INSTITUT d'OPTIQUE GRADUATE SCHOOL

LIVRABLE

PROJET VISION INDUSTRIELLE

Banc industriel de tri de formes et de couleurs

VIEWSORT

Projet électronique et traitement de l'information, SupOptique **2023**

Bayero Aboubakar
Gabillet Thomas
Garnier Julien

Rapport technique
Objectifs techniques, réalisation et suivi pas à pas du projet

VIEWSART

Livrable final

Tables des matières

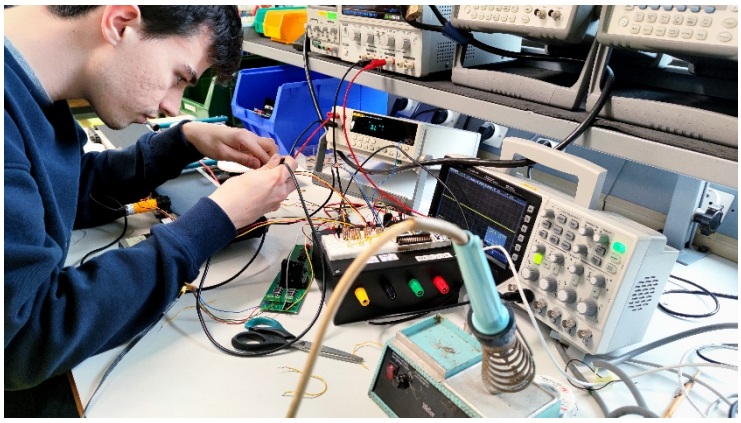
Membres du groupe

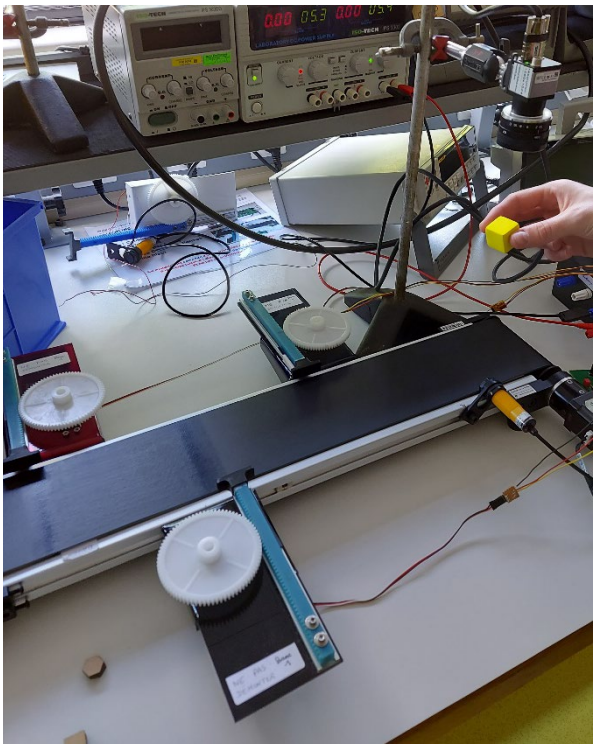
Aboubakar Bayero

Thomas Gabillet

Julien Garnier

"Nous attestons que ce travail est original, que nous citons en référence toutes les sources utilisées et qu'il ne comporte pas de plagiat"





Introduction

Présentation du projet

Dans le milieu industriel, le procédé de tri selon le type d'objet est très utilisé. Par exemple, le procédé de tri avec traitement d'images de détection de formes ou de couleurs, permet de séparer les pièces avec des défauts ou bien associer les objets à leur emballage respectif selon leur couleur.

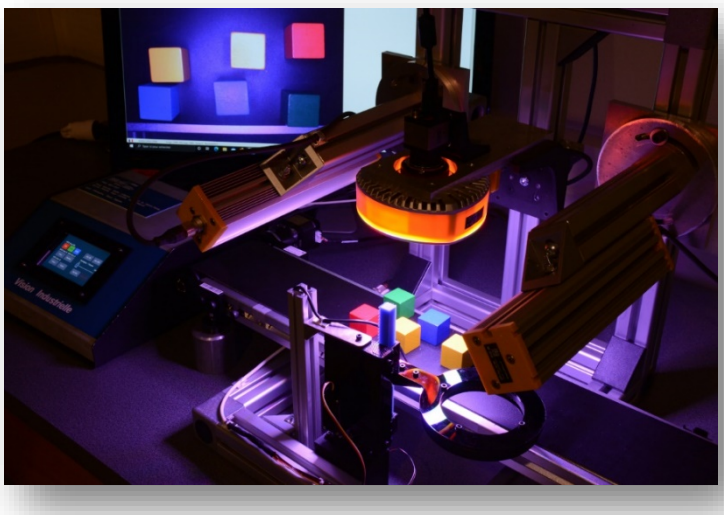
Le projet consiste à concevoir un système de tri par détection visuelle à l'aide d'une caméra, en répartissant différents objets selon la forme ou bien la couleur dans différents compartiments.

Pour cela, VIEWSORT compte modéliser un système de « vision industrielle » afin de mettre en place une détection de tri d'objets sur une maquette de convoyeur pilotée par un microcontrôleur en interaction avec un programme de détection de formes et de couleurs. Une IHM (Interface Homme Machine) est réalisée par une application qui permet de sélectionner le type de tri et les objets à trier tout en pilotant automatiquement les moteurs.

VIEWSORT

Ce système pourra par la suite être utilisé dans de nombreuses applications industrielles de tri de composants. Sur des chaînes de production, nous pensons que ce système permettra d'automatiser et d'optimiser ce tri.

Nous pensons à des entreprises comme M&M's ou Lego qui pourront utiliser ce système lors de la fabrication de leurs produits.



Objectifs

La mise en mouvement du convoyeur se fait par l'intermédiaire d'un moteur pas à pas piloté par une carte Nucléo.

Nous souhaitons réaliser la détection de 3 types d'objets possibles, donc nous avons besoin de 3 boîtes de tri, dans lesquelles les objets seront poussés par des servomoteurs pilotés également par la carte Nucléo.

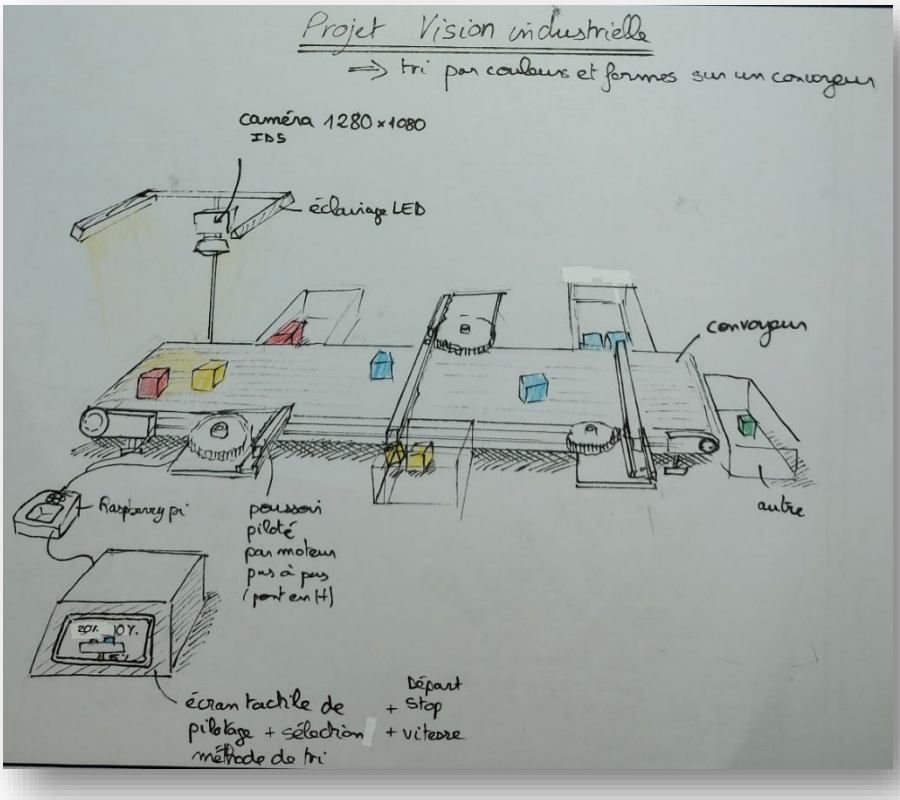
La communication en liaison série se fera entre la Nucléo et un PC qui réceptionnera une photographie d'un instant capturé à la détection d'un objet par la cellule photosensible en TOR.

Un programme Python permettra de réaliser le traitement d'images et de réaliser la reconnaissance en « couleur » ou en « forme » de l'objet détecté, pour ensuite renvoyer l'information aux servomoteurs, afin de le pousser dans sa boîte associée.

Une application réalisée avec la bibliothèque PyQt permettra de réaliser les interactions avec le convoyeur et l'utilisateur.

Schéma de principe

Elaboration du schéma de fonctionnement avant le démarrage du projet.



Cahier des charges

Le cahier des charges contient l'ensemble des éléments et règles qui devront être suivi dans le cadre de la réalisation du projet. Ce document défini donc le cadre du développement du projet de sa phase initiale à sa phase finale.

Liste du matériel :

Nous avons à notre disposition du matériel qui va être utilisé dans la réalisation du système. Le système est composé de :

- Un convoyeur (Dobot)

https://en.dobot.cn/service/download-center?keyword=&products%5B%5D=321

- 3 servomoteurs
- Une caméra (UI-3240CP Rev. 2), RGB 1280 x 1024 px

https://fr.ids-imaging.com/store/ui-3240cp-rev-2.html

- 3 boites / compartiments
- Un PC et son écran (interface Python) et Keil Studio (C++)
- Une carte Nucléo

https://fr.farnell.com/stm32-nucleo-development-board

- Photoelectric switch (E3F – DS30)

https://www.finglai.com/products/sensors/cylinder-amplifier-photoelectric-sensors/E3F-DS30/E3F-DS30C4.html

Contraintes du système

Ce projet doit respecter de nombreuses contraintes pour que le système fonctionne dans son utilisation souhaitée. Les contraintes peuvent être regroupées suivant la partie du système qu'elles affectent.

Contraintes « caméra » :

- Luminosité suffisante pour détecter les formes ou les couleurs.
- Utilisation de la caméra (UI-3240CP Rev. 2) (qualité et ouverture)

Contraintes « interface utilisateur » :

- Temps de réponse suffisamment faible au cours du fonctionnement du système
- Mise en place d'un bouton « urgence » pour arrêter complètement le système en cas de problème.

Contraintes « moteurs + pousseurs » :

- Les pousseurs doivent être au ras du sol (1mm) pour pousser les objets de toute taille hors du convoyeur.
- Vitesse des moteurs suffisamment grande.

Contraintes « convoyeur » :

- Vitesse constante.
- Charge totale des objets limitée à 500g.
- Connaitre précisément la position des moteurs, caméra sur le convoyeur pour calculer le temps parcouru par l'objet avant de le pousser.

Contrainte « capteur infrarouge » :

- Assez proche du convoyeur pour détecter les objets fins.

Contraintes « objet » :

- L'objet doit pouvoir être poussé par les servomoteurs (masse et taille à définir).
- Ne pas poser plusieurs objets en même temps pour ne pas pousser les mauvais cubes (mise en série des objets et suffisamment espacés).
- Pas d'objet de la même couleur que le convoyeur.
- Les objets doivent être placés au milieu du convoyeur pour être détectés par le capteur.

Contrainte « boites » :

- Volume des boites suffisant (capacité de stockage). Quand la capacité de stockage est atteinte on arrête le système

Performance du système

Nous dressons ici une liste exhaustive des performances du système ainsi qu'un nombre de caractéristiques quantifiable que nous utiliserons lors de la conception du système.

Performances « caméra » :

- Qualité : 1280 x 1024
- Ouvertures : 16, 8, 4, 2, 1.4
- Fréquence d'image max : 60

Performance « servomoteurs » :

- Signal d'entrée : signal PWM (Nucléo : 3,3V crête)

Performances « convoyeur » :

- Charge maximale : 500g
- Longueur : 600mm
- Largeur : 100mm
- Vitesse maximale : 120mm/s
- Masse : 4.2 kg

Moteur pas à pas :

- 100 pas
- Alimentation en 12V (5,7V dans notre cas)
- Signal d'entrée : signal PWM (Nucléo : 3,3V crête)

Performances « capteur infrarouge » :

- Distance de détection : 5cm – 30cm
- Alimentation : 6 - 36 Vdc
- Temps de réponse : 1,5ms
- Signal de sortie : tout ou rien

Performance « boite » :

- Tailles : 15.5cm x 7cm x 3cm

Performances « PC » :

- Langage de programmation : Python
- Distribution Windows
- Communication série avec la Nucléo par port USB

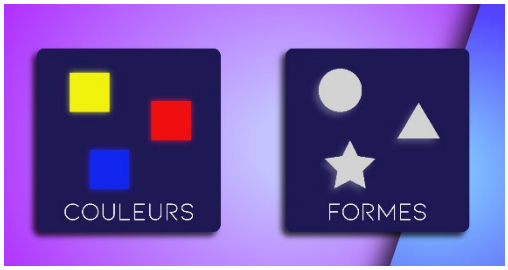
Performances « Nucléo » :

- Langage de programmation : C++
- Tensions entrée/sortie : 0V/3,3V

Notice d'utilisation

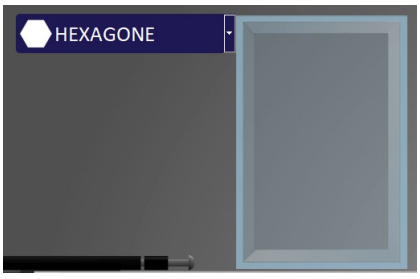
Notre projet de vision industrielle composé d'une caméra, d'un convoyeur, de servomoteurs et d'une Nucléo est accompagné d'une interface utilisateur permettant de contrôler l'ensemble du système simplement. Voici les différentes étapes pour utiliser notre système dans les bonnes conditions :

- Lancer le programme de l'interface utilisateur. Vous arrivez sur cette 1ère page qui vous demande de choisir le mode de tri que vous voulez effectuer. Cliquez simplement sur l'icône que vous souhaitez.

