Préparation TIPE - Découverte de l'Arduino

L'objectif de cette séance est de présenter rapidement l'environnement de programmation et la carte Arduino associée, qui, pour son prix modique (≈ 20 € pour celle qui est utilisée) propose des fonctionnalités très intéressantes pour l'acquisition et le traitement de données ainsi que le pilotage de systèmes (moteurs, etc.), ce qui peut-être très utile pour vos TIPE.

1 Présentation de l'environnement de programmation des cartes Arduino

La programmation de toutes les cartes de la famille Arduino, dont les modèles Uno (port USB large, type imprimante) ou Leonardo (port mini USB, type téléphone portable) utilisés dans le cadre de cette séance, se fait selon exactement le même code dans un langage simplifié adapté du C / C++, basé sur l'utilisation de « classes » (macro-commandes) faciles à comprendre et à modifier, y compris par des personnes non spécialistes de la programmation.

```
// fonction d'initialisation de la carte
void setup()
{
// contenu de l'initialisation
}

// fonction principale, elle se répète à l'infini
void loop()
{
// contenu de votre programme
}
```

FIGURE 1 – Structure générique d'un programme Arduino.

La fonction **setup()** est appelée une seule fois lorsque le programme commence. C'est ici que l'on va initialiser notre système, mettre en place les différentes sorties utilisées et écrire les fonctions que l'on veut voir executé une seule fois.

La fonction **loop()** est executée en boucle, comme son nom l'indique. Dès que toutes les instructions écrites ici ont été excutées, la fonction est appelée de nouveau, encore et encore. C'est ici que l'on écrit le code qui servira à faire l'acquisition et le traitement de données ainsi que le pilotage de nos moteurs par exemple.

Une différence **fondamentale** entre la programmation Arduino et la programmation en Python est la gestion du **typage** des variables ! Si en Python, le type d'une variable (int, float, ...) est attribué **automatiquement**, ce n'est pas le cas en programmation Arduino. Dès lors que l'on définit une nouvelle variable, on doit **spécifier** son type !

Type	Domaine	Étendue décimal	Étendue en bits
char	$\in \mathbb{N}$	-128 à +127	8 bits
int	$\in \mathbb{N}$	-32 768 à +32 767	16 bits
long	$\in \mathbb{N}$	-2 147 483 648 à +2 147 483 647	32 bits
float	$\in \mathbb{R}$	$-3.4 \cdot 10^{38}$ à $+3.4 \cdot 10^{38}$	32 bits
boolean	$\in \mathbb{R}^+$	0 à 1	1 bit

FIGURE 2 – Taille d'encodage des différents typages usuels des variables en programmation Arduino.

Exemple :

```
// variable est fausse, car elle vaut FALSE, du terme anglais "faux"
boolean variable = FALSE;

// définition de variables
int a = 15;
float z = 0;
```

Comme vous pouvez le remarquer, chaque ligne est terminée par un **point-virgule** ! De plus, pour définir des **variables globales**, qui seront accessibles à la fois par la fonction **setup()** et la fonction **loop()**, il est nécessaire de les définir dans l'espace **au-dessus** de la fonction **setup()** ! Sinon, comme en Python, leur existence est restreinte au « scope » de la fonction au sein de laquelle elles ont été définies.

Remarque : Puisqu'une carte Arduino a un espace mémoire limité, comme tout micro-contrôleur, on privilégie toujours le type utilisant le moins de bits qui est suffisant pour l'objectif visé par la variable définie. Par exemple, si une variable sert juste à indiquer un état **booléen**, il est inutile de l'encoder en tant que **long** sur 32 bits. On économise ainsi de l'espace mémoire.

Remarque : Une autre manière d'écrire une variable booléen ayant comme valeur **TRUE** est de lui affecter la valeur **HIGH** ou la valeur **1**. Inversement, pour une variable booléen ayant comme valeur **FALSE**, on peut de manière équivalente lui affecter la valeur **LOW** ou **0**.

