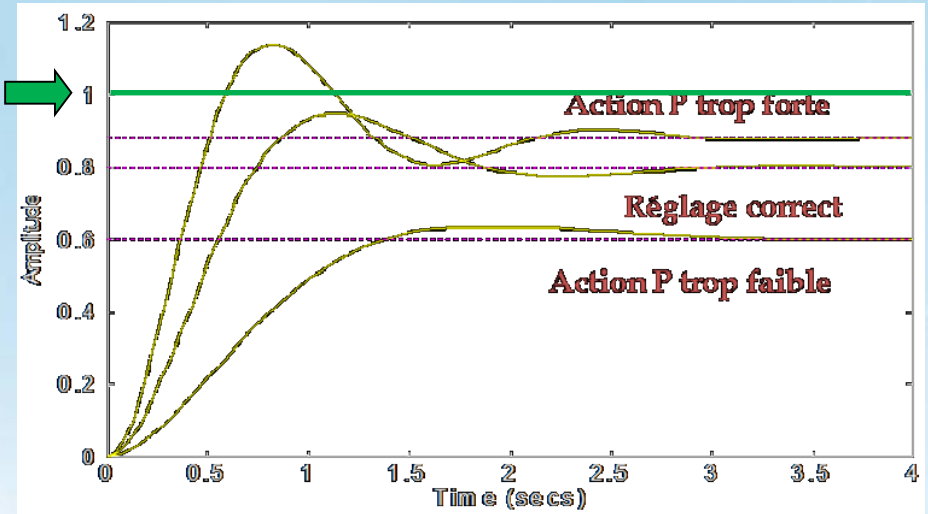
- PID mixte ou académique :



P – LE GAIN

- Le rôle de l'action proportionnelle est d'accélérer la réponse de la mesure, ce qui a pour conséquence de réduire l'écart entre la mesure et la consigne.
- Si lors d'un changement de consigne avoir un P plus grand permet de réagir plus vite, en régime stationnaire un P grand rend la sortie du régulateur moins stable.
- Lors d'une perturbation, la mesure s'écarte de la consigne, la régulation proportionnelle tend à la ramener tout en laissant subsister un écart résiduel lorsque le régime permanent est atteint.

Consigne



Test en asservissement sur un procédé instable

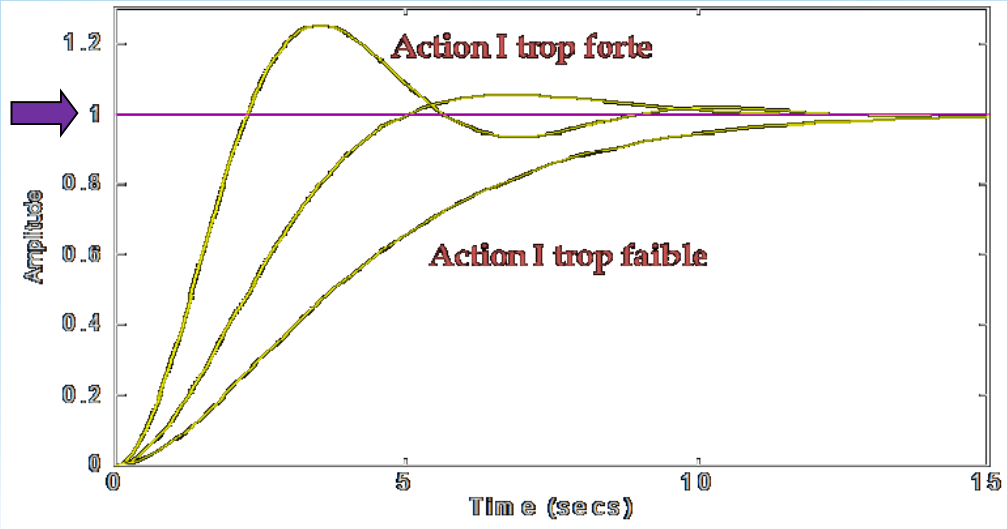
I - L'INTEGRALE

- Le rôle de l'action intégrale est d'annuler l'écart entre la mesure et la consigne. Le signal de sortie du régulateur en intégrateur seul est proportionnel à l'intégrale de l'écart :

  mesure - consigne.

- **L'action intégrale est généralement associée à l'action proportionnelle.**
- Comme dans le cas de l'action proportionnelle, une augmentation excessive de l'action intégrale (diminution de Ti) peut être source d'instabilité.