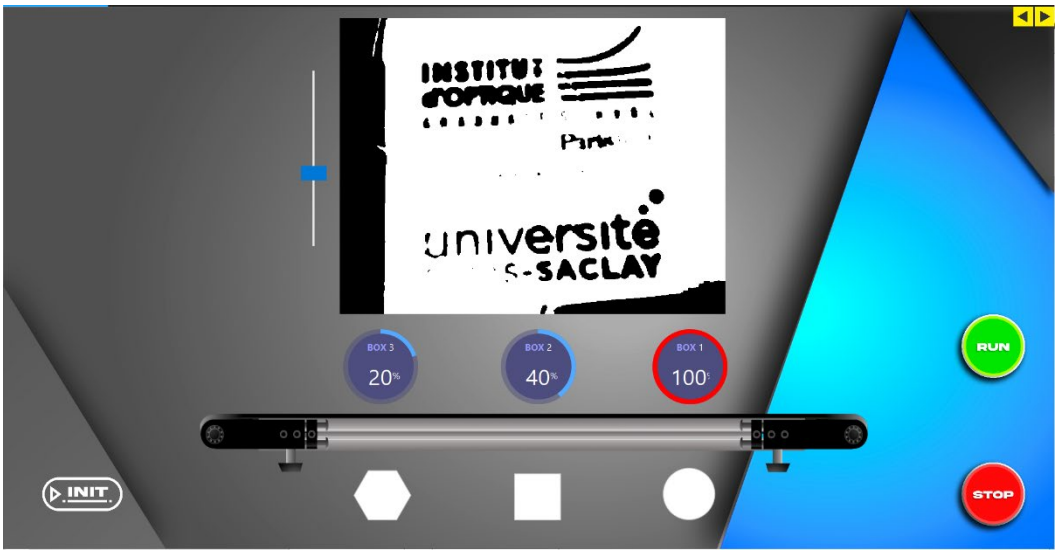


- La 2ème page vous demande de choisir pour chaque servomoteur, la forme ou la couleur que vous voulez lui associer pour qu'il ne puisse pousser que les objets ayant cette caractéristique.

- Pour choisir la forme ou la couleur à associer, cliquez sur le menu déroulant à côté des servomoteurs puis choisissez la forme ou la couleur à associer.



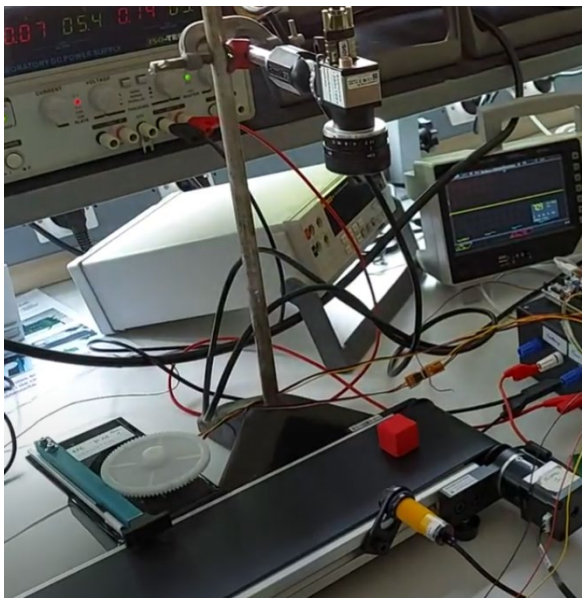
- Une fois ces réglages effectués, vous pouvez cliquer sur le bouton **start** qui va mettre en marche le convoyeur. Vous arrivez ainsi sur cette 3ème page qui correspond à la page de pilotage et d'avancement du tri que vous voulez effectuer.



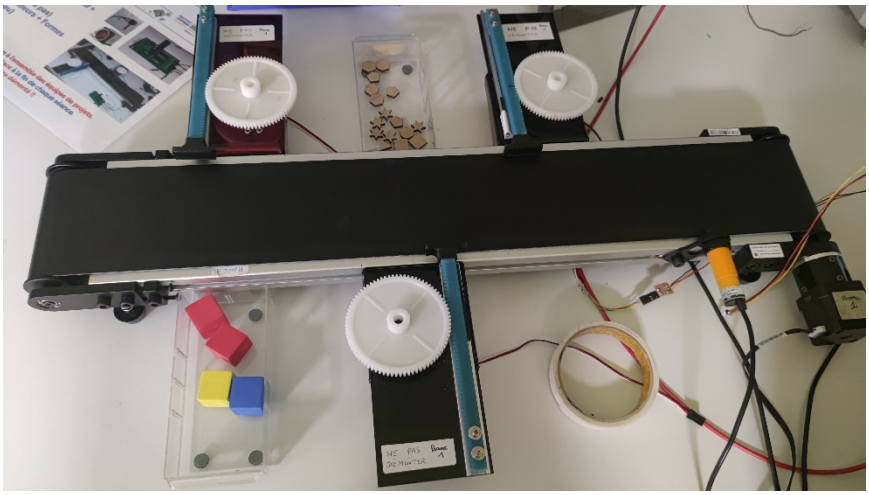
- Vous pouvez voir sur cette page la forme ou la couleur que vous avez associée à chaque servomoteur, le taux de remplissage de chaque caisson lors du tri des formes/couleurs et l'image prise par la caméra pour détecter la forme/couleur de l'objet. Un curseur permet de seuiller l'image.

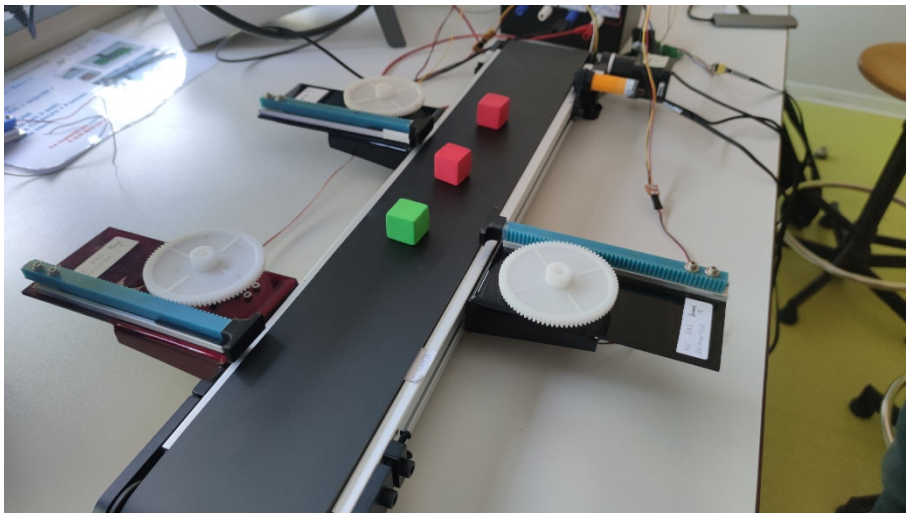
- Vous pouvez à tout moment en cas de problème, stopper le système avec le bouton **stop**. Puis, vous pouvez le relancer comme si de rien n'était avec le bouton **run** (et init réinitialise les pourcentages de remplissage des boîtes).

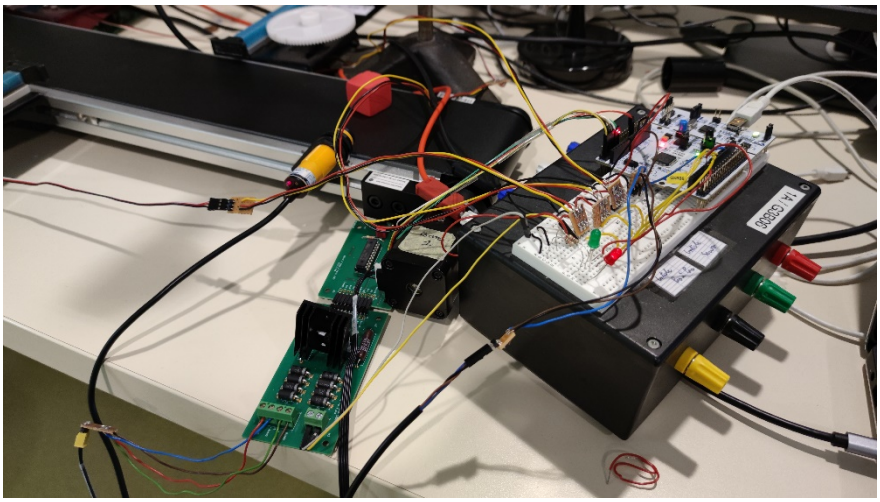
- Vous pouvez donc ensuite commencer à placer les objets pour les trier sur le convoyeur, devant la caméra, au milieu du convoyeur de préférence pour avoir une meilleure détection.



Quelques photos du système







Description – les briques du projet

Schéma bloc de fonctionnement du système :

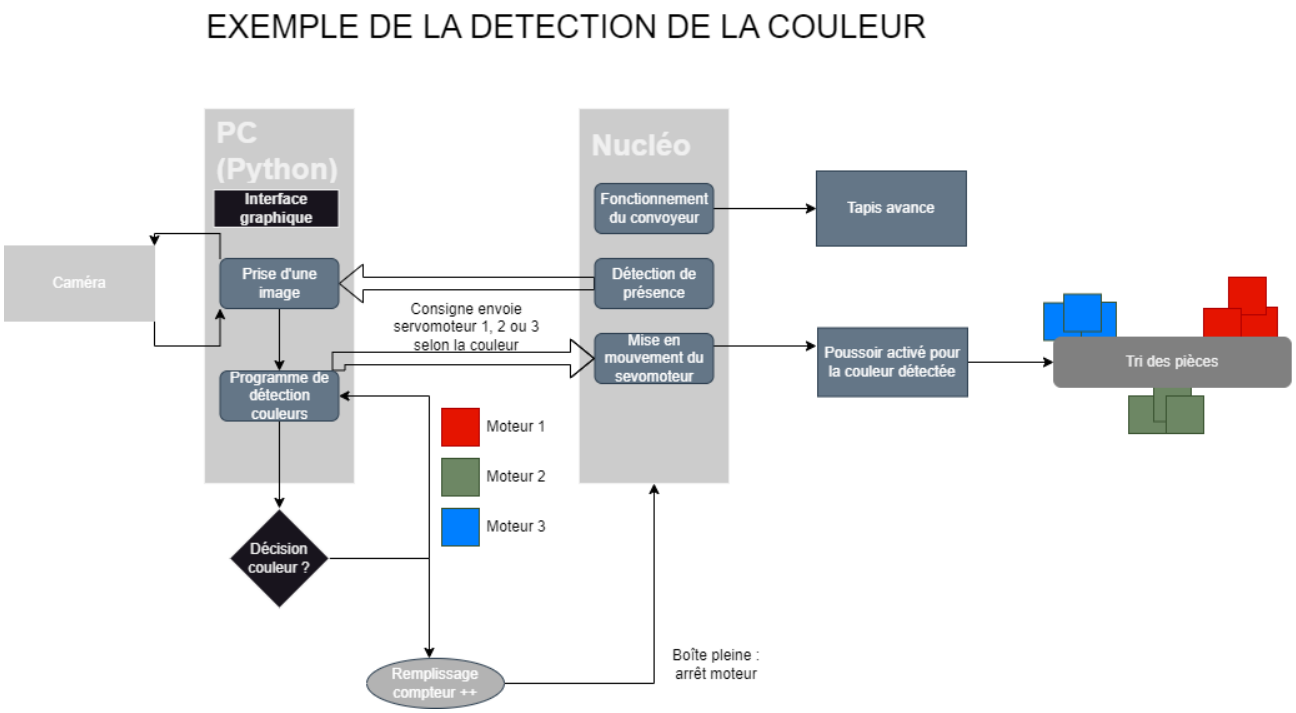


Schéma électronique et de communication du système :