2 Prise en main de la carte Arduino

Manipulation : Sur la carte Arduino à disposition, on vous aidant de la FIGURE suivante et de la description associée, identifier et localiser les différents composants ainsi que les ports d'entrées et de sorties, appelées « pins ».

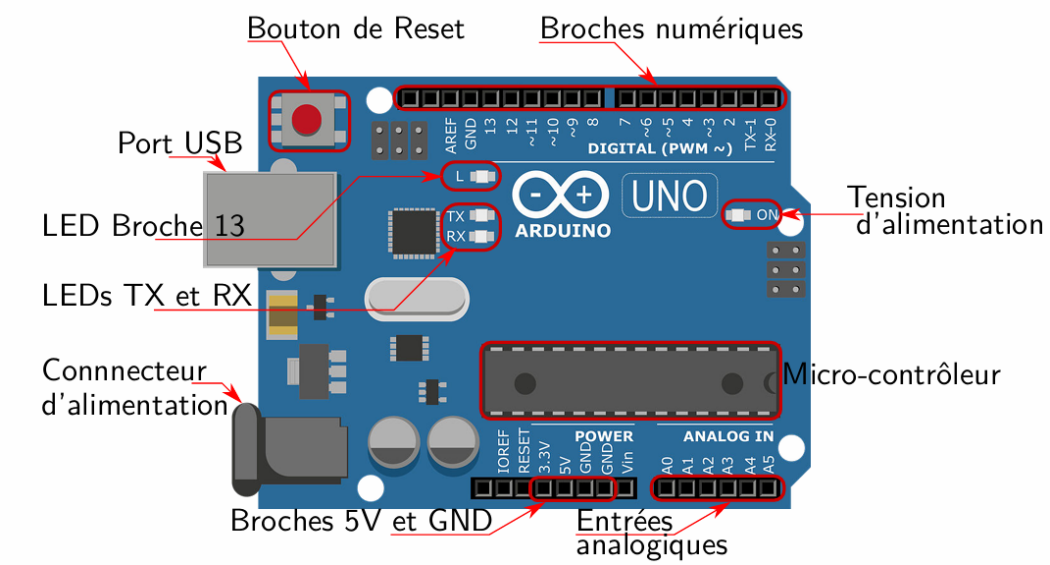


FIGURE 3 – Description des différents « pins » de la carte Arduino Uno/Leonardo.

En plus des ports **3.3V**, **5V** et **GND** (0 Volt) qui fournissent deux sources de tension régulée permettant d'alimenter des circuits électroniques de faible puissance en 3.3 Volts ou 5 Volts, on retrouve 14 **broches numériques** (les entrées-sorties logiques) et 6 broches d'entrées analogiques.

Entrées-sorties Logiques :

Ces pins, numérotés de 0 à 13, peuvent prendre deux états logiques : l'état **HIGH** (5 Volts) ou l'état **LOW** (0 Volt). La fonction permettant de choisir entre ces deux états est **digitalWrite(pin_number, state)**.

De plus, les 6 pins ayant le symbole ∽ peuvent servir de sorties PWM afin d'obtenir une tension de sortie modulée entre 0 et 5 Volts, en utilisant la fonction **analogWrite(pin_number, alpha)** avec α correspondant au rapport cyclique de la modulation, codée sur 8 bits ($\alpha = 0 \Longrightarrow V_s = 0$ Volt et $\alpha = 255 \Longrightarrow V_s = 5$ Volts)