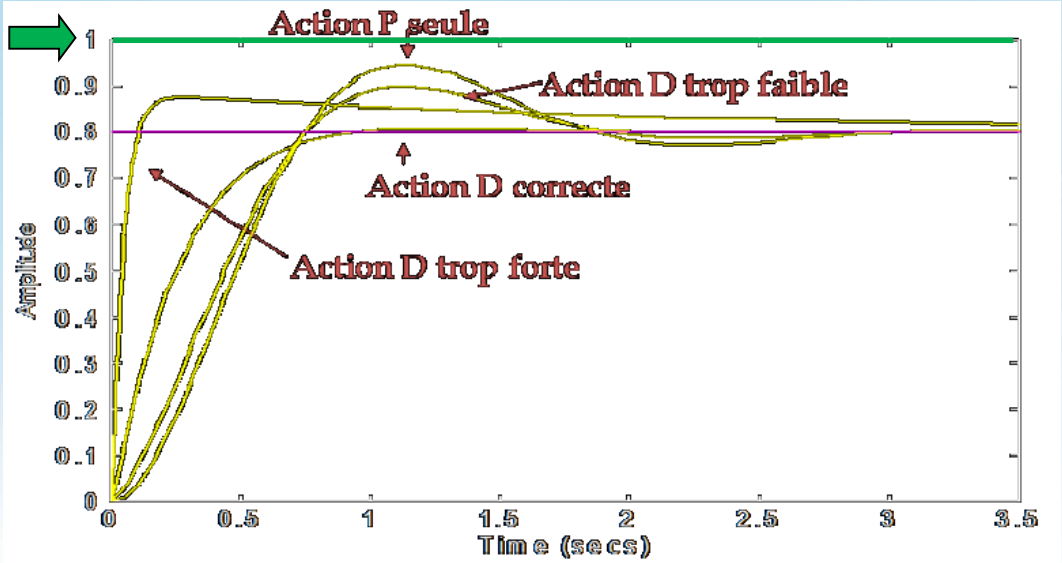
Consigne



D – LA DERIVÉE

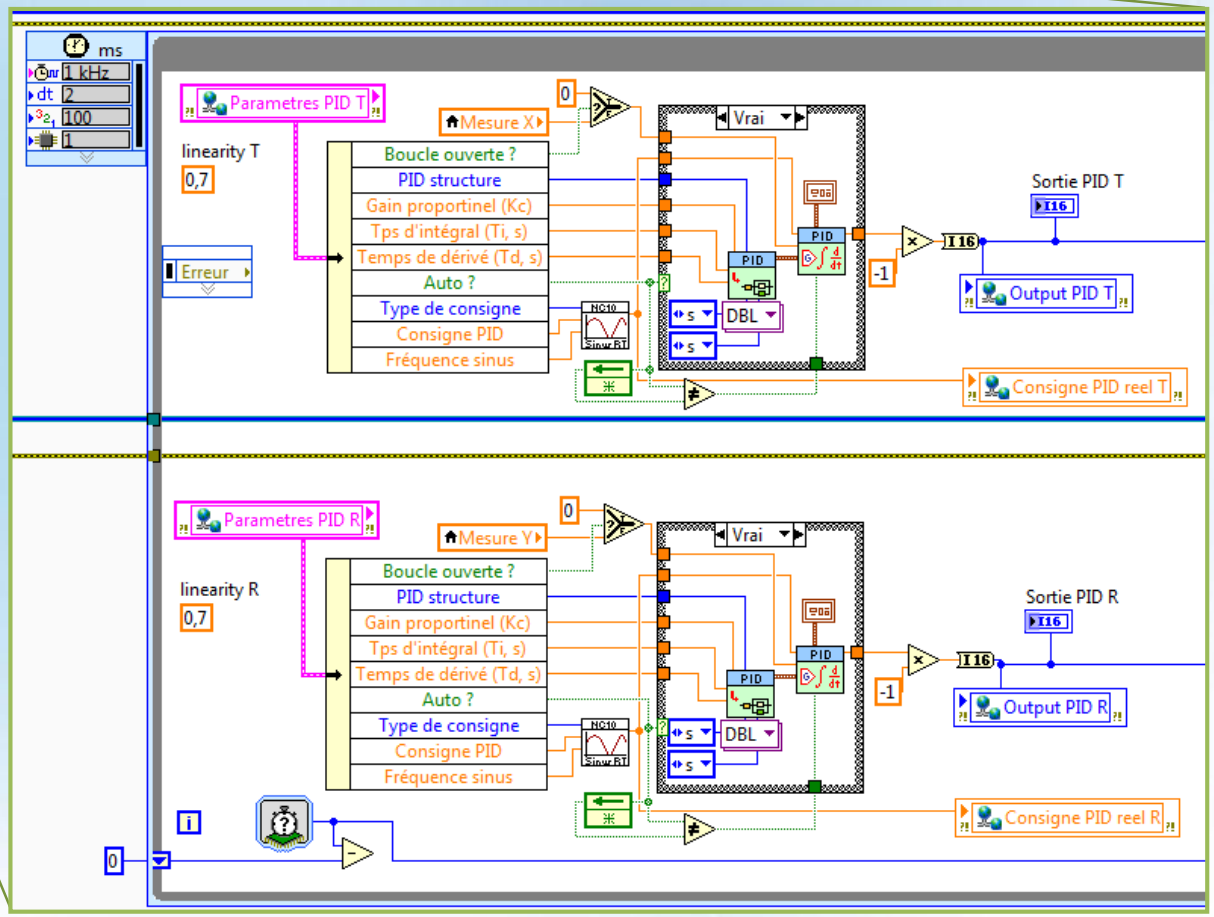
- Le rôle de l'action dérivée est de compenser les effets du temps mort (retard) du procédé. Elle a un effet stabilisateur mais une valeur excessive peut entrainer l'instabilité.
- Son rôle est identique quelle que soit la nature du procédé. La sortie du dérivateur est proportionnelle à la vitesse de variation de l'écart.
- **Notons que l'action dérivée ne peut pas être utilisée seule.**
- Dans le cas d'un signal de mesure bruité, la dérivée amplifie le bruit rendant son utilisation délicate ou impossible.