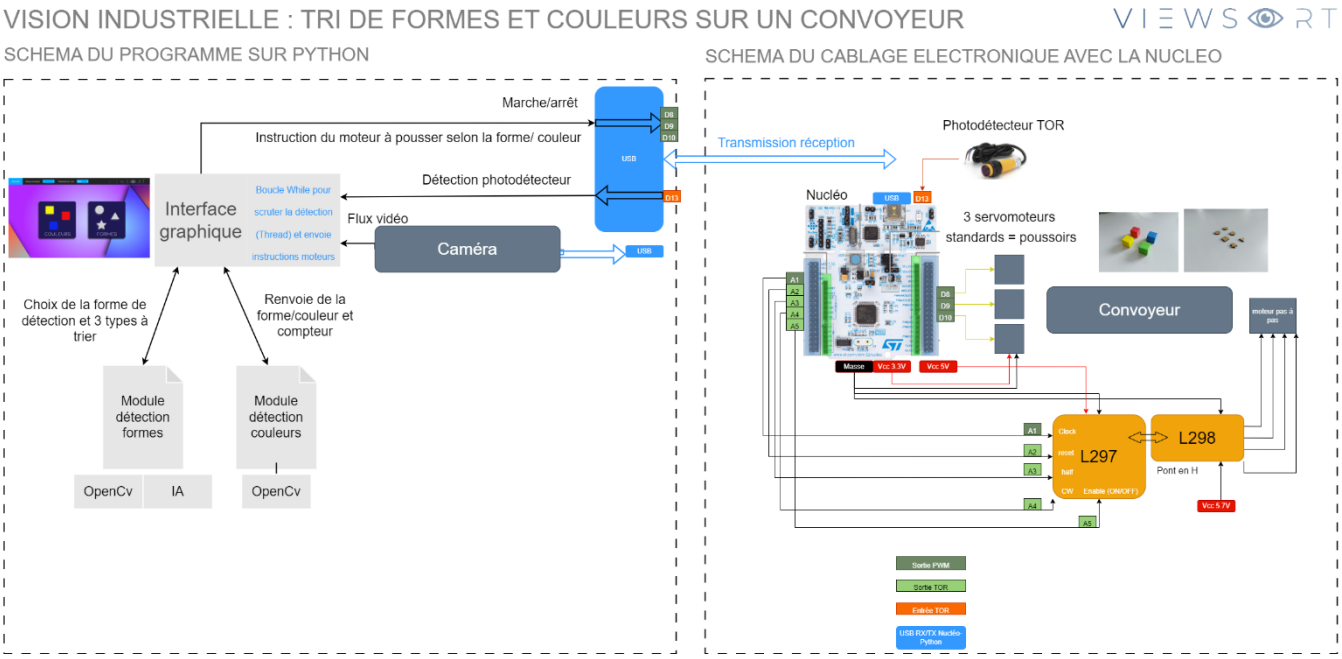
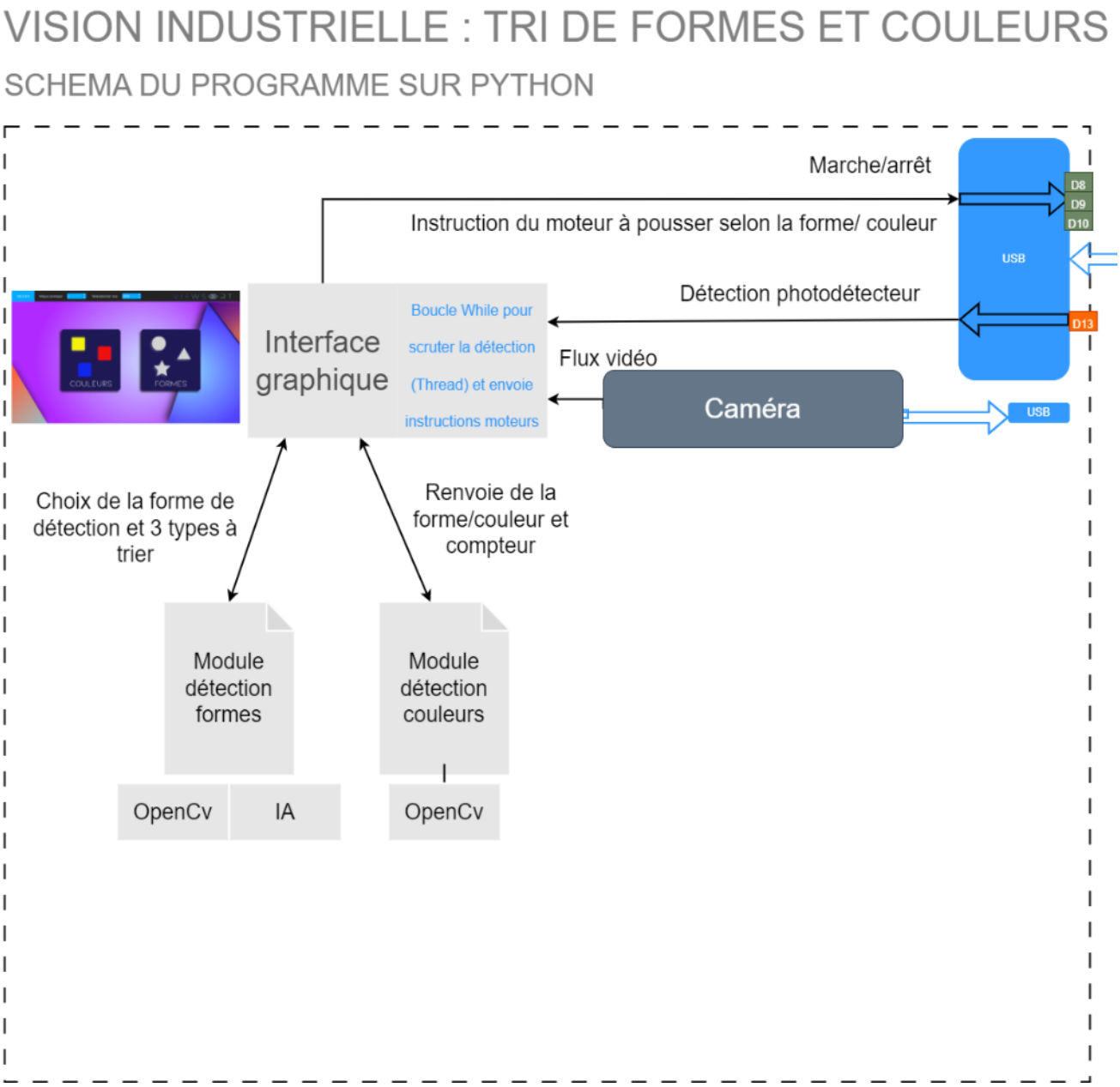


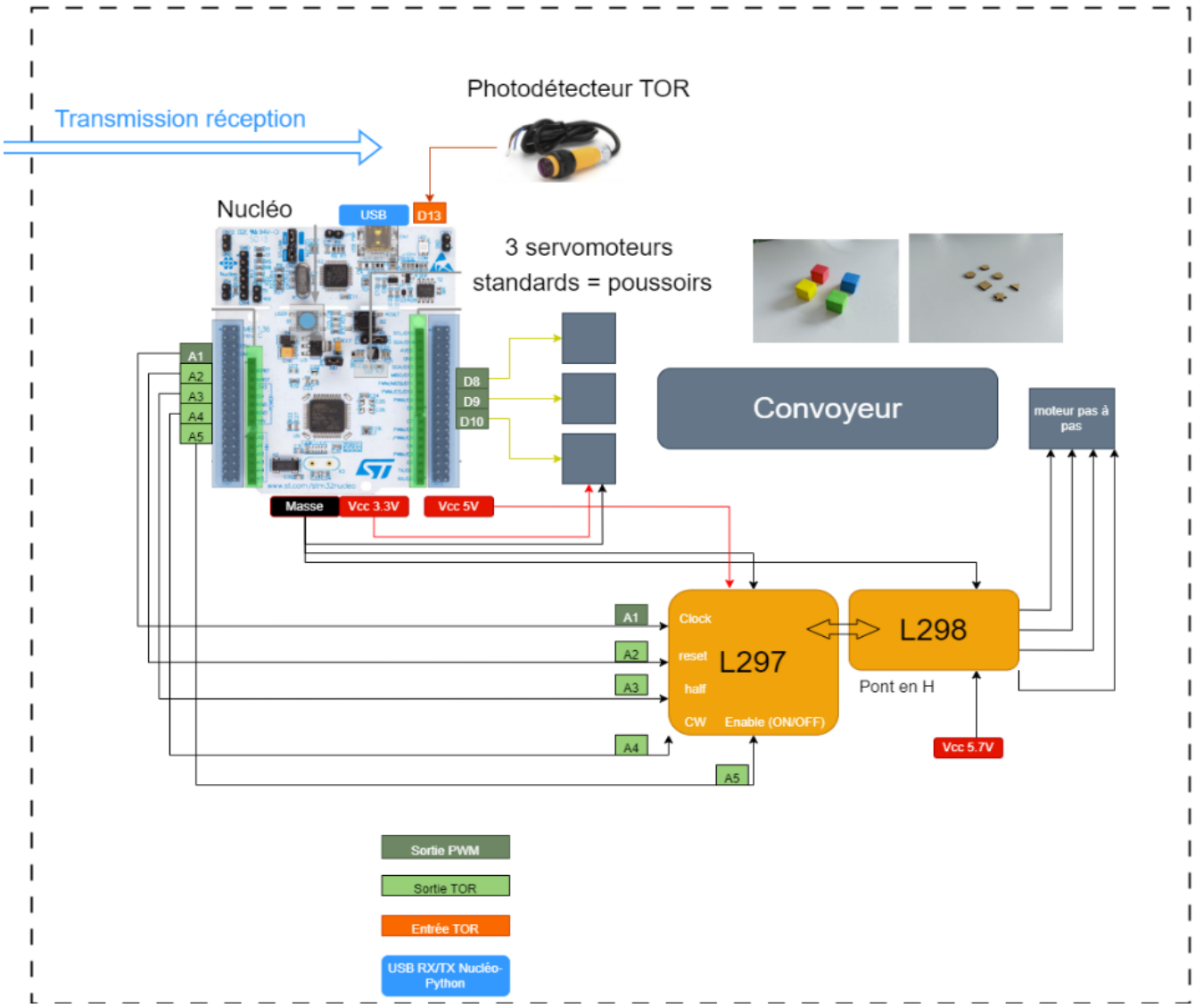
Schéma bloc de communication entre la partie électrique et programmation



SUR UN CONVOYEUR

VIEWSRT

SCHEMA DU CABLAGE ELECTRONIQUE AVEC LA NUCLEO



Le banc industriel, pilotage avec la Nucléo :

Le moteur pas à pas

Objectif : Fonctionnement et branchement du moteur pas à pas

Le moteur pas à pas permet **de faire fonctionner le tapis du convoyeur**. Le moteur doit être alimenté en 5,7V donc la carte Nucléo ne suffit pas puisqu'elle délivre au maximum 3,3V. On utilise pour alimenter le moteur une alimentation externe, mais le moteur est commandé avec la carte Nucléo.

Le moteur pas à pas est constitué d'un rotor (en rotation) et d'un stator (fixe). Le rotor est constitué d'un aimant qui va pouvoir être orienté selon la polarisation qu'on implique au niveau du stator. Le stator est constitué de 4 enroulements statoriques qui vont par paire :

- L'enroulement A et B sont ensemble
- L'enroulement C et D également

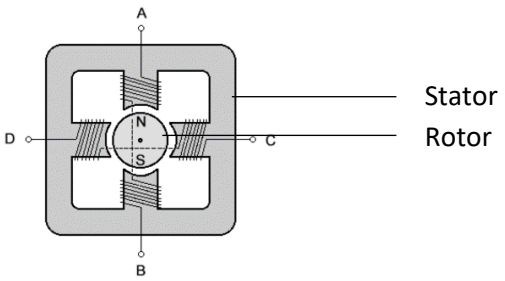


Figure 1 : Moteur bipolaire pas à pas (source : https://www.positron-libre.com/)

C'est pourquoi le moteur utilisé à 4 fils reliés aux enroulements statoriques :

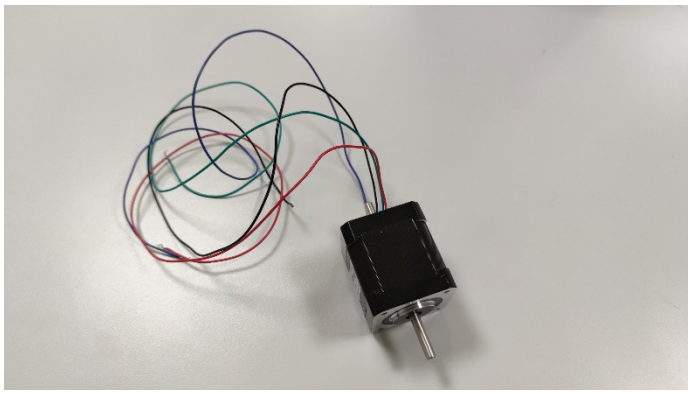


Figure 2 : Moteur pas à pas et les 4 fils pour faire avancer le moteur d'un pas

Donc en impliquant un courant entrant de A vers B et de C vers D, selon les enroulements, A et C deviennent des pôles Sud et B et D des pôles Nord, car le champ magnétique B est formé orthogonalement au courant. Le rotor est alors attiré en une position pour aligner son pôle nord aux pôles sud créés et inversement.

