ExplicationsFonctionRecalibrageTrieur

Le trieur est contrôlé par un moteur associé à un encodeur.
L'encodeur communique, ordonne et planifie une consigne de position. Le moteur s'exécute, du mieux qu'il peut, pour suivre la consigne.

Quand l'objet Trieur est initialisé, il est demandé à l'encodeur de prendre la position actuelle du moteur comme le 0. Ces positions sont notées en *ticks*.
Une rotation dans un sens fait augmenter la position (et donc le nombre de *ticks*), et dans l'autre la fait diminuer.

La distance en *ticks* entre deux positions est notée INTERVALLE_TICKS_MOULIN = 240.
Dans notre cas, pour faire passer le trieur d'une position à la suivante, il faut que l'encodeur ordonne d'ajouter 240 *ticks* à la consigne de position.
Pour faire un tour complet, il faut ajouter $6 \times 240 = 1440$ *ticks*.

Dans le code, cela est relativement simple et marche conformément.
On peut créer des commandes avec contrôle d'état et savoir quand le trieur est arrivé, ou presque arrivé (selon un nombre de *ticks* restant à parcourir).

Mais l'exactitude de la position physique du trieur laisse à désirer.
Il nous faut un encodeur physique qui nous dit qu'à tel instant le trieur est dans telle position.

Il a été ajouté un capteur magnétique et un petit aimant placés judicieusement sur le trieur. Quand l'aimant est devant le capteur, une information devient disponible.

On peut donc penser une fonction pour calibrer la trieur. Cette fonction s'appuie sur l'hypothèse que la consigne encodeur n'est jamais trop loin de la consigne idéale (celle qui faudrait pour que le trieur soit parfaitement comme on le souhaite). Si elle est fausse, la fonction ne peut pas faire office.
On verra que suffisamment veut dire que la consigne encodeur nous ferait tomber plus proche de la position trieur que l'on souhaite plutôt qu'une autre.
Donc il y a de la marge.

Le fonction qui recalibre :

```
    /**
     * Recalibrates the moulin's target position and logical
position based on the magnetic switch activation.
     */
    private void recalibrateMoulin() {
        int remainingDistance =
Math.abs(getMoulinMotorRemainingDistance());
        double intervals = (double) remainingDistance /
INTERVALLE_TICKS_MOULIN;
```

```
        int intPartOfRounded = (int) Math.round(intervals);
        double differenceToIntRounded = (intervals -
intPartOfRounded);
        double absDiff = Math×abs(differenceToIntRounded);
        int diffTicks = (int) Math.round(absDiff *
INTERVALLE_TICKS_MOULIN);

        if (diffTicks != 0) {
            if (differenceToIntRounded > 0) {

incrementMoulinTargetPosition(OFFSET_MAGNETIC_POS -
diffTicks);
            } else {

incrementMoulinTargetPosition(OFFSET_MAGNETIC_POS +
diffTicks);
            }
        }

        moulin.hardSetPosition(MAGNETIC_ON_MOULIN_POSITION +
intPartOfRounded);
    }
```

Elle ne s'exécute qu'une fois dès le capteur magnétique remarque l'aimant.

Explications :

`getMoulinMotorRemainingDistance()`

Donne le nombre de ticks restant à parcourir pour arriver à la consigne.

`Math.abs(getMoulinMotorRemainingDistance())`

Math.abs() force la valeur à être positive.

`double intervals = (double) remainingDistance / INTERVALLE_TICKS_MOULIN;`

Le (double) force le résultat de l'opération à être un décimal.
D'après les hypothèses, remainingDistance correspond à une distance restante à parcourir suffisamment proche d'une position de trieur. Donc le résultat de cette opération devrait être un décimal proche d'un entier. Cette entier est le nombre de position qu'il nous reste à parcourir.
Exemples :
Si intervals = 2.98 alors il nous reste 3 positions à parcourir.
intervals = 3.1 alors il nous reste 3 positions à parcourir.
intervals = 0.95 alors il nous reste 1 positions à parcourir.
intervals = 4.95 alors il nous reste 5 positions à parcourir.

```
int intPartOfRounded = (int) Math.round(intervals);
```
Cette ligne permet justement de trouver cet entier. Math.round() arrondi la valeur à l'entier le plus proche. (Int) permet de forcer l'entier à correspondre au type entier.

```
double differenceToIntRounded = (intervals - 
intPartOfRounded);
```
Cette récupère la différence entre l'entier (intPartOfRounded) et le décimal (intervals). Ca correspond à l'erreur de positionnement que devrait faire le trieur. Cette erreur est normalisé. Elle est en pourcentage par distance entre positions en *ticks*.

```
double absDiff = Math×abs(differenceToIntRounded);
int diffTicks = (int) Math.round(absDiff * 
INTERVALLE_TICKS_MOULIN);
```
On en prend la valeur absolue. Puis on la convertie en *ticks* en multipliant par INTERVALLE_TICKS_MOULIN, la distance entre deux positions en *ticks*.
diffTicks correspond donc à l'erreur, en *ticks*.

```
if (diffTicks != 0) {
    if (differenceToIntRounded > 0) {
        // If we rounded down, subtract the difference to 
correct the target position
        incrementMoulinTargetPosition(OFFSET_MAGNETIC_POS - 
diffTicks);
    } else {
        // If we rounded up, add the difference to correct 
the target position
        incrementMoulinTargetPosition(OFFSET_MAGNETIC_POS + 
diffTicks);
    }
}
```
Si cette différence n'est pas nulle, on ajoute ou on enlève, suivant le signe de l'erreur, diffTicks à la consigne actuelle.
Si differenceToIntRounded est positive, c'est qu'on allait trop loin, alors on enlève diffTicks.
Et inversement.

```
OFFSET_MAGNETIC_POS
```
Est une constante qui correspond à l'écart de positionnement entre la position du capteur magnétique et la position du trieur quand le capteur détecte

l’aimant. L’aimant est détecté avant d’être parfaitement devant le capteur.

```
// Set the moulin's logical position directly to the
calibrated position
moulin.hardSetPosition(MAGNETIC_ON_MOULIN_POSITION +
intPartOfRounded);
```

Cette ligne permet de forcer la position du trieur à la position trieur désirée.