Consigne



REGULATION PID

RT Pilotage Nacelle NC00.vi



Type de régulation

P – Le gain

I – L'intégrale

D – La dérivé

REGULATION PID

nacelle_arduino.ino

nacelle_arduino | BLcontroller.h | EEPROMAnything.h | I2Cdev.cpp | I2Cdev.h | IMU | MPU6050.cpp | MPU6050.h | SerialCom.h | SerialCommand.cpp | SerialComm

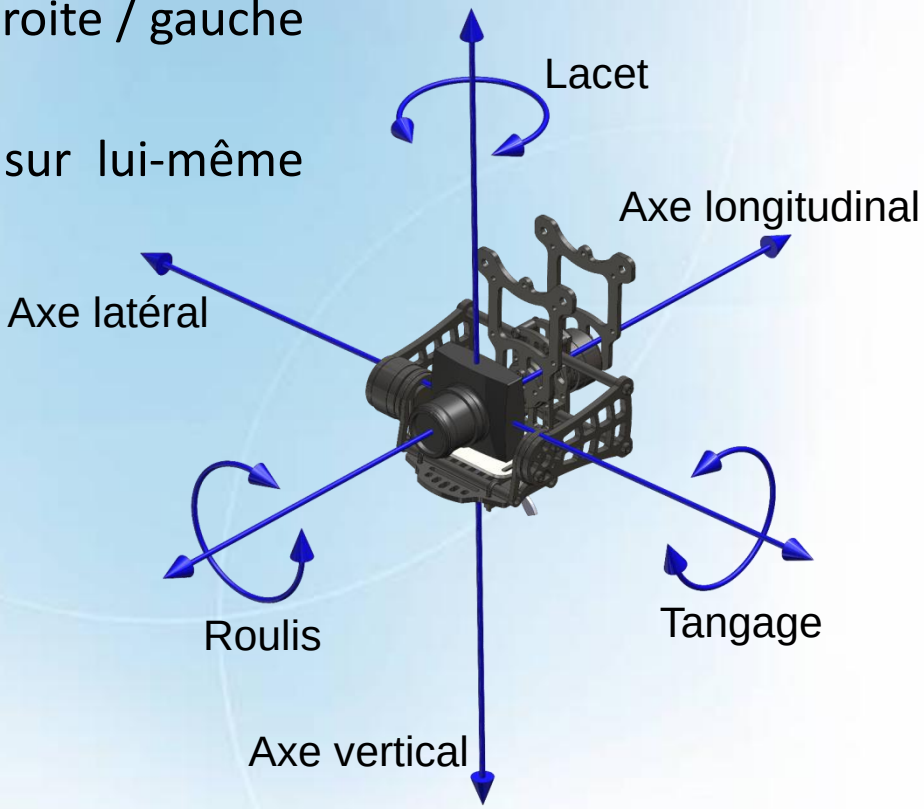
```
/************************/
/* PID Controller       */
/************************/
// PID integer inplementation
//   DTms  ... sample period (ms)
//   DTinv ... sample frequency (Hz), inverse of DT (just to avoid division)
int32_t ComputePID(int32_t DTms, int32_t DTinv, int32_t in, int32_t setPoint, int32_t *errorSum, int32_t *errorOld, int32_t Kp, int16_t Ki, int32_t Kd)
{
  int32_t error = setPoint - in;
  int32_t Ierr;

  Ierr = error * Ki * DTms;
  Ierr = constrain_int32(Ierr, -(int32_t)1000*100, (int32_t)1000*100);
  *errorSum += Ierr;

  /*Compute PID Output*/
  int32_t out = (Kp * error) + *errorSum + Kd * (error - *errorOld) * DTinv;
  *errorOld = error;

  out = out / 4096 / 8;

  return out;
}

/*********************************************/
/* Main Loop                                 */
/*********************************************/
```

MOTEURS BRUSHLESS

- Les moteur présents sur le système sont des moteurs brushless outrunner (le rotor est autour du stator)
- 7 paires de pôles
- Pas de capteur à effet Hall.
- Dans notre cas nous utilisons ces moteurs pour tourner à basse vitesse et pour garder une position fixe. C'est une utilisation peu conventionnelle car il est plus commun d'utiliser les moteurs brushless à grande vitesse.

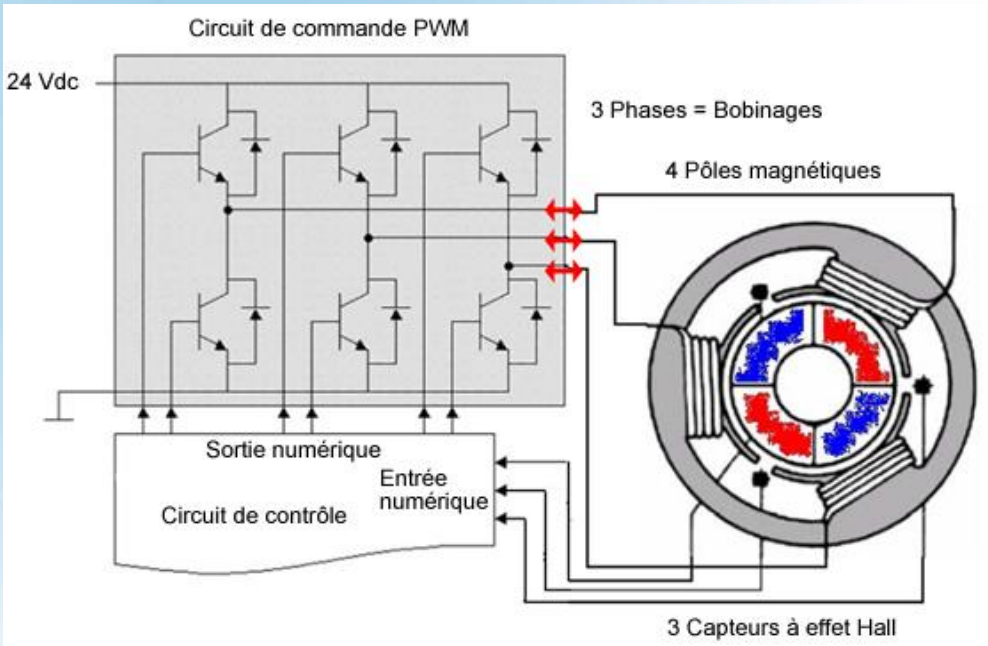
TANGAGE / ROULIS / LACET

- Tangage : mouvement d'inclinaison haut / bas (rotation sur l'axe latéral)
- Roulis : mouvement d'inclinaison droite / gauche (rotation sur l'axe longitudinal)
- Lacet : mouvement de rotation sur lui-même (rotation sur l'axe vertical)