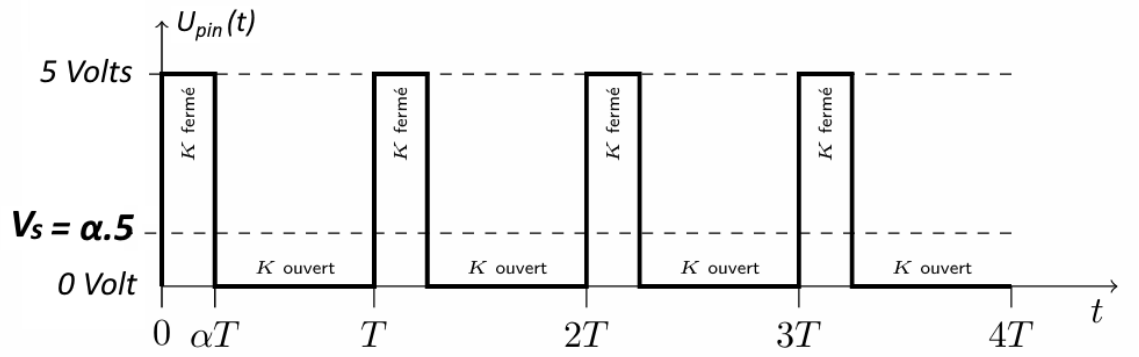


FIGURE 4 – Rappel du fonctionnement de la modulation à largeur d'impulsion (MLI ou PWM en anglais).

Tous ces pins peuvent aussi servir d'entrée numériques pour lire les données provenant de capteurs numériques en utilisant la fonction **digitalRead(pin_number)**.

Entrées Analogiques :

Ces pins, numérotés de A0 à A5, permettent de « lire » des entrées provenant de capteurs analogiques variant entre 0 et 5 Volts. Elles sont associées à un CAN codant la tension d'entrée sur 10 bits (0 pour 0 Volt à 1023 pour 5 Volts) en utilisant la fonction **analogRead(pin_number)**.

Communication avec l'ordinateur :

Pour téléverser le programme, on utilise la liaison USB pour communiquer avec la carte par une liaison série. Une fois un code téléversé sur une carte, la mémoire de type « Flash » de celle-ci permet de conserver le programme écrit « indéfiniment » en mémoire, même sans alimentation.

Remarque : Pour cette séance, en plus de servir à la communication série entre le PC et la carte Arduino, on utilise aussi cette liaison USB avec l'ordinateur pour fournir l'alimentation à la carte. Dans le cas général, il est aussi possible d'alimenter la carte Arduino par une source externe de tension comprise entre 7 et 12 Volt en utilisant l'entrée V_{in} présente sur la carte.

3 Premier programme : Allumage d'une LED

Pour la première activité, on va tester un programme simple pour allumer la LED associée au pin 13 de la carte Arduino.

Manipulation : Ouvrez le script `Blink.ino`, téléversez-le sur la carte Arduino en cliquant sur « Verifier » puis sur « Téléverser ». Si besoin, sélectionner le port COM sur lequel est branché la carte Arduino. *N'hésitez pas à m'appeler si vous bloquer à cette étape.*

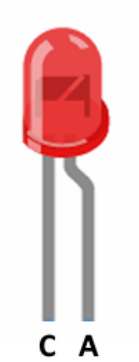
Q 1. Observer la LED associée au pin 13. En déduire le rôle des fonctions **pinMode()** et **delay()**.

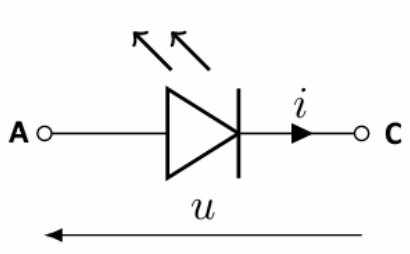
Q 2. Modifier le programme pour que la LED soit allumée pendent 2 s et éteinte pendant 0,5 s.

4 Deuxième programme : Création d'un feu tricolore

On va maintenant allumer une LED qui n'est pas directement sur la carte.

Matériel à disposition : Une plaquette de prototypage, une feuille de présentation du câblage de la plaquette, trois résistances 220 Ω et trois diodes luminescente (LED) : une verte, une orange et une rouge.





Lorsque l'on utilise une diode, l'électricité ne peut circuler que dans un sens. L'anode, reliée à l'alimentation, est la patte la plus **longue** et la cathode et la patte la plus courte.

Pour une question de protection électrique de la diode, celle-ci doit toujours être **associée en série à une résistance** pour limiter le courant la traversant. Sinon, elle risque d'exploser et l'on aimerait éviter ça !