Cependant, une fois les pôles alignés, le rotor n'a effectué qu'un pas. Pour faire tourner le moteur, il faut réaliser à une fréquence rapide (dans notre cas 1ms), plusieurs pas. Il faut donc alterner les courants entrants et sortants pour inverser les pôles du stator :

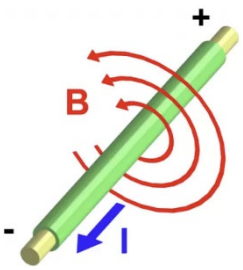


Figure 4 : champ magnétique créé en présence d'un courant (source : https://lenergie-solaire.net)

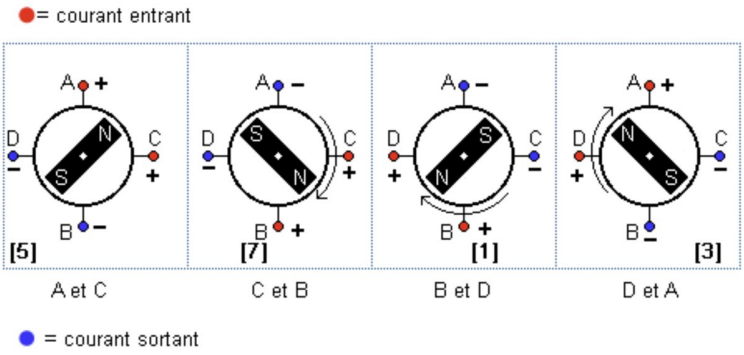


Figure 3 : Fonctionnement du moteur pas à pas en pas entier (source : https://www.positron-libre.com/)

Pour commuter le passage du courant et faire avancer « pas à pas » le moteur, on doit donc intégrer un pont en H. C'est le rôle du composant L298 constitué de 6 entrées (cf. Figure 5).

On doit relier les 4 fils du moteur pas à pas aux entrées de la carte de puissance L298 qui forme le pont en H :

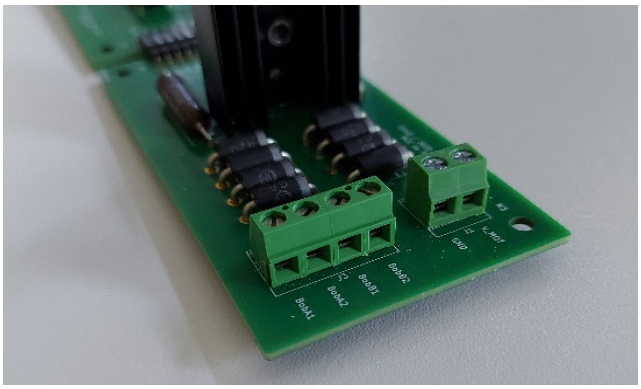


Figure 5 : Module L298 (pont en H et adapteur de puissance)

- **BobA1**, **BobA2** : Entrée de commande de la paire d'enroulement A (enroulements A, B de la figure 3)
- **BobB1**, **BobB2** : Entrée de commande de la paire d'enroulement B (enroulements C, D de la figure 3)

Donc, pour savoir quels enroulements, sont ensemble on peut prendre un fil et relier deux enroulements ensembles. Si le tapis présente une résistance alors, les 2 enroulements sont ensemble et font partie de l'ensemble « A » par exemple. On récapitule la schématisation du moteur pas à pas pour l'adapter au branchement vers le L298 :

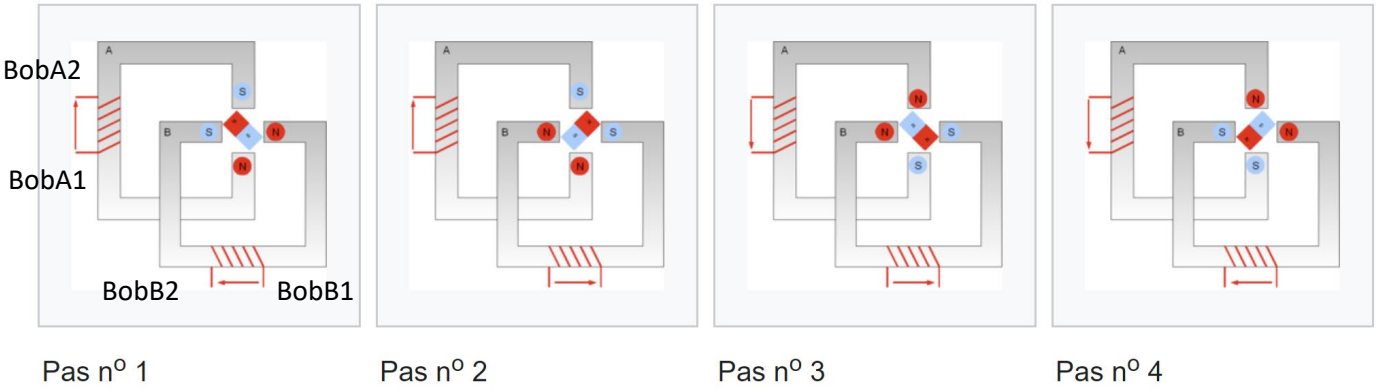


Figure 6 : Paire de bobines du moteur pas à pas et fonctionnement en pas entier (Source : Wikipedia)

Pour alimenter le moteur, on doit relier en entrée de la carte L298 :

- **Vin = 5,7V** : Tension d'entrée qui permet le fonctionnement optimal du moteur obtenu par une source DC externe.
- **GND** : la masse à la nucléo

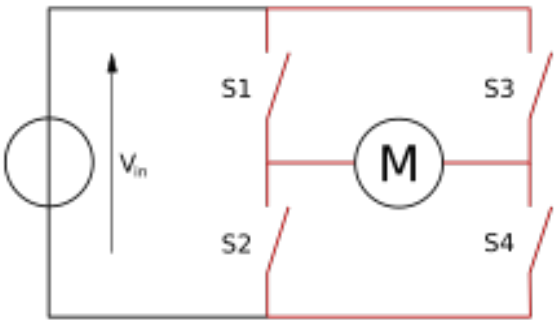


Figure 7 : Pont en H pour inverser le sens de rotation du moteur (source : Wikipédia)

Le pont en H permet d'alterner les courants dans le sens positif ou négatif pour pouvoir inverser le sens de rotation du moteur en l'alimentant en continu. Si S1 et S4 sont fermés et S2 et S3 ouverts alors le courant DC traverse le moteur pour le faire tourner dans le sens positif. Si S3 et S2 sont fermés et S3 et S4 ouverts alors le courant est parcouru dans le moteur dans l'autre sens, le moteur tourna alors dans le sens inverse.

Pilotage du moteur pas à pas :

Ensuite, il faut pouvoir piloter avec une Nucléo le moteur pas à pas. Au L298, on ajoute le module L297 qui est une carte de pilotage adaptée à la Nucléo dotée des entrées :

- **5V** : Power Input / Tension de 5V continue pour l'alimentation apportée par la Nucléo.

- **GND** : Ground / La masse du Nucléo.

- **CLOCK** : Digital Input / Entrée du signal d'horloge permettant de contrôler la vitesse de rotation du moteur pas à pas. Le signal d'horloge envoyé est de période T=1ms et de rapport cyclique 50%.

- **CW** : Digital Input / Entrée TOR permettant de contrôler le sens de rotation du moteur (commutation du pont en H)

- **ENABLE** : Digital Input / Entrée TOR permettant d'activer à 1 la mise en rotation du moteur : ON = enable à 1 et OFF = enable à 0.

- **RESET** : Digital Input / Entrée TOR permettant de réinitialiser la position initiale du moteur.

  ⚠ Pour que le moteur fonctionne on doit initialiser le reset à 1 avant le démarrage.

- **HALF** : Digital Input / Entrée TOR permettant de choisir entre un mode en pas entier ou en demi pas.

- **Vref** : Analog Input / Tension de référence pour la limitation en courant (pas utilisée)

- **SenseA / SenseB** : Analog Output / Mesure du courant passant dans les ponts A et B (via une résistance – voir étage de puissance) (pas utilisée)

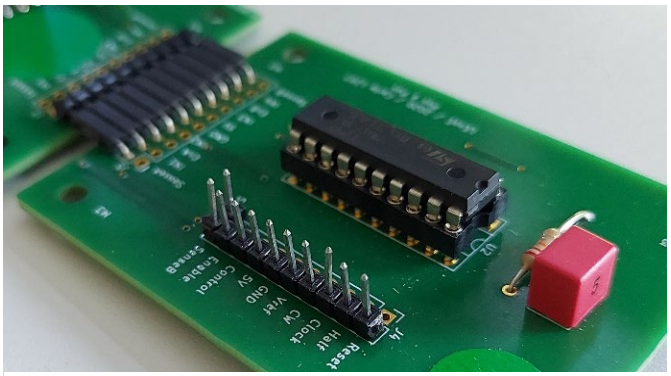


Figure 8 : Module L297 pour commander le moteur pas à pas depuis la carte Nucléo

On relie de cette manière le module L297 et L298 :

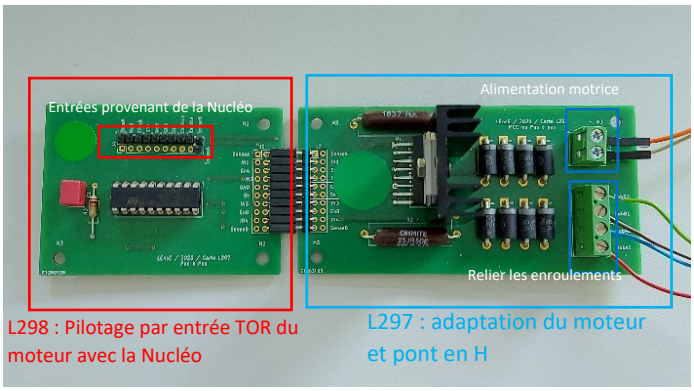


Figure 9 : Assemblage de la partie commande du moteur.

On peut ensuite directement relier le moteur :

Câble du moteur pas à pas du convoyeur relié au L297

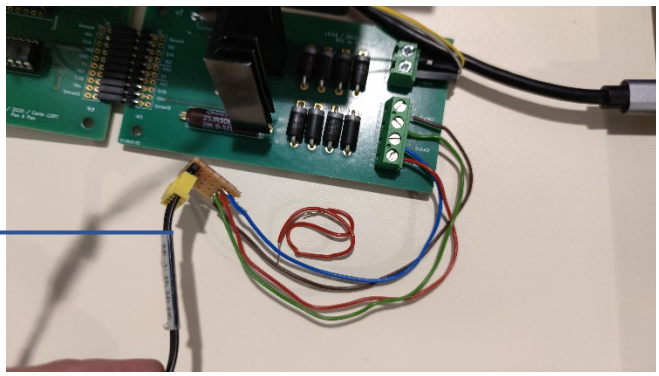


Figure 10 : Branchement du convoyeur

Au niveau de la Nucléo, on utilise la partie gauche pour piloter le moteur pas à pas, où il faut placer 6 entrées à relier au module de commande L297 :

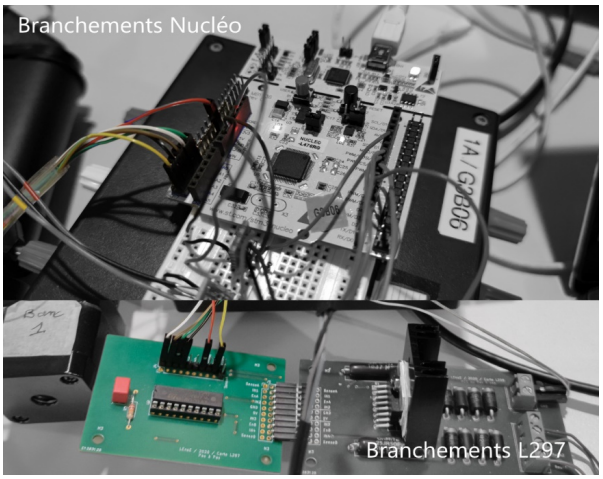


Figure 11 : Branchements entre le L297 et la Nucléo

Il suffit de relier la partie alimentation en reliant les 5,6V-5,9V et la masse au module L298 :

Figure 12 : Alimentation du moteur pas à pas

Enfin la Nucléo est un microcontrôleur doté d'entrées et sorties (TOR, PWM…) permettant de commander par exemple le moteur, en envoyant des signaux numériques convertis en signaux électriques au module L297 :

Nucléo	L297	couleur	Utilisation
A1	clock	vert	signal d'horloge en sortie PWM de période 1ms pour changer le pas (fréquence du moteur)
A2	reset	blanc	À 1 en initialisation
A3	half	marron	pas entier = 1 / demi pas=0
A4	CW	noir	sens horaire / anti-horaire
A5	enable	jaune	ON=1 / OFF=0
5V	5V	rouge	Alimentation pour le contrôle

Initialisation des broches avant le main :

```
//MOTEUR
// Alimentation en 5,7V continu sur le L298, un courant de 0,7-0,8A est créé
PwmOut clock_pwm(A1);
DigitalOut reset(A2);
DigitalOut half(A3);
DigitalOut sens_clockwise(A4);
DigitalOut enable(A5);
```

Envoie d'un signal PWM de période 1ms et de rapport cyclique 50% :

```
int main()
{
    //MOTEUR PAS à PAS
    clock_pwm.period_ms(1);
    clock_pwm.write(0.5);

    reset=1; //Initialisation obligatoire
    half=1;
    sens_clockwise=1;//1 sens horaire 0 sens anti-horaire
    enable = 0;
    while(1){

    }
}
```

Allumer la source de tension et le convoyeur tourne !!!

Tests de validation :

⚠ Pour que le moteur fonctionne on doit initialiser le reset à 1 avant le démarrage.