MOTEURS BRUSHLESS

- Un « moteur brushless » ou « moteur sans balais » ou « machine synchrone auto pilotée à aimants permanents », est une machine électrique de la catégorie des machines synchrones dont le rotor est constitué d'un ou plusieurs aimants permanents et pourvu d'origine d'un capteur de position rotorique (capteur à effet Hall).
- En appliquant une modulation de largeur d'impulsion (PWM), le circuit de commande peut faire varier la tension moyenne envoyée au moteur pour contrôler sa vitesse.
- 3 capteurs à effet Hall, intégrés au stator mesurent la position angulaire du rotor. Chaque fois que les pôles magnétiques du rotor passent près des capteurs à effet Hall, ils émettent un signal haut ou bas, indiquant qu'ils passent le pôle Nord ou Sud. Ces signaux permettent au circuit de commande de discerner le meilleur moment pour inverser les impulsions sur les 3 bobines.



MOTEURS BRUSHLESS NACELLE

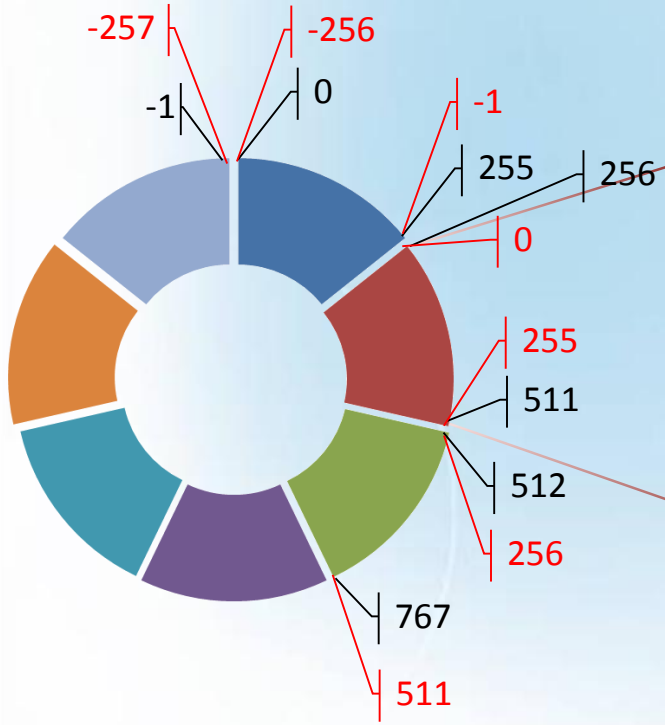
- Les moteurs brushless utilisés sur la nacelle ont 7 paires de pôles :
  - Il y a donc 7 positions par tour => 1 pas = une rotation de 51,43°
  - Il est donc impossible de faire de la régulation en position angulaire avec une telle précision.
  - Il est nécessaire de créer des points intermédiaires à l'instar des moteurs pas à pas (« microstepping »)