Manipulation : Créer un nouveau script que vous nommerez `Feu_tricolore.ino`, il suffit pour cela de cliquer sur « Fichier » puis « Nouveau ». On va tout d'abord écrire un programme qui permet d'allumer la LED verte puis un programme permettant de créer un feu tricolore à partir des trois LED.

Q 3. Proposez un schéma de câblage pour allumer la LED. *Avant de le réaliser, appelez-moi pour vérification !*

Q 4. En vous inspirant du script `Blink.ino`, écrivez un programme permettant d'allumer la LED verte pendant 1 s puis de l'éteindre pendant 1 s.

Q 5. Mettez en place le câblage et complétez le code précédent afin de réaliser un système de feu tricolores.

5 Troisième programme : Lecture de la position angulaire d'un potentiomètre

On va maintenant lire la mesure issue d'un capteur analogique. On utilise pour cela un potentiomètre angulaire qui fournit une tension V_a entre 0 Volt et $V_{alim} = 5$ Volt image de sa position angulaire θ_{pot} comprise entre 0 et 350°.

Q 6. Quelle est la résolution de l'ensemble {capteur+CAN} ? *On rappelle que la résolution est la plus petite variation d'angle interpretable par la carte Arduino.*

Remarque : La notion d'acquisition « analogique » doit être comprise du point de vue de l'utilisateur : au niveau de la carte Arduino, une conversion analogique → numérique (CAN ou ADC en anglais) est nécessaire pour que le microcontrôleur puisse traiter les informations.

Manipulation : Ouvrez le fichier `Potentiometre.ino` et analysez-le. L'objectif va être de compléter la ligne manquante afin de lire la valeur d'angle mesurée par le potentiomètre puis de l'afficher, en degré, sur l'ordinateur .

Q 7. Que fait la fonction **delay()** ? En déduire la fréquence d'acquisition des mesures.

Q 8. Faire un schéma explicatif du principe de fonctionnement d'un potentiomètre angulaire.

Q 9. Réaliser le câblage du potentiomètre avec la carte Arduino et compléter le script pour lire la valeur de θ_{pot}.

Q 10. Afficher la valeur mesurée en degré sur le terminal fourni avec l'environnement de programmation. Pour cela cliquer sur « Outils » puis « Moniteur série ». En tournant manuellement le potentiomètre, vérifier que la valeur affichée est cohérente avec la position angulaire réelle du potentiomètre.

Câblage du potentiomètre

6 Quatrième programme : Contrôle manuel du feu tricolore

On se propose maintenant d'améliorer le système de feu tricolore réalisé précédemment afin qu'il réponde au mieux aux deux cas d'utilisation suivant :

- En cas d'embouteillage important, on aimerait pouvoir prendre le contrôle manuel du feu actif. Cela permettait, par exemple, afin de pouvoir laisser plus longtemps le feu au vert dans le but de désengorger une voie bouchée.
- Afin de minimiser la consommation énergétique du feu tricolore, il est inutile qu'il s'allume en permanence à son intensité lumineuse maximale. On aimerait pouvoir régler cela manuellement.

On va utiliser la mesure d'angle de deux potentiomètres afin de commander indépendamment chacun de ces deux aspects.

6.1 Contrôle du feu actif en utilisant un premier potentiomètre

Objectif : Créer un script Arduino qui permet de sélectionner quel diode, parmi la verte, l'orange et la rouge, s'allume en fonction de la position mesurée d'un potentiomètre angulaire. On considéra que pour $\theta_{pot} \in [0, 100°]$ le feu doit être au vert, que pour $\theta_{pot} \in [101, 200°]$ le feu doit être à l'orange et qu'il sera au rouge le reste du temps.

Q 11. Proposer un schéma de câblage du système de contrôle du feu actif. *Avant de le réaliser, appelez-moi !*

Q 12. Écrire un programme Arduino qui permet de réaliser l'objectif visé.

Q 13. Afficher, en degré, la valeur θ_{pot} mesurée sur le terminal fourni avec l'environnement de programmation et vérifier le contrôle manuel du feu actif.