Si le moteur vibre, vérifier les branchements (faux contact) au niveau des 4 fils du moteur, ou bien sur l'alimentation augmenter légèrement la tension. Vérifier dans le cas où il ne tourne pas en un cycle entier, l'association des bobines entre les 4 fils. S'il ne tourne pas, on vérifie également à vide, la présence d'un courant au niveau de la source de tension. Si le courant est vide, cela signifie que les 2 paires de fils ne vont pas ensemble. La période de 1ms du signal qui « rythme » le moteur a été trouvée expérimentalement pour que le moteur ne saccade pas.

Dans d'autres cas, on vérifie, sur chaque branchement le signal envoyé/reçu à l'oscilloscope.

Les servomoteurs

Objectif : faire fonctionner les poussoirs avec un servomoteur :

Les servomoteurs permettent le fonctionnement, des poussoirs qui permettent de pousser les cubes dans les boîtes. Les servomoteurs sont constitués de 3 fils :

Figure 13 : Broches d'un servomoteur (Source : Lense)

- Rouge : **Vcc** = l'alimentation de 3,3V de la Nucléo
- Noir : **GND** = la masse
- Jaune/Blanc : **CMD** = la broche de contrôle en entrée PWM

L'objectif est **de faire avancer et reculer les poussoirs** :

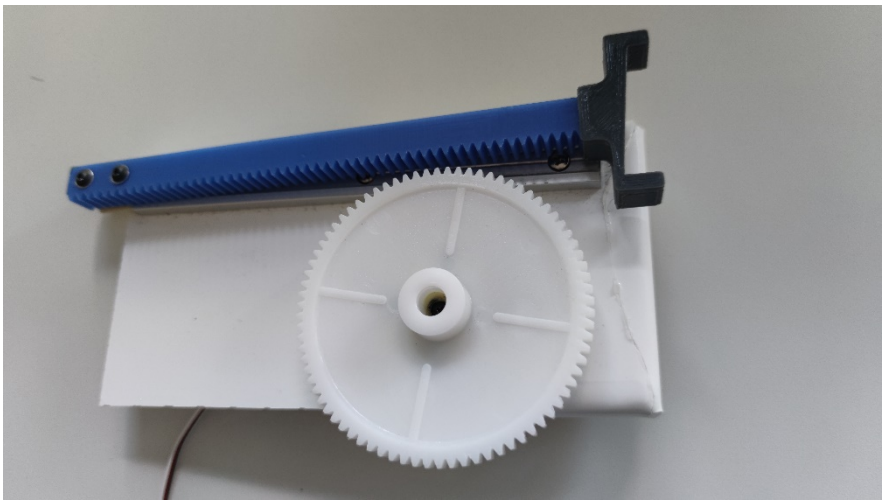
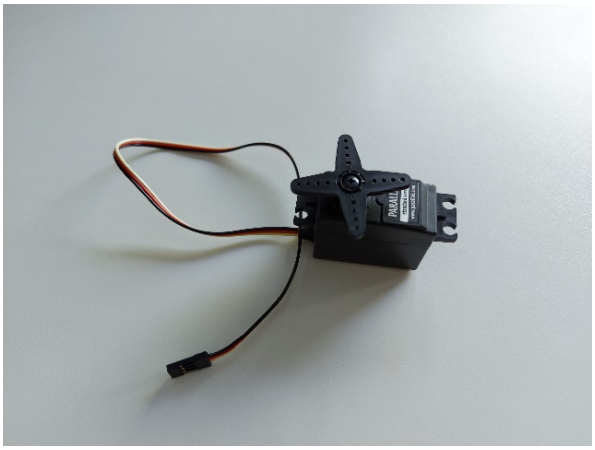


Figure 14 : Poussoir pour faire tomber les cubes de couleurs

Le contrôle des poussoirs est effectué par un servomoteur « standard ».



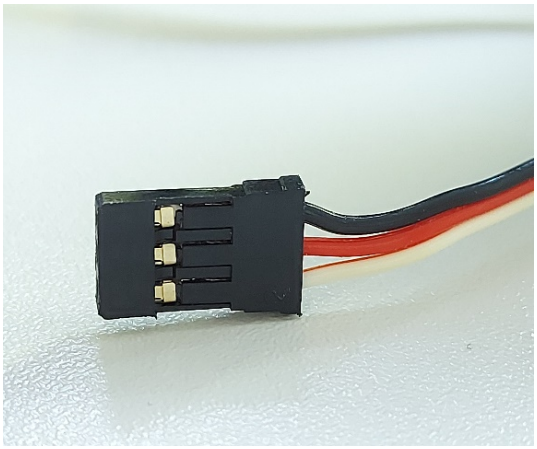


Figure 15 : Servomoteur standard

Contrôle des servomoteurs :

Le servomoteur est **pilotable en angle** en faisant varier **le rapport cyclique du signal**.

La position angulaire de 0° correspond à un temps haut de 1,5 ms du signal (par rapport à 20 ms de période).

Puis pour un temps > 1ms on fait tourner à droite le moteur et <1ms on le fait tourner à gauche.

La commande permet d'envoyer une valeur entière de ce temps haut pour piloter en angle le servomoteur pour pousser le poussoir ou le faire revenir en position initiale.

Voici le fonctionnement simple d'un poussoir en utilisant un bouton qui permet de l'actionner et lorqu'il est relâcher de revenir en position initiale. Comme pour le moteur pas à pas, la partie électronique est pilotée avec la Nucléo en C++, programme compilé avec Keil Studio :

```
//SERVO
// Delivre 3,3V en PWM
PwmOut servo_mot(D5);
DigitalIn bouton_servo(D3) ;

//SERVOMOTEUR
servo_mot.period_ms(10);         // Initialisation période
int r = 0;


while(1)
{
    if(bouton_servo ==1)
        {
            servo_mot.pulsewidth_us(2200); // On pousse le poussoir
            r = 1;
        }
    if(r == 1 && bouton_servo == 0)
    {
        servo_mot.pulsewidth_us(1000); // on fait revenir le poussoir en position 0
    }
}
```

Figure 17 : Pilotage d'un servomoteur (poussoir) avec un bouton

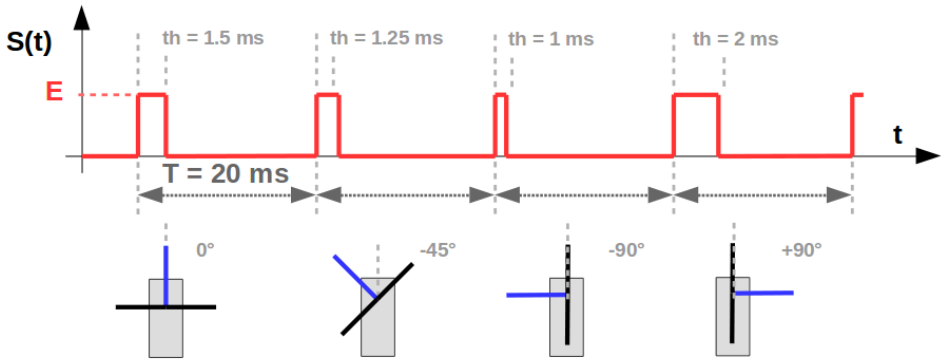


Figure 16 : Contrôle du servomoteur en angle avec le rapport cyclique (source : Lense)

Cependant, la difficulté va venir lorsque l'on ne doit pas piloter un, mais plusieurs servomoteurs.

Le pilotage Nucléo en C++

La Nucléo est pilotée en C++ pour notre projet. Nous utilisons le logiciel **Keil studio** pour compiler le programme pour le donner à la Nucléo.

Initialisation :

On commence par attribuer à chaque entrée/sortie une broche de la Nucléo :

```
//SERVO 1
// Delivre 3,3V en PWM
PwmOut servo_mot1(D8);

//SERVO 2
// Delivre 3,3V en PWM
PwmOut servo_mot2(D9);

//SERVO 3
// Delivre 3,3V en PWM
PwmOut servo_mot3(D10);
```

```
//Test envoi données leds
DigitalOut led(D6);
DigitalOut led_2(D5);
```

```
DigitalIn detecteur(D13);
```

Nous utilisons une boucle **while** globale dans notre main qui va nous permettre de faire tourner en continu notre programme. Avant cette boucle **while**, on place dans le main toute les initialisations (servomoteurs, moteur pas à pas, connexion avec le PC, ticker) :

```
int main()
{
    //MOTEUR PAS à PAS
    clock_pwm.period_ms(1);
    clock_pwm.write(0.5);

    reset=1; //Initialisation obligatoire
    half=1;
    sens_clockwise=1;//1 sens horaire 0 sens anti-horaire
    enable = 0;
    int a = 0;

    //SERVOMOTEURS
    int i, time = 1500;
    servo_mot1.period_ms(10);         // Initialisation période
    servo_mot2.period_ms(10);         // Initialisation période
    servo_mot3.period_ms(10);         // Initialisation période

    //CONNECTION PYTHON
    my_pc.baud(115200);
    my_pc.attach(&reception_pc_nucleo, UnbufferedSerial::RxIrq);

    // initialisation du ticker pour la décrementation
    toggle_ticker.attach(&decrementation, DET_T);
```

Problématique de la détection simultanée :

Un de nos objectifs initial était de pouvoir avoir plusieurs objets sur le convoyeur en simultanée et donc ne pas se restreindre à poser un objet, attendre qu'il soit poussé, puis poser l'objet suivant.

Cette problématique est en réalité plus complexe qu'elle en a l'air. On ne peut pas simplement pour chaque objet compter le temps qu'il va mettre à parcourir la distance du capteur au moteur puis le pousser.

Une première méthode naïve consistait à mettre des **thread.sleep** (temps d'attente = wait) le temps de parcours de l'objet. Or, cette fonction stop complétement le programme et a donc un impact sur les autres objets présent sur le convoyeur. Le programme ne peut être indépendant pour chaque détection : les objets suivants doivent attendre la fin du **thread.sleep** précédent, si une détection a lieu sur ce temps d'attente, elle n'est pas prise en compte.

Les tickers :

Nous avons résolu ce problème en ajoutant à notre programme des **tickers.**

Ils permettent de créer une horloge interne pour le programme sans prendre en compte les autres instructions du programme.

Ainsi, en créant ce que l'on peut appeler une « **décrémentation** » périodique, on peut remplacer les **thread.sleep** en comptant le temps que va mettre l'objet à arriver au moteur sans restreindre le programme. On utilise alors la routine d'interruption indépendante du main. Cette méthode permet de « garder en mémoire » les objets présents sur le convoyeur.

Fonctionnement global :

Voici le déroulement global du programme, de la détection jusqu'à l'action du moteur :

- Tout d'abord l'objet passe devant le détecteur donc on rentre dans cette condition **if**. Le programme envoie donc l'instruction **'d'** au PC, qui permet de prendre une photo avec la caméra pour déterminer la caractéristique (couleur ou forme) de l'objet.

```
if(detecteur == 1 && delay==0) // si le capteur a detecté un objet et si le délai de detection est à
{
    char message = 'd';
    my_pc.write(&message,1);// on envoie l'info au pc comme quoi on a eu une detection donc il faut prendre une photo
    delay = 100;
}
```

Remarque : la variable **delay** permet de ne rentrer qu'une fois dans la condition **if** pour un objet. En effet le délai de détection du détecteur est bien plus rapide que la

vitesse de passage de l'objet devant le détecteur. Sans **delay**, un objet serait détecté plusieurs fois. Ici, un passage d'un objet, équivaut à une unique détection.

- Ensuite, une fois que le PC à déterminé la caractéristique de l'objet, il renvoie une information à la Nucléo, pour savoir quel moteur doit être actionné pour cet objet.

```
void reception_pc_nucleo(void)
{
    my_pc.read(&data, 1);     // get the received byte

    // Information sur le servomoteur qu'il faut "activer" pour un objet
    if(data == '1')
    {
        objet_detecte = 1;
    }
    if(data == '2')
    {
        objet_detecte = 2;
    }
    if(data == '3')
    {
        objet_detecte = 3;
    }
```

Le programme va donc rentrer dans la condition **if** ci-dessous.

```
if(objet_detecte != 0) // si le pc a envoyé l'info qu'un objet a été detecté
{
    int etat_libre = check_etat_libre();  // On a trouvé une case libre i dans n_liste : aucun objet en cours
    objet_liste[etat_libre]=2;

    init_compteur(etat_libre);
    objet_detecte = 0; // on remet l'état à 0 pour ne passer qu'une fois dans la condition if
}
```

C'est là que toute la magie des listes va faire son apparition. On peut se demander à première vu comment faire pour garder en mémoire les objets sur le convoyeur s'il y en a plusieurs.

La méthode la plus optimisée (et la plus classe) consiste à créer des listes qui s'auto-actualisent en fonction de l'arrivée ou de la sortie des objets sur le convoyeur.