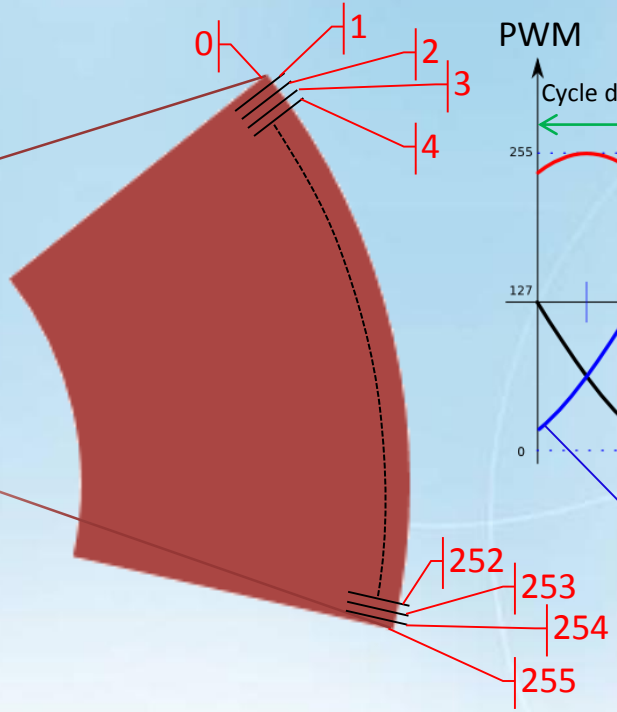
En savoir plus sur le pilotage effectué

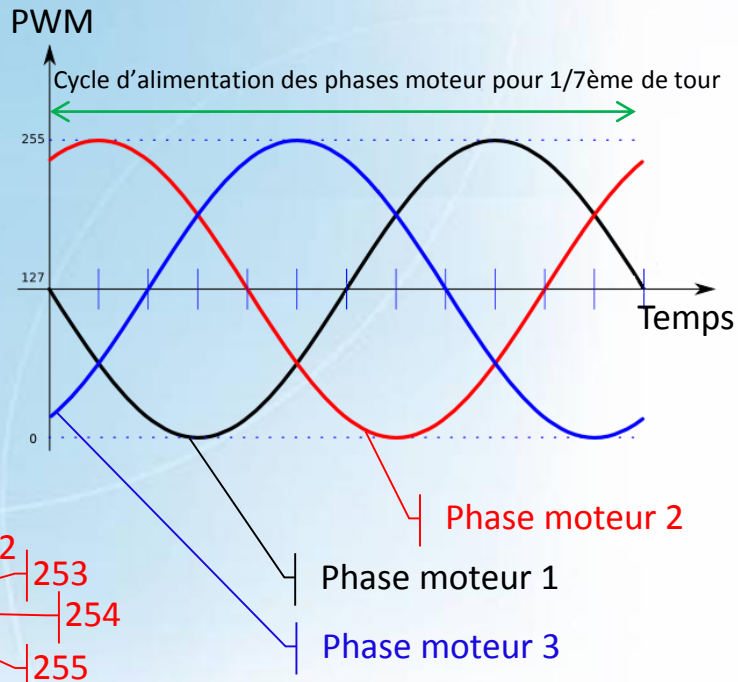
- Les moteurs brushless utilisés sur la nacelle n'ont pas de capteur à effet Hall :
  - Il est possible d'utiliser la force contre électromotrice (FCEM) pour commuter les phases, mais dans notre cas nous ne pouvons pas utiliser cette méthode car le moteur se trouve sur une position fixe la majorité du temps.
  - Le pilotage moteur est donc fait en boucle ouverte.

MOTEURS BRUSHLESS NACELLE

- En pilotant le moteur en microstepping, nous allons ajouter 256 positions entre 2 pôles. Donc 7x256= 1792 pts / tour moteur => 360° / 1792 = 0.20 ° On obtiendra alors une précision de 0,2°.
- Chaque position de 0 à 255 correspond à une valeur de PWM comprise entre 0 et 255 (exemple pour la position 0 on transmet **127 phase 1** puis **127+85 = 212 phase 2** (décalage de 120°) puis **127+85+85 = 297 => 297-256 = 41 phase 3** (décalage de 120°)







PILOTAGE

- Le pilotage des moteurs brushless est réalisé en trois parties :
  - 1ère partie : Génération d'une sinusoïde et stockage des points constituants cette sinusoïde dans un tableau

En savoir plus sur la 1er partie

  - 2ème partie : Création des 3 phases moteur à partir de la valeur de sortie du PID