6.2 Contrôle de l'intensité lumineuse des feux tricolores en utilisant un deuxième potentiomètre

Objectif : Compléter le script Arduino précédent pour ajouter la possibilité de contrôler l'intensité lumineuse des diodes en fonction de la position mesurée d'un deuxième potentiomètre angulaire. On considéra que pour $\theta_{pot} = 350°$ on souhaite avoir l'intensité lumineuse maximale alors que pour $\theta_{pot} = 0°$, on souhaite que les LED soient éteintes.

Pour réaliser cela, on se propose de moduler la tension d'alimentation de l'ensemble {résistance+diode} en fonction de l'angle θ_{pot}.

Q 14. Proposez une façon de réaliser cette modulation de tension à partir de la carte Arduino. *Appelez-moi pour validation !*

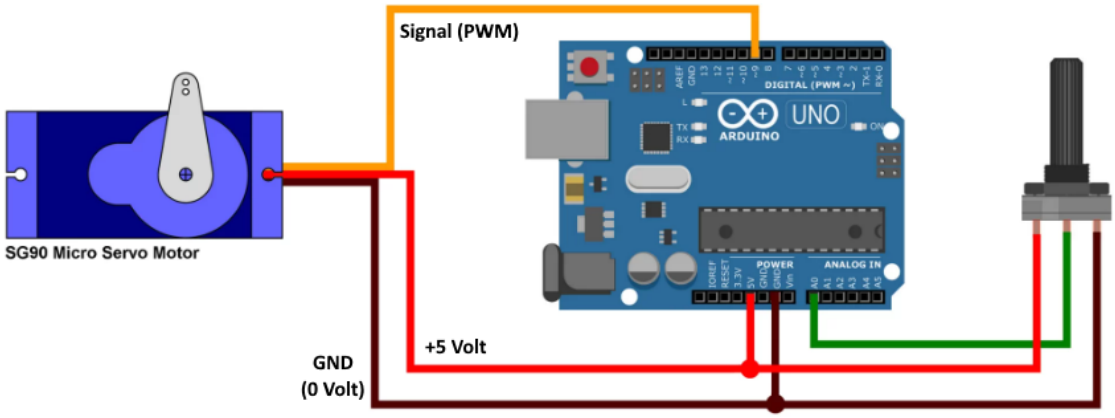
Q 15. Écrire le programme ajoutant la fonctionnalité souhaitée au système de feu tricolore.

Indication : Vous pouvez utiliser la fonction **map(x, x_min, x_max, y_min, y_max)** qui permet de changer l'échelle d'une variable x par transformation affine. Pour x variant de x_{min} à x_{max}, la fonction **map()** retourne une variable y variant de y_{min} à y_{max}.

7 Dernier programme : Commande manuelle d'un servo-moteur asservi en position

Pour cette dernière partie, on va commander manuellement un servo-moteur asservi en position pour qu'il reproduise la position angulaire que l'on impose manuellement au potentiomètre.

Manipulation : Reproduire le câblage suivant entre le potentiomètre, le servo-moteur et la carte Arduino.



Fonctionnement des servo-moteurs DF9GMS :

On rappelle qu'un servo-moteur est un « super-composant » constitué d'un moteur électrique, d'un capteur de position angulaire et d'une petite carte de commande permettant de traiter cette mesure et de la comparer à une consigne externe donnée pour générer une tension d'alimentation du moteur afin qu'il suive la consigne externe fournie.

Dans le cas des servo-moteurs à disposition, ceux-ci sont asservis en position. Cela signifie que la consigne externe qu'on leur fourni via le fil « Signal (PWM) » est interprétée comme une position à atteindre.

Cette commande de position, c'est-à-dire, est envoyée au servo-moteur sous forme d'un signal PWM dont le rapport cyclique encode l'angle θ_{mot} à atteindre. Comme cela est décrit sur la FIGURE 5, la PWM attendue par le servo-moteur moteur a une période $T = 20$ ms, l'angle consigne varie continument de 0°, encodé par une largeur d'impulsion de 1 ms (rapport cyclique $\alpha = 1/20$), à un angle de 180°, pour un rapport cyclique $\alpha = 2/20$.