On initialise donc 2 listes (qui sont la partie centrale du code) :

```
//Ticker 1
// On établit une liste de longueur finie qui contient les états détectés en cours et leur temps qui décroit d
int objet_liste[NB_OBJET]={0}; // [1, 2, 0, 0, 1, 3...] valeur pour pousser moteur 1 moteur 2 ou 3. La valeur
int n_liste[NB_OBJET]={0}; // [300, 500, 0, 0, 500...] : liste des compteurs de temps qui décroissent à 0.
// contient les temps N associés aux états occupés détecté. A la détection la valeur est N puis décroit toutes
// A 0 l'état est libre
```

objet_liste : renseigne sur quel moteur doit pousser l'objet (1 = moteur 1, 2 = moteur 2, 3 = moteur 3)

n_liste : renseigne sur le nombre de passage dans la boucle **while** que doit faire le programme avant d'activer le moteur (sur le temps de parcours de l'objet sur le convoyeur en quelque sorte : contient le temps en unité d'attente 10ms avant de pousser les moteurs : 300 = 3s)

Il faut bien comprendre qu'un objet quelconque arrivant sur le convoyeur va se voir associer une case vide n pour les 2 listes qui vont donc permettre de stocker ses informations.

On peut se demander maintenant si les listes doivent être infinie ? En effet comment faire si le convoyeur fonctionne sur une période de temps très longue et qu'il trie un grand nombre d'objet ? Faut-il un grand nombre de case dans les listes pour stocker tous les objets ?

En réalité la réponse est assez subtile. Comme dit précédemment, les listes s'auto-actualisent. A chaque passage dans la boucle **while**, chaque case de la liste **n_objet** est décrémentée de 1 avec le ticker :

```
void decrementation()  //décrémente, le compteur de N à 0
{
    if(run == 1)
    {
        int i = 0;
        for(i=0; i<NB_OBJET; i++)
        {
            if(n_liste[i] != 0) // && objet_liste[i]!=0 pas besoin car pas besoin de cette vérif car la décrementation arrive à 0 automa
            {
                n_liste[i]--;
            }
            if(n_retour_liste[i]!=0) // && retour_liste[i]!=0 pas besoin car pas besoin de cette vérif car la décrementation arrive à 0
            {
                n_retour_liste[i]--;
            }
        }
        // delai de détection au passage du cube, = temps de passage pour ne pas continuellement détecter quand le cube est devant
        if(delay != 0)
            delay --;
    }
}
```

Quand la case arrive à 0, c'est le signal pour pousser le servomoteur (car l'objet est arrivé à sa position). Puis on arrête de décrémenter pour ne pas aller dans les négatifs.

De même pour la liste **objet_liste**, on remet la case à 0 quand l'objet a été poussé. En conclusion, quand l'objet est poussé, ces informations sont effacées dans les 2 listes. C'est pourquoi il est possible de mettre l'objet suivant sur cette case. Lorsque la décrémentation (temps d'attente avant de pousser) est terminée, la case de la liste se libère pour un autre objet.

On a donc pas besoin d'une liste infinie pour une infinité d'objet, mais seulement une liste plus grande que le nombre d'objet maximal qu'on peut mettre sur le convoyeur en simultané. On réutilise les cases des objets poussés pour les objets arrivant sur le convoyeur.

Cette détection case vide se fait avec la fonction **check_etat_libre()** qui permet de renseigner une case vide à attribuer pour l'objet qui vient d'être détecté :

```
int check_etat_libre() // regarde continuement la case la plus proche dans la liste qui est vide pour y associer le prochain objet
{
    int i=0; // indice de check d'un état libre
    while(objet_liste[i]!=0)
    {
        i++;
    }
    return i;
}
```

Puis on initialise le compteur de la liste **n_liste** pour l'objet avec la fonction **init_compteur()** pour remettre le temps d'attente associé au moteur 1, 2 ou 3 (qui va se décrémenter à nouveau par la suite) :

```
// 1er ticker déclenchement servo
void init_compteur(int etat_libre) // initialise un compteur quand un objet est detecté
{

    if(objet_liste[etat_libre]==1) // si l'objet detecté doit être poussé par le servo 1
    {
        n_liste[etat_libre] = N_1;
    }

    if(objet_liste[etat_libre]==2) // si l'objet detecté doit être poussé par le servo 2
    {
        n_liste[etat_libre] = N_2;
    }

    if(objet_liste[etat_libre]==3) // si l'objet detecté doit être poussé par le servo 3
    {
        n_liste[etat_libre] = N_3;
    }

}
```

Les compteurs sont plus ou moins grand suivant la distance à laquelle les servomoteurs sont du détecteur.

```
for(i=0 ; i<NB_OBJET; i++)
{
    if(n_liste[i]==0 && objet_liste[i]!=0)
    {

        if(objet_liste[i] == 1)
        {
            servo_mot1.pulsewidth_us(2400);
            init_compteur_retour(i);
            objet_liste[i] = 0;
            retour_liste[i] = 1;
        }

        if(objet_liste[i] == 2)
        {
            servo_mot2.pulsewidth_us(2400);
            init_compteur_retour(i);
            objet_liste[i] = 0;
            retour_liste[i] = 2;
        }

        if(objet_liste[i] == 3)
        {
            servo_mot3.pulsewidth_us(2400);
            init_compteur_retour(i);
            objet_liste[i] = 0;
            retour_liste[i] = 3;

        }
    }
}
```

- Enfin, on active en continue dans la boucle **while**, une boucle **for** qui parcourt la liste **n_liste**. Si une case est vide dans **n_liste** et qu'on a bien un objet, alors il faut pousser le servomoteur qui est associé à l'objet.

C'est à ce moment qu'on réinitialise la case correspondante à l'objet dans la liste **objet_liste**, une fois l'objet poussé.

Remarque : pour le retour des servomoteurs à leur position initiale, on réutilise exactement la même méthode que pour pousser un objet vers l'avant avec des tickers. On utilise des listes qui gardent en mémoire le fait que le servomoteur doit revenir en arrière à cet instant après avoir poussé l'objet.

```
//Ticker 2
// Retour des moteurs à leur position initiale
int retour_liste[NB_OBJET]={0};
int n_retour_liste[NB_OBJET]={0}; // [300, 500, 0, 0, 500...] : liste des compteurs de temps qui décroissent à 0.
```

```
// 2eme ticker retour
void init_compteur_retour(int i) // initialise un compteur quand le servo commence à pousser pour qu'il puisse revenir par la suite
{
    n_retour_liste[i] = N_retour;
}
```

```
for(i=0 ; i<NB_OBJET; i++)
{
    if(n_retour_liste[i]==0 && retour_liste[i]!=0)
    {
        if(retour_liste[i] == 1)
        {
            servo_mot1.pulsewidth_us(1000);
            retour_liste[i] = 0;
        }

        if(retour_liste[i] == 2)
        {
            servo_mot2.pulsewidth_us(1000);
            retour_liste[i] = 0;
        }

        if(retour_liste[i] == 3)
        {
            servo_mot3.pulsewidth_us(1000);
            retour_liste[i] = 0;
        }
    }
```

La détection caméra de couleurs et formes :

La détection de formes et de couleurs se fait avec OpenCv en Python de manière similaire.

```
import cv2
```

On travaille d'abord avec la WebCam, puis on intégrera la caméra de UeyeCockpit.

Pour récupérer le flux continu on importe OpenCv, puis on récupère le canal de la WebCam avec **VideoCapture(0, cv2.CAP_SHOW).** Dans une boucle **while**, on lit alors l'image (frame) à intervalle régulier (non vu par l'œil humain) avec **video.read().** On affiche l'image avec la méthode **imshow(''Frame'', img).** Il ne faut pas oublier de tout détruire ensuite à la sortie de la boucle **while**.

```
import cv2


if __name__ == "__main__":


    # Lecture de l'image
    # Récuperer la Webcam à 0
    video = cv2.VideoCapture(0, cv2.CAP_DSHOW)

    #FLux vidéo continu
    while True:
        ret, frame = video.read() #retour et récupérer l'image
        cv2.imshow("Acquisition", frame) # afficher l'image


        if cv2.waitKey(1)==ord('q'): #Pour arrêter la vidéo en appuyant sur 'q'
            break


    video.release()
    cv2.destroyAllWindows() # Ne pas oublier de tout détruire en sortie de la boucle while
```

Le programme de la détection de couleurs

La détection de couleurs est appelée en une ligne en lui passant pour paramètre l'image acquise issue de la boucle continue notée frame, et remplissage qui constitue un dictionnaire qui stocke le comptage des couleurs détectées sous la forme remplissage = {« color » : int nombre}

```
remplissage = {'ROUGE': 0, 'ORANGE': 0, 'JAUNE': 0, 'VERT_CLAIR': 0,
               'VERT_FONCE' : 0, 'BLEU_CLAIR': 0, 'BLEU_FONCE': 0,
                'VIOLET': 0, "ROSE" : 0, "MARRON" : 0, "INDEFINI" : 0}
```

```
def detection couleurs(frame, remplissage):
```

La fonction renvoie en particulier la couleur détectée à l'instant où la frame est passée en paramètre et le dictionnaire remplissage est mis à jour et fait office de compteur de couleurs.

Pour cela, on doit convertir le mode BGR (Blue Green Red) de l'image acquise par OpenCv (frame) au format HSV (Hue Saturation Value) qui permet de récupérer les valeurs de teintes, saturation et de luminosité.

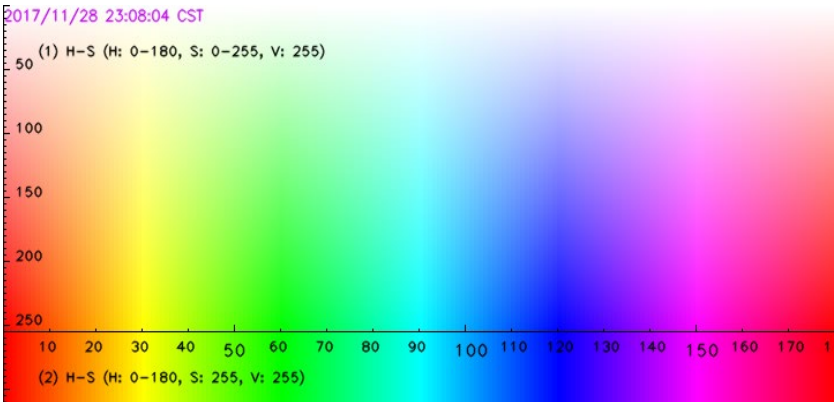


Figure 18 : Valeurs de teintes (H) et de saturation (V)

Source : https://stackoverflow.com/questions/10948589/choosing-the-correct-upper-and-lower-hsv-boundaries-for-color-detection-withcv

image_hsv[x,y] permet de récupérer la valeur [H, S, V] du pixel pointé.

Cette échelle définit la teinte dans OpenCv de 0 à 180° (cylindre de révolution sur le rouge).

On seuille les différentes couleurs selon cette échelle (en réalité, on choisit les valeurs de teinte expérimentalement avec un programme OpenCv qui renvoie les teintes pour des objets choisis).

On envoie le pixel du centre à la fonction **pick_color(pixel_center)**. Le pixel du centre est défini au centre de la figure détectée et contient la valeur [H, S, V]. On retourne alors la couleur détectée.

```python
def pick_color(pixel_center):
    """ pick_color(ndarray) -> str

     Fonction permettant de renvoyer la couleur acquise selon le pixel centrale
     de données HSV (Hue Saturation Value)


    Parameters
    ----------
    pixel_center : ndarray

    Returns
    -------
    color : str
        couleur trouvée selon la teinte/saturation/luminosite du pixel central.

    """

    hue_value = pixel_center[0] # Teinte

    saturation = pixel_center[1] # Saturation

    value=pixel_center[2] # Luminosité

    color = "Indéfini"
    if value<30:
        color = "INDEFINI"
    elif (hue_value < 45 or hue_value > 160) and saturation<160 and value<120: #10, 120, 100
        color = "MARRON"
    elif (hue_value > 140 and hue_value < 177) and saturation<90 and value > 120:
        color = "ROSE"#174

    elif hue_value < 8 :#2 - 178
        color = "ROUGE"
    elif hue_value < 18:
        color = "ORANGE"#7
    elif hue_value < 35:
        color = "JAUNE"#21
    elif hue_value < 50:
        color = "VERT_CLAIR"#44
    elif hue_value < 90:
        color = "VERT_FONCE"#87
    elif hue_value < 118:
        if saturation < 200:
            color = "BLEU_CLAIR"
        else :
            color = "BLEU_FONCE"
    elif hue_value < 140:
        color = "VIOLET"#123-135
    elif hue_value < 180: # 177, 163, 234
        color = "ROUGE"
    else:
        color="ROUGE"

    return color
```

Comment trouver alors le pixel du centre ? On réalise l'acquisition vidéo, et on va seuiller la frame acquise selon la valeur de la saturation dans l'image. Le convoyeur dispose d'un tapis noir peu saturé avec des cubes de couleurs saturés qui circulent, ce qui va permettre de ne récupérer que les cubes de couleurs dans l'image. On fixe une limite basse de détection avec **lower=np.array([5,95, 120]).** En dessous de ces valeurs le masque cache les parties non saturées ou trop claires donc non détecables avec les reflets. Un curseur de seuil permettra de remplacer S=95 par une autre valeur de saturation.

masque=cv2.inRange(image_hsv, lower, upper) # Création du masque

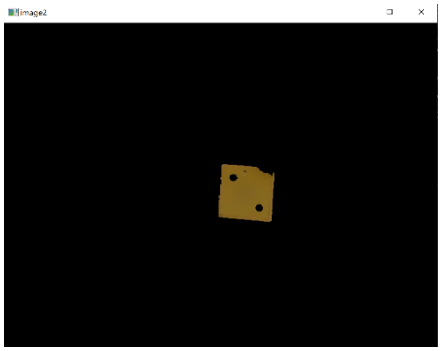


Figure 19 : Image masquée ne révélant que le carré jaune saturé

```python
# ================================================================
#         Rognage et seuillage video du convoyeur pour l'affichage
# ================================================================


#Définition d'une zone de détection
# y1 = 200
# y2 = 400
# x1 = 100
# x2 = 600
# img_convoyeur = frame[y1:y2, x1:x2]
img_convoyeur = frame

image_hsv=cv2.cvtColor(img_convoyeur, cv2.COLOR_BGR2HSV)# changement de BGR (OpenCv) à HSV
height, width, ret = image_hsv.shape

#Seuillage pour créer un masque selon la saturation de la couleur dans l'espace HSV
#frame_threshold = cv.inRange(frame_HSV, (low_H, low_S, low_V), (high_H, high_S, high_V))
lower=np.array([5,95, 120])# valeurs HSV min du seuillage de la couleur
upper=np.array([255,255,255])

lower[1]=config.current_slider # Valeur du seuillage définit par le curseur selon la saturation
masque=cv2.inRange(image_hsv, lower, upper)# création du masque qui supprime toutes les valeurs < lower
image_hsv=cv2.blur(image_hsv, (7,7)) # arrondir les contours par un filtre blur

image2=cv2.bitwise_and(img_convoyeur, img_convoyeur, mask=masque)# application du masque à l'image

cv2.imshow('Acquisition', frame)
cv2.imshow("Convoyeur", img_convoyeur)
cv2.imshow('image2', image2)
```

Figure 20 : Seuillage de l'image en saturation

L'image convertie en HSV, on applique le masque pour afficher son impact sur l'image.