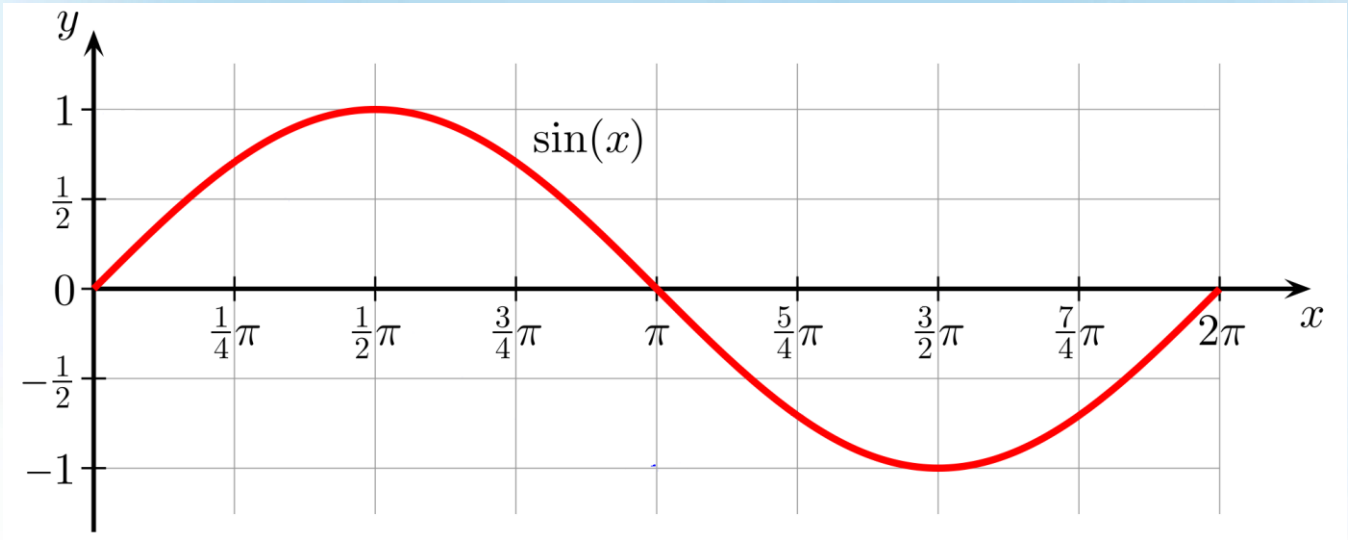
En savoir plus sur la 2ème partie

  - 3ème partie : Modifier la puissance / le couple du moteur brushless

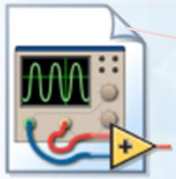
En savoir plus sur la 3ème partie

PILOTAGE : partie 1

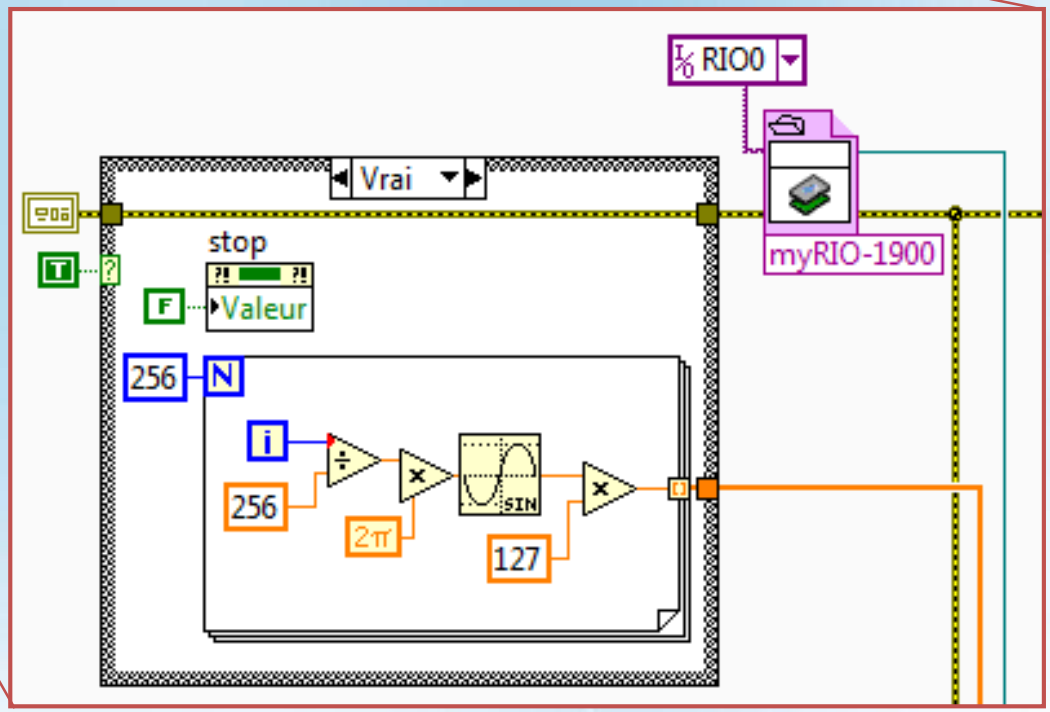
- Génération d'une sinusoïde et stockage des points constituants cette sinusoïde dans un tableau :
  - Le tableau contient 256 (résolution => axe x) points permettant de tracer une sinusoïde
  - Les valeurs max et min (amplitude du sinus => axe y) présentes dans ce tableau sont 127 et -127, donc .



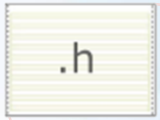
PILOTAGE : partie 1



RT Pilotage
Nacelle NC00.vi



PILOTAGE : partie 1



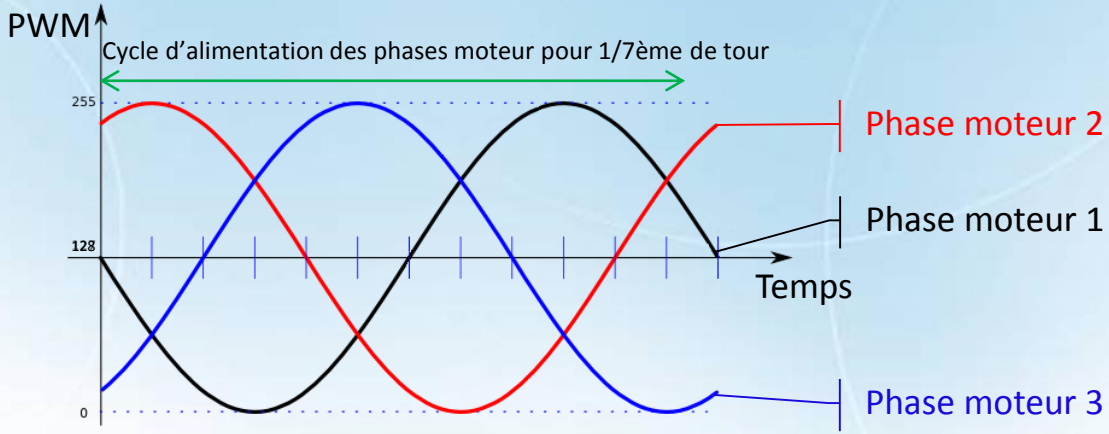
BLcontroller.h

nacelle_arduino | BLcontroller.h | EEPROMAnything.h | I2Cdev.cpp | I2Cdev.h | IMU | MPU6050.cpp | MPU6050.h | SerialCom.h | SerialCommand.cpp