Q 16. En vous inspirant de ce que vous avez fait jusqu'à présent, écrivez un script Arduino qui permet de générer le signal PWM de commande du servo-moteur pour qu'il suive l'angle θ_{pot} du potentiomètre.

Remarque : Le servo-moteur commandé avec notre PWM « custom » n'a pas du tout un mouvement fluide ! Cela vient en grande partie du fait que notre signal PWM n'est pas du tout synchronisé sur l'horloge interne de la carte Arduino. C'est cette horloge qui définit le temps, et surtout sa discrétisation, sur laquelle se base les fonctions **delay()**, **delayMicroseconds()** ou **analogWrite()**. Par conséquent, la largeur d'impulsation de notre modulation n'est pas du tout précise.

Gérer proprement ces problèmes de synchronisation temporelle dépasse largement le cadre de la CPGE et demanderait beaucoup d'efforts... En pratique, des personnes bien intentionnées ont déjà travaillé à notre place et ont partagé leur code. Il nous reste qu'à utiliser leurs librairies toutes prêtes.

Manipulation : On se propose de découvrir une de ces librairies, appelée « Servo ». Pour cela, cliquez sur « Fichier », « Exemples », « Servo » puis « Knob ». Analyser le code puis téléverser-le.

C'est quand même plus fluide ! C'est ce type de librairie que vous serez amené à utiliser en TIPE si vous avez besoin de contrôler un servo-moteur. Gardez en tête que pour le TIPE, on essaie toujours d'utiliser du code existant ! Rien ne sert de « recoder la roue » et il ne viendrait à l'esprit de personne de vouloir recoder la fonction sinus en Python alors que la fonction `numpy.sin()` *fait très bien le travail !*