Ensuite, à travers ce masque, on recherche (comme pour la détection de formes), les contours extérieurs de ce qui pourrait être un objet dans l'image avec :

elements=cv2.findContours(masque, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)[-2]

Cela renvoie une liste d'éléments qui contient toutes les coordonnées des points qui construisent la forme détectée. On approxime la forme par un cercle pour obtenir le rayon et les coordonnées centrales.

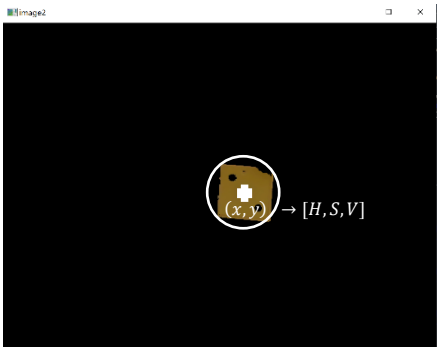


Figure 21 : Obtention du centre de la forme colorée

pixel_center = image_hsv[int(y), int(x)]
#données HSV du centre de la forme détectée

color = pick_color(pixel_center)
#récupérer la couleur du pixel du centre

remplissage[color] += 1
#Ajouter la couleur détectée au compteur de la couleur

Finalement, comme c'est le détecteur qui va engendrer la prise d'image au moment ou un cube passe devant sur le convoyeur, on simule la détection en appuyant sur le clavier sur la touche « t » ce qui affiche l'image obtenue et renvoie la couleur et le compteur à jour.

```
if cv2.waitKey(20) == ord('t'):
    #Lecture image -> remplacer par la détection de la cellule TOR
    color, remplissage = detection_couleurs(frame, remplissage)
    cv2.imshow('acquisition', frame)
```

Le carré jaune est détecté. En appuyant en continu sur la touche « t » et en déplaçant le cube, le programme ne fait pas d'erreur, où ici 416 coups de détection ont permis de trouver la couleur « jaune ».

Figure 22 : Détection de la couleur

Un autre programme permet de définir les couleurs selon les données HSV pour pouvoir changer les tolérances de teintes, saturation, luminosité.

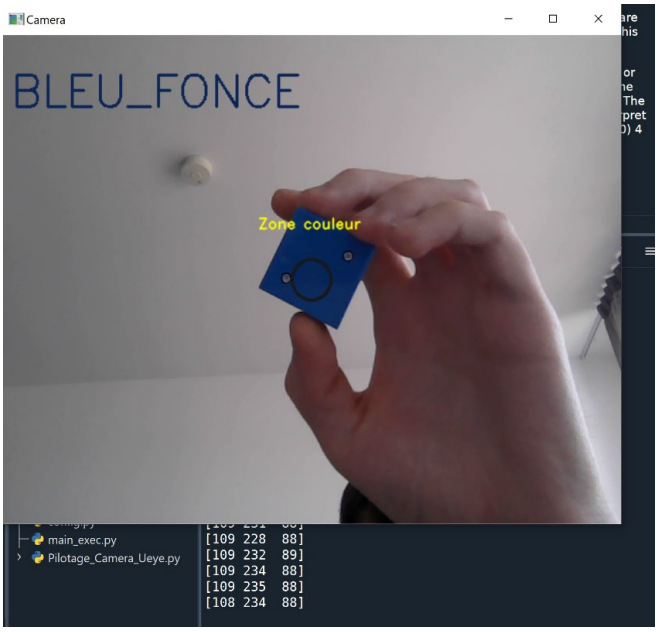


Figure 23 : Valeurs HSV qui s'affiche dans la console

Le programme de la détection de formes

Le programme de détection de formes est basé de la même manière, on envoie l'image acquise (frame) dans la fonction ci-contre :

```
def detection_formes(frame, remplissage):
```

Le dictionnaire remplissage stocke le comptage des formes détectées sous la forme : **remplissage = {« forme » : int nombre}.**

La fonction renvoie en particulier la forme détectée, à l'instant où la frame est passée en paramètre, et le dictionnaire remplissage est mis à jour et fait office de compteur de formes.

Pour cela, on ne va pas seuiller l'image de la caméra selon la saturation, mais selon la luminosité. On va passer par une image noire et blanche pour binariser et renforcer les contours.

```
shape = ""

#Définition d'une zone de détection
# y1 = 200
# y2 = 400
# x1 = 100
# x2 = 600
# img_convoyeur = frame[y1:y2, x1:x2]
img_convoyeur = frame

img_gray_conv = cv2.cvtColor(img_convoyeur, cv2.COLOR_BGR2GRAY)

img_gray_conv = cv2.blur(img_gray_conv, (4, 4))


seuil = config.current_slider

# Seuillage binaire tel que niveau_de_gris > 40 on met en blanc=255 les formes
ret, thresh = cv2.threshold(
    img_gray_conv, seuil, 255, cv2.THRESH_BINARY)


# Détection de contours, en mode = cv2.RETR_EXTERNAL  (uniquement contour extérieur sans hiérarchie détecté)
# et approximation simple
contours, ret = cv2.findContours(
    thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
#Contour est un tuple qui contient toutes les figures et leur points détectés dans une liste
```

Figure 24 : Traitement sur l'image et obtention des contours

On convertit l'image BGR en niveau de gris, on lisse l'image avec un blur puis on vient seuiller l'image en noir et blanc avec un seuil définit par un curseur dans l'interface graphique.

On détecte alors les contours selon le seuil thresh de l'image binarisée :

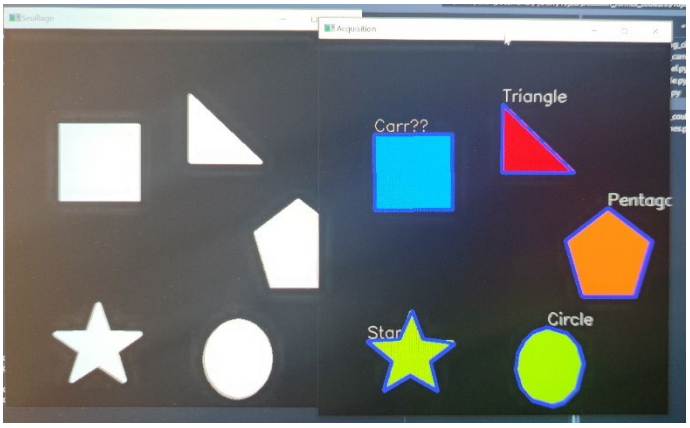
contours, ret = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

« contour » contient tous les points pour former une figure sous la forme : [[forme1], [forme2], forme[i]…] où toutes les formes[i] contiennent les coordonnées des pts (x1,y1), (x2, y2), …

```
#Parcours des figures
for c in contours:

    #Tenir en compte que des formes de tailles importantes
    area = cv2.contourArea(c)
    if area > SIZE_AREA_PX_MIN and area < SIZE_AREA_PX_MAX:

        #approximation des polygones (curves, erreur epsilon, contour fermé), pour obtenir des formes fermées
        #Epsilon est très important pour la détection de la forme
        approx = cv2.approxPolyDP(c, 0.03*cv2.arcLength(c, True), True)

        #Dessin des contours (image, contours des formes, couleur, épaisseur trait)
        cv2.drawContours(img_convoyeur, [approx], 0, (255, 0, 0), 3)

        # #Zone verte, pour simplifier le centre de gravité lors du franchissment de la ligne
        (x, y, w, h) = cv2.boundingRect(c)
        cv2.rectangle(img_convoyeur, (x, y),
                      (x+w, y+h), (0, 255, 0), 1)
        x_grav = x+int(w/2)
        y_grav = y+int(h/2)

        #Affichage centre de gravité
        cv2.circle(img_convoyeur, (x_grav, y_grav), 3, (0, 0, 255), -1)



        #alors on détecte la forme
        shape = pick_shape(approx)
        remplissage[shape] += 1  # Compteur de la forme
```

Pour toutes les formes notées « c » dans contours on calcule l'aire pour exclure les formes trop grosses (taches de lumière qui peuvent être détectées) ou trop petites (poussières). Dans le cas où la forme est adaptée on utilise :

```
approx = cv2.approxPolyDP(c, 0.03*cv2.arcLength(c, True), True)
```



La fonction permet de relier les points pour obtenir une forme fermée. Ainsi, approximation, constitue la forme a détectée et ne constitue que les points nécessaires pour reconstituer la forme, c'est-à dire pour un carré 4pts, un triangle 3pts…

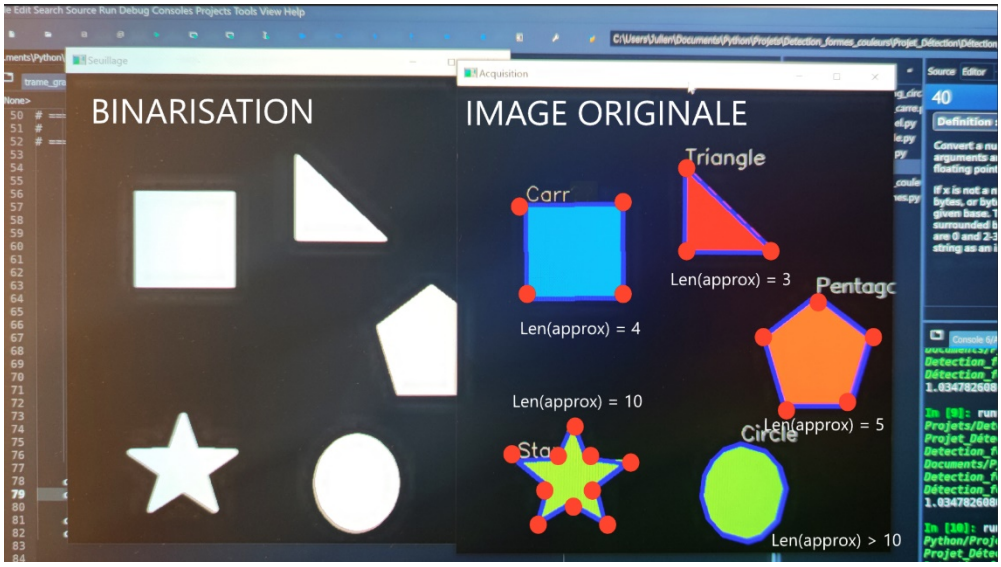


Figure 25 : Approximation des formes avec approxPolyDP

Il ne reste qu'à donner approx à la fonction **pick_shape(approx)** qui associe les points liés à l'approximation à la forme géométrique. Si approx contient 6pts, alors len(approx) = 6 et la forme est un hexagone qui dispose de 6 coins.

```
#alors on détecte la forme
shape = pick_shape(approx)
remplissage[shape] += 1  # Compteur de la forme
```

On retourne pour l'image donnée la forme « shape » sous la forme d'une str. De même le dictionnaire de remplissage des formes est incrémenté.

```
pick_shape(approx):
"""
pick_shape(ndarray approx) -> str

Fonction permettant de renvoyer la figure détectée selon le nombre de coins
acquis.

Parameters
----------
approx : ndarray
    Toutes les coordonnées des coins de la figure récupérées avec OpenCv.

Returns
-------
shape : str
    Figure détectée selon le nombre de coins dans la figure
"""
shape = ""
if len(approx) == 3:
    shape = "TRIANGLE"
elif len(approx) == 4:
    x1, y1, w, h = cv2.boundingRect(approx)
    aspect_ratio = float(w) / float(h)
    if aspect_ratio >= 0.85 and aspect_ratio <= 1.15:
        shape = "CARRE"
    else:
        shape = "RECTANGLE"

elif len(approx) == 5:
    shape = "PENTAGONE"

elif len(approx) == 6:
    shape = "HEXAGONE"

elif len(approx) == 12:
    shape = "ETOILE"

else:
    shape = "CERCLE"

print(f"forme : {shape}")

return shape
```