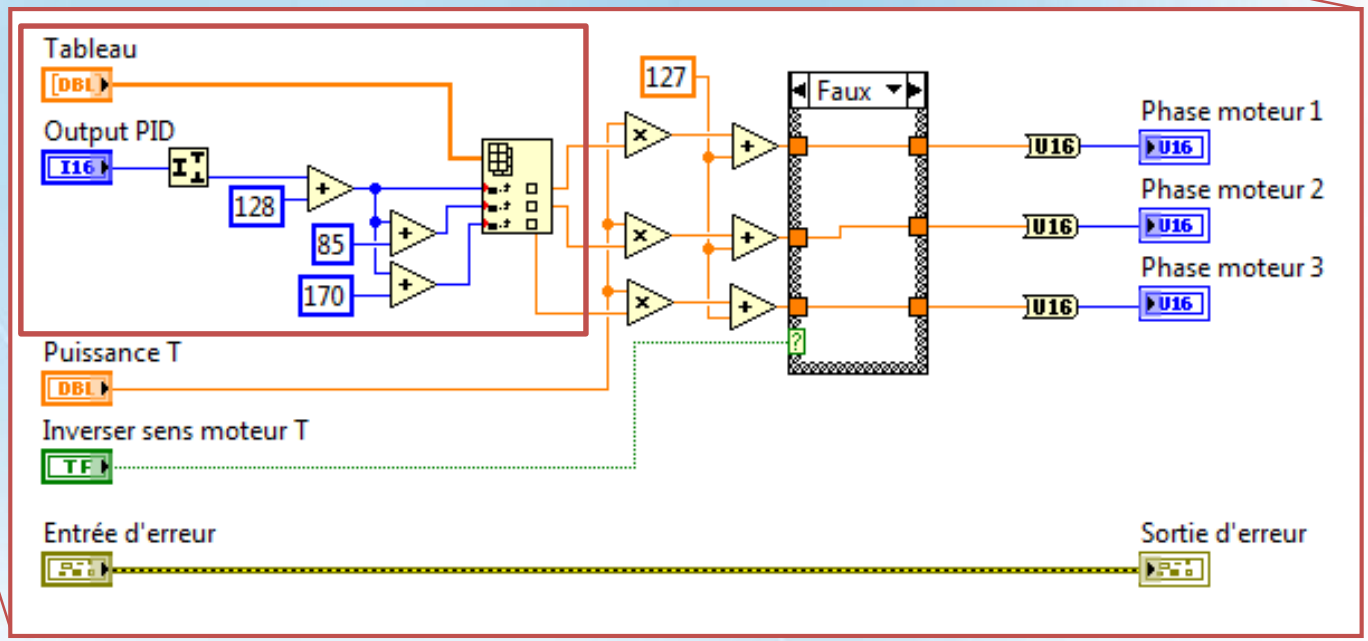
```
    pwm_b_motor1 = (uint8_t)pwm_b;
    pwm_c_motor1 = (uint8_t)pwm_c;
  }
}

void calcSinusArray()
{
  for(int i=0; i<N_SIN; i++)
  {
    pwmSinMotor[i] =  sin(2.0 * i / N_SIN * 3.14159265) * 127.0;
  }
}

void initMotorStuff()
{
  cli();
  calcSinusArray();
  sei();
}

/*********************************/
/* Motor Control IRQ Routine     */
/*********************************/
// is called every 31.5us
// minimize interrupt code (20 instructions)
ISR( TIMER1_OVF_vect )
{
```

PILOTAGE : partie 2

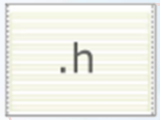
- Création des 3 phases moteur à partir de la valeur de sortie du PID :
  - Tronquer la valeur de sortie pour ne récupérer que les 8 premiers bits afin de faire varier constamment la sortie PID de 0 à 255 permettant d'être conforme aux nombres de points dans le tableau (tableau comprenant 256 valeurs).
  - On ajoute 128 à la valeur du PID afin d'osciller autour de cette valeur.
  - Puis afin de créer 3 phases moteur décalées de 120° (360/3=120°) on décale de 85 points (256/3=85) la phase 2 par rapport à la 1 et de 170 points 256*2/3=170) la phase 3 par rapport à la 1.
  - Enfin on utilise les 3 valeurs précédentes en tant qu'index dans le tableau de sinus afin de générer 3 phases moteur (PWM) de formes sinusoïdales



PILOTAGE : partie 2



PILOTAGE : partie 2



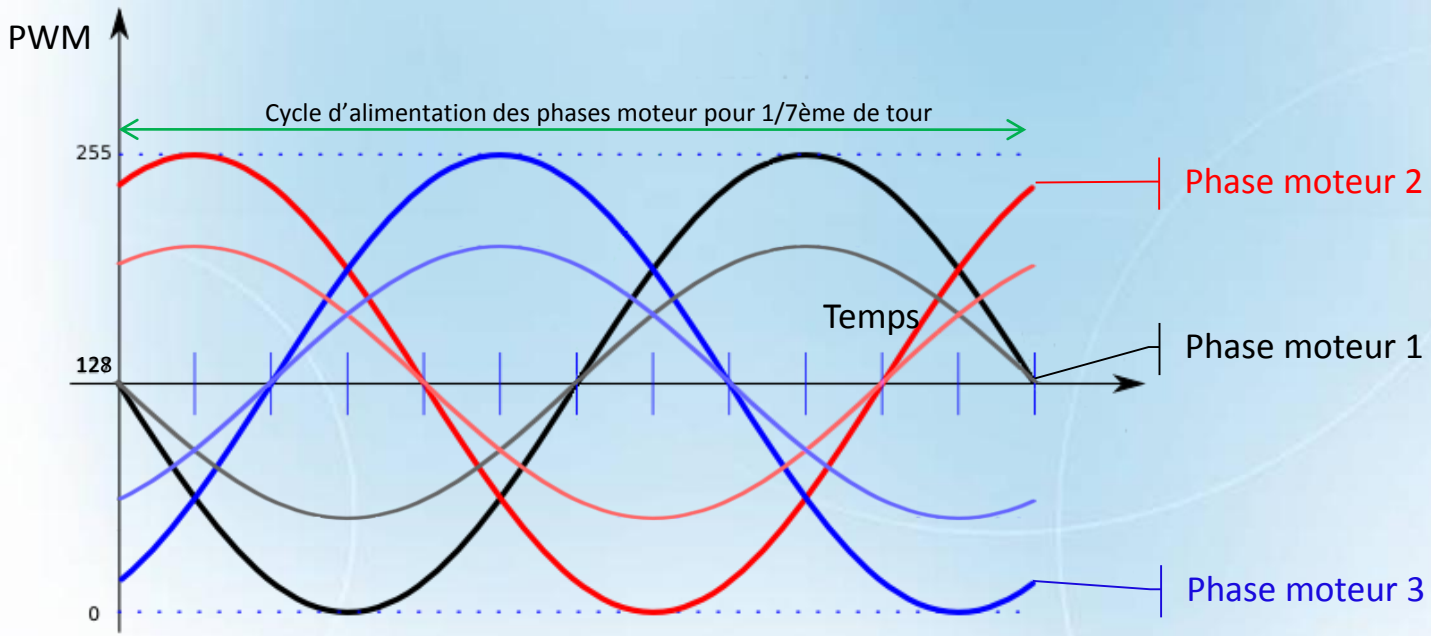
BLcontroller.h

nacelle_arduino | BLcontroller.h | EEPROMAnything.h | I2Cdev.cpp | I2Cdev.h | IMU | MPU6050.cpp | MPU6050.h | SerialCom.h | SerialCommand.cpp