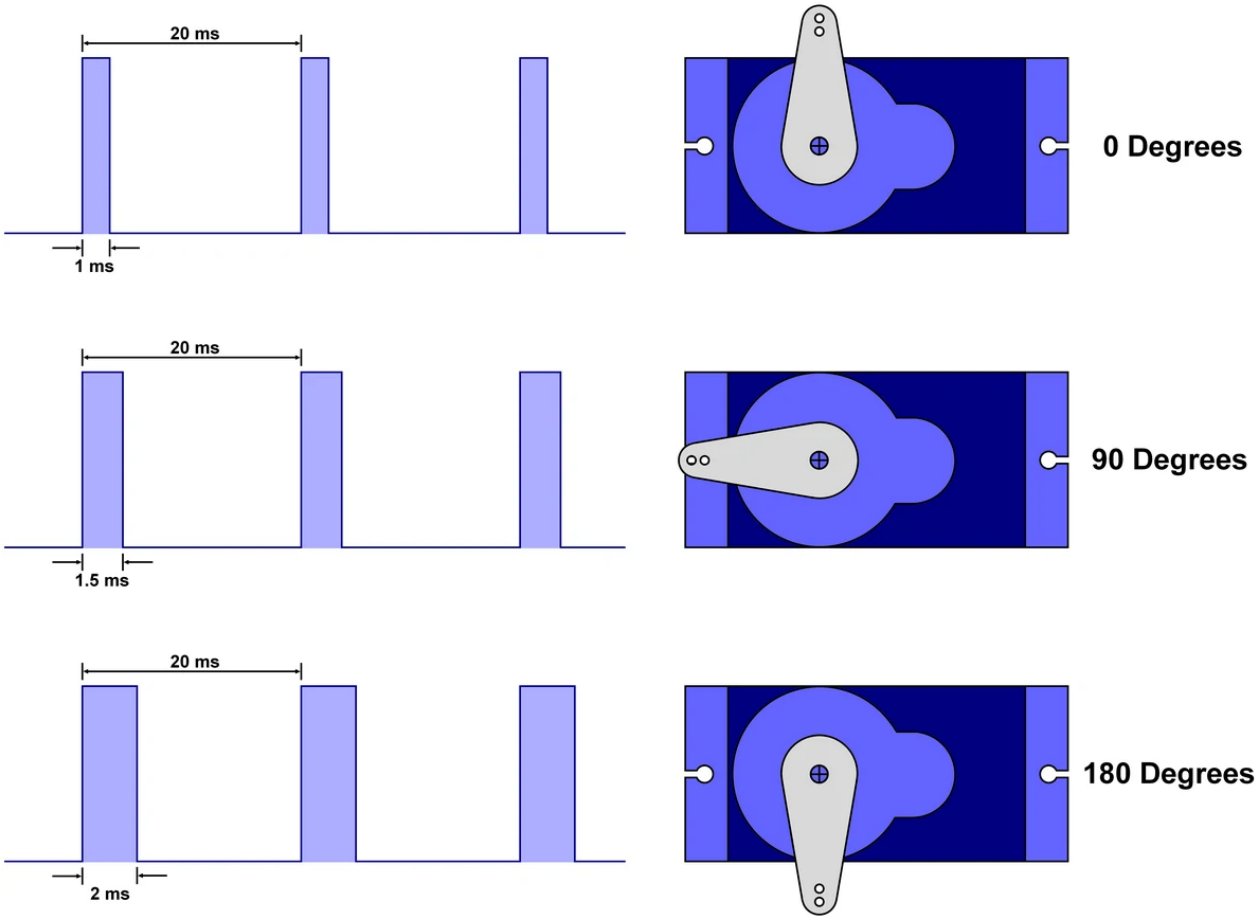


FIGURE 5 – Description de l'encodage de la position θ_{mot} à atteindre en fonction du rapport cyclique α donné.

Annexe - Syntaxe du langage

- **Condition if/else :**
  - The **`if... else`** allows greater control over the flow of code than the basic **`if`** statement, by allowing multiple tests to be grouped together. An **`else`** clause (if at all exists) will be executed if the condition in the **`if`** statement results in `false`. The **`else`** can proceed another **`if`** test, so that multiple, mutually exclusive tests can be run at the same time.

    Each test will proceed to the next one until a true test is encountered. When a true test is found, its associated block of code is run, and the program then skips to the line following the entire if/else construction. If no test proves to be true, the default else block is executed, if one is present, and sets the default behavior.

    Note that an **`else`** if block may be used with or without a terminating **`else`** block and vice versa. An unlimited number of such **`else if`** branches is allowed.

```
1 if (condition1) {
2   // do Thing A
3 }
4 else if (condition2) {
5   // do Thing B
6 }
7 else {
8   // do Thing C
9 }
```

- **Boucle for :**
  - The **`for`** statement is used to repeat a block of statements enclosed in curly braces. An increment counter is usually used to increment and terminate the loop. The **`for`** statement is useful for any repetitive operation, and is often used in combination with arrays to operate on collections of data/pins.

```
1 for (initialization; condition; increment) {
2   // statement(s);
3 }
```

    Par exemple, on peut écrire : **`for`** `(int i = 0; i < 10; i = i + 3)`. Remarque, plutôt que d'écrire `i=i+1`, on peut écrire `i++`.

- **Opérateurs booléens :**
  - `&&` : Logical `&&` results in `true` only if both operands are `true` (**`and`** en python) ;
  - || : Logical || results in a `true` if either of the two operands is `true` (**`or`** en python) ;
  - ! : Logical ! results in a `true` if the operand is `false` and vice versa (**`not`** en python).

- **Opérateurs de comparaison :**
  - Comme en Python, on a : != (not equal to), < (less than), <= (less than or equal to), == (equal to), > (greater than), >= (greater than or equal to)

- **Opérateurs Arithmétiques :**
  - % (remainder), ⋆ (multiplication), + (addition), − (subtraction), / (division), = (assignment operator)

- **Commentaires :**
  - `/* */` : block comment
  - `//` : single line comment