Finalement, en plaçant des formes, celles-ci sont détectées :

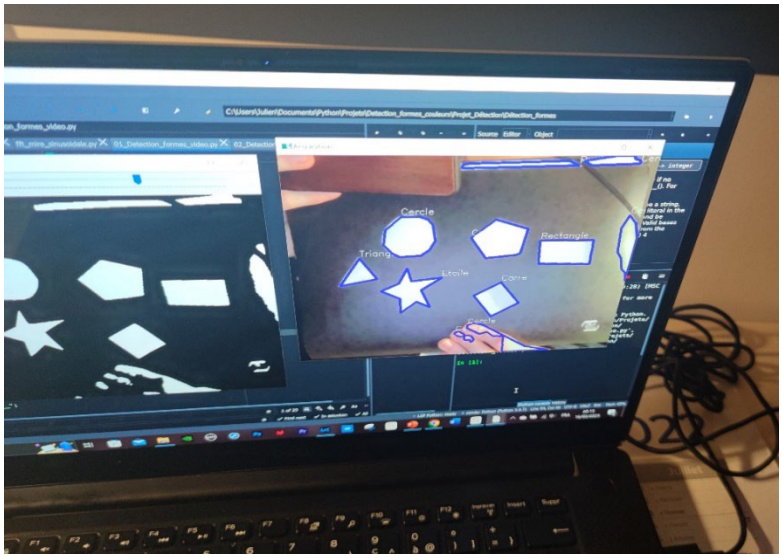
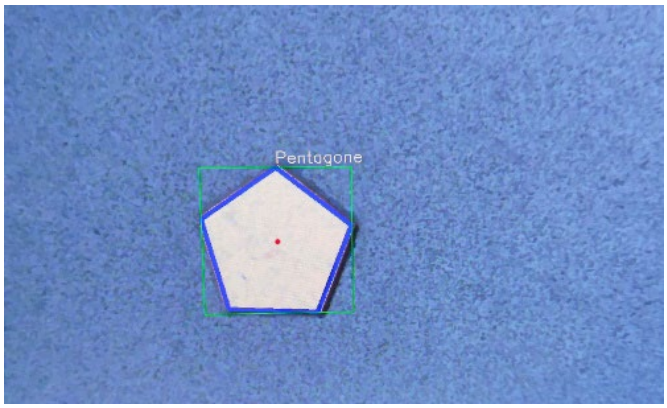


Figure 26 : Détection des formes

Appliqué au convoyeur par la suite, on peut reconnaître par exemple ce pentagone :



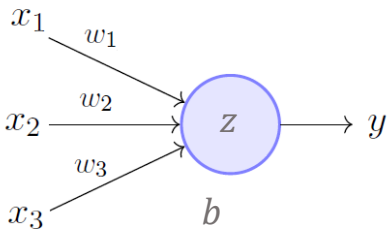
Le boost de l'intelligence artificielle

Nous avons choisi d'expérimenter l'intelligence artificielle (IA) sur ce projet de vision industrielle. Plus précisément le Machine learning dans la section Deep learning, donc un réseau de neurones. L'IA va nous permettre de détecter les formes sur le banc du convoyeur.

Principe général :

Un réseau de neurones est un ensemble d'entrées, qui subissent des opérations linéaires, par différents paramètres, pour arriver à des sorties.

L'exemple le plus simple de réseau de neurone est un réseau à un unique neurone, le **perceptron** :



Perceptron Model (Minsky-Papert in 1969)

(Source : towardsdatascience.com)

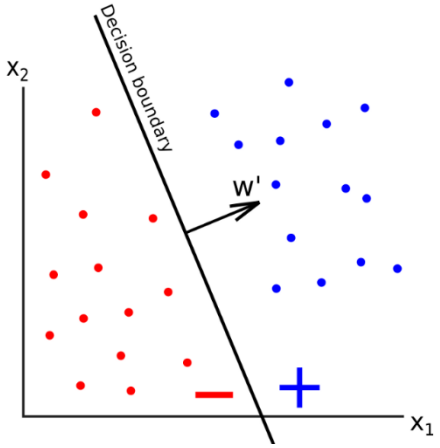
Les opérations mathématiques de bases nous donnent pour le perceptron la combinaison linéaire des entrées pondérées de coefficients :

$$z = w_1x_1 + w_2x_2 + w_3x_3 + b$$

Position du problème :

Prenons un exemple simple pour notre réseau d'un unique neurone.

Considérons 2 familles d'objets qui se distinguent par leur couleur : bleu et rouge. On cherche la meilleure délimitation possible (la **frontière de décision** ou **decision bundary**) pour séparer ces 2 familles (on prend le cas simple où aucun point n'est vraiment mélangé avec l'autre famille).



(Source : nablasquared.com)

Chaque point peut être repéré par ses « caractéristiques » ici ses coordonnées x_1 et x_2. On voit bien que les points rouges ont une valeur de x_1 plutôt petite alors que les points bleus ont une valeur de x_1 plutôt grande. C'est cette caractéristique en particulière qui va nous permettre de différencier les 2 familles.

Il faut bien comprendre que les réseaux de neurones que l'on essaye de vous expliquer n'ont qu'un unique objectif : **différencier.**

Revenons à notre perceptron : les paramètres que l'on va appeler « **poids** » w_1, w_2, w_3 et le « **biais** » b sont choisis aléatoirement au début. On peut donc pour chaque point calculer la valeur de z.

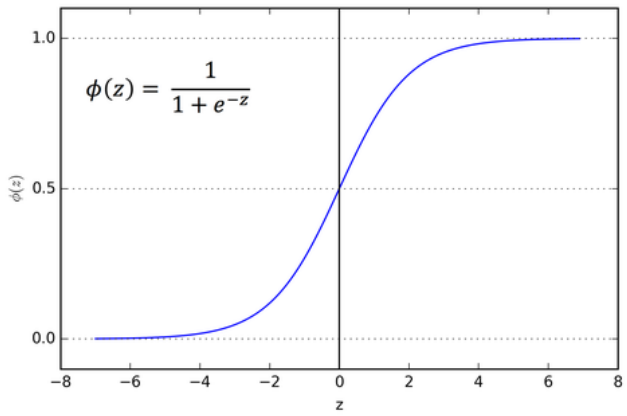
« Calcul » de l'appartenance aux familles :

On veut savoir avec le réseau de neurones, à quelle famille appartient chaque point pour pouvoir les séparer en toute connaissance de cause. C'est pour cela que le calcul ne se termine pas à la valeur de z. Il faut maintenant associer à chaque z une probabilité d'appartenance à la famille bleu ou à la famille rouge.

Plus un point est loin de la frontière de décision plus il est probable qu'il appartienne à la famille de ce côté-là de la frontière. (ex : si x_1 est très grand il est plus probable que le point soit bleu que rouge, sans qu'on en soit sûr).

On note **fonction d'activation**, la fonction qui va permettre de quantifier la probabilité d'appartenance à une famille, donc qui va donner une valeur allant de 0 à 1 pour chaque point (0 : appartient à la famille A et 1 : appartient à la famille B)

Il existe de nombreuses fonctions d'activation, dans cet exemple nous utiliserons la fonction sigmoïde :



Ainsi le résultat $p(z) = \frac{1}{1+e^{-z}}$ nous donne la probabilité d'appartenance aux familles pour chaque point.

On voit bien que si $z = 0$, donc si le point est pile sur la frontière de décision, on a $p(0) = 0.5$ donc on a 50% de chance que le point soit bleu ou rouge (en effet on est entre les 2 familles).

Pour conclure un neurone peut être caractérisé comme étant une simple fonction linéaire :

$$y = p(z) = p(w_1x_1 + w_2x_2 + w_3x_3 + b)$$

On note **modèle** l'ensemble des paramètres du réseau de neurones. Ainsi, chaque combinaison de paramètre est un modèle pour le réseau de neurones.

« Apprentissage » du réseau de neurones :

Or tout le problème des réseaux de neurones est que cette simple définition ne marche pas du tout. En effet, en prenant les poids et le biais du neurone au hasard lors du calcul, il y a très peu de chance de trouver la bonne droite séparatrice entre les 2 familles. Par exemple pour les 2 familles, d'équation :

$$w_1x_1 + w_2x_2 + b = 0 \ \ donne\ x_1 = \frac{-(w_2x_2 + b)}{w_1} \ \ (sur\ la\ frontière\ de\ décision\ z = 0)$$

Il faut ainsi trouver les bonnes valeurs des paramètres pour obtenir la meilleure séparation possible entre les 2 familles. Donc en quelque sorte minimiser « l'erreur » que l'on fait sur l'appartenance des familles.

On introduit une **fonction coût** qui va permettre tout d'abord de **quantifier l'erreur du modèle.**

On utilise dans notre exemple **la fonction log-loss** qui va calculer l'erreur, avec **N** étant le nombre de données à traiter (le nombre d'objets à différencier) :

$$-\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

Binary Cross-Entropy / Log Loss

Il faut savoir que le résultat est arbitraire et c'est la suite qui est intéressante.

L'étape cruciale d'un réseau de neurones va être la « correction » du modèle donc des paramètres, via la connaissance de l'erreur sur le résultat. Pour comprendre le fonctionnement d'un réseau de neurones, voici comment il « apprend » :

1- Le réseau prend un modèle aléatoire et réalise toutes les applications linéaires avec ce modèle (on parle de la 1ère itération du réseau ou 1ère génération).

2- La fonction de coût calcule l'erreur sur les résultats pour cette itération.

3- Le réseau utilise une méthode mathématique de rétroaction qui va modifier les paramètres donc modifier le modèle pour minimiser l'erreur à la prochaine itération.

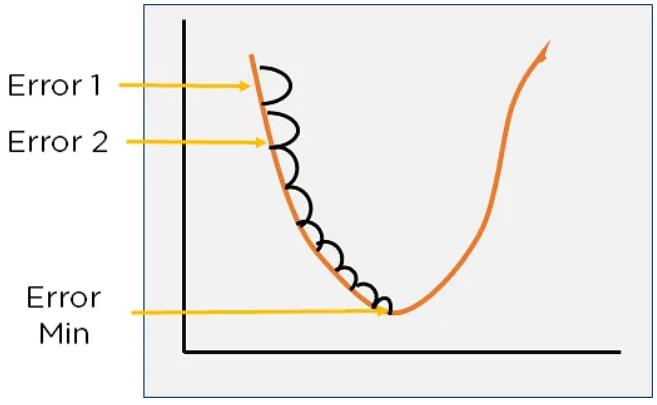
4- Le réseau recommence les calculs avec ce nouveau modèle (2ème itération) puis calcule l'erreur, et recommence ainsi de suite…

Cette méthode est réalisée à la suite jusqu'à ce que l'erreur soit satisfaisante pour dire que le modèle obtenu à la fin est bon.

<u>La méthode du gradient :</u>

C'est une des méthodes de rétroaction qui permet de modifier (donc « revenir en arrière ») le modèle pour la prochaine itération du réseau de neurones de façon à minimiser l'erreur donc minimiser la fonction coût.

C'est la partie la plus complexe à expliquer dans un réseau de neurones. Pour faire simple, on calcule le gradient de la fonction coût.



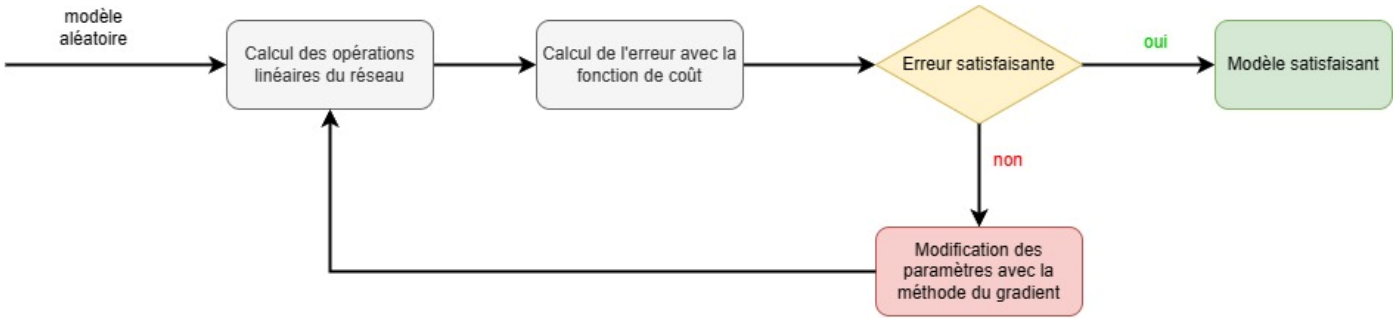
(Source : simplilearn.com)

C'est un peu comme une fonction de Newton, mais cette fois-ci avec n dimensions, n représentant le nombre de paramètres du modèle.

On peut voir une analogie à cette recherche de minimum d'erreur :

Imaginez une falaise avec une pierre en haut. La pierre va chercher à minimiser son énergie potentielle et va donc aller vers un creux en bas de la falaise qui va minimiser cette énergie. C'est exactement ce que fait la méthode de descente du gradient, en calculant pour chaque itération, le gradient des paramètres, elle permet d'aller vers le minimum d'erreur de la fonction coût (itération par itération).

Voici donc le principe global d'un réseau de neurones :