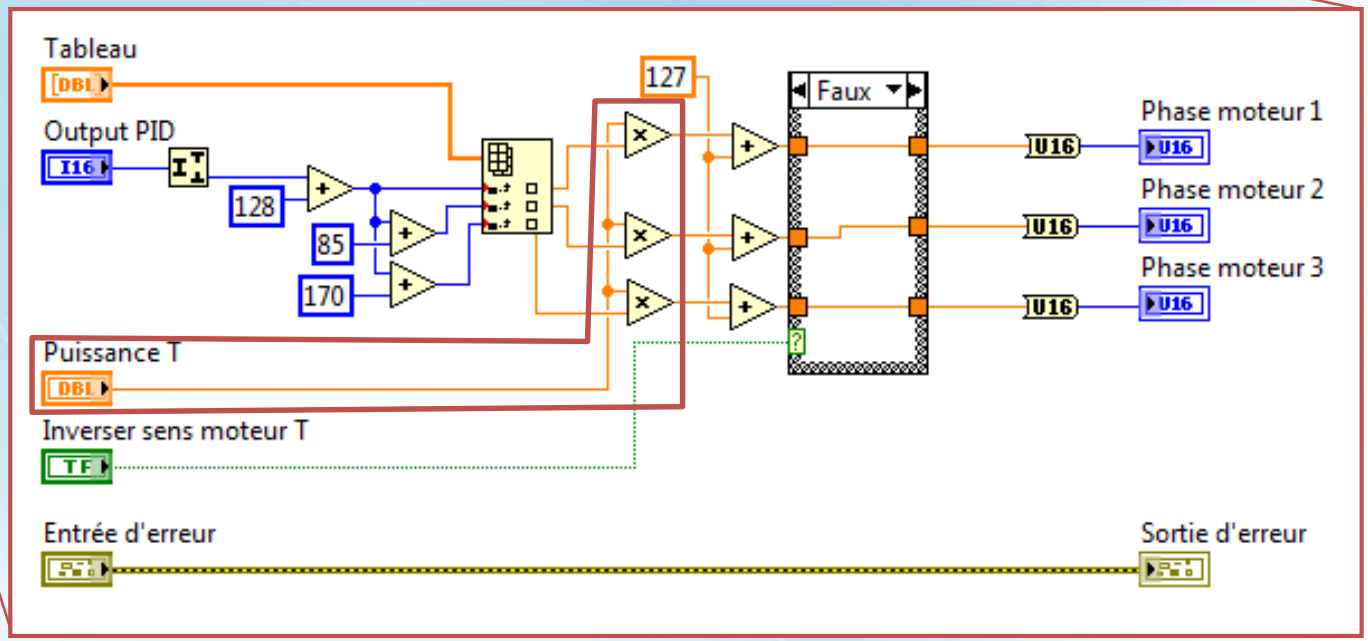
```
void MoveMotorPosSpeed(uint8_t motorNumber, int MotorPos, uint16_t maxPWM)
{
  uint16_t posStep;
  uint16_t pwm_a;
  uint16_t pwm_b;
  uint16_t pwm_c;

  // fetch pwm from sinus table
  posStep = MotorPos & 0xff;
  pwm_a = pwmSinMotor[(uint8_t)posStep];
  pwm_b = pwmSinMotor[(uint8_t)(posStep + 85)];
  pwm_c = pwmSinMotor[(uint8_t)(posStep + 170)];

  // apply power factor
  pwm_a = maxPWM * pwm_a;
  pwm_a = pwm_a >> 8;
  pwm_a += 128;

  pwm_b = maxPWM * pwm_b;
  pwm_b = pwm_b >> 8;
  pwm_b += 128;

  pwm_c = maxPWM * pwm_c;
  pwm_c = pwm_c >> 8;
  pwm_c += 128;
```

PILOTAGE : partie 3

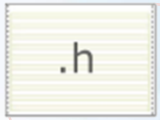
- Modifier la puissance / le couple du moteur brushless :
  - On multiplie les 3 valeurs des phases précédentes par un coefficient compris entre 0 et 1 afin de pouvoir diminuer le couple moteur



PILOTAGE : partie 3



PILOTAGE : partie 3



BLcontroller.h

nacelle_arduino | BLcontroller.h | EEPROMAnything.h | I2Cdev.cpp | I2Cdev.h | IMU | MPU6050.cpp | MPU6050.h | SerialCom.h | SerialCommand.cpp

```
void MoveMotorPosSpeed(uint8_t motorNumber, int MotorPos, uint16_t maxPWM)
{
  uint16_t posStep;
  uint16_t pwm_a;
  uint16_t pwm_b;
  uint16_t pwm_c;

  // fetch pwm from sinus table
  posStep = MotorPos & 0xff;
  pwm_a = pwmSinMotor[(uint8_t)posStep];
  pwm_b = pwmSinMotor[(uint8_t)(posStep + 85)];
  pwm_c = pwmSinMotor[(uint8_t)(posStep + 170)];

  // apply power factor
  pwm_a = maxPWM * pwm_a;
  pwm_a = pwm_a >> 8;
  pwm_a += 128;

  pwm_b = maxPWM * pwm_b;
  pwm_b = pwm_b >> 8;
  pwm_b += 128;

  pwm_c = maxPWM * pwm_c;
  pwm_c = pwm_c >> 8;
  pwm_c += 128;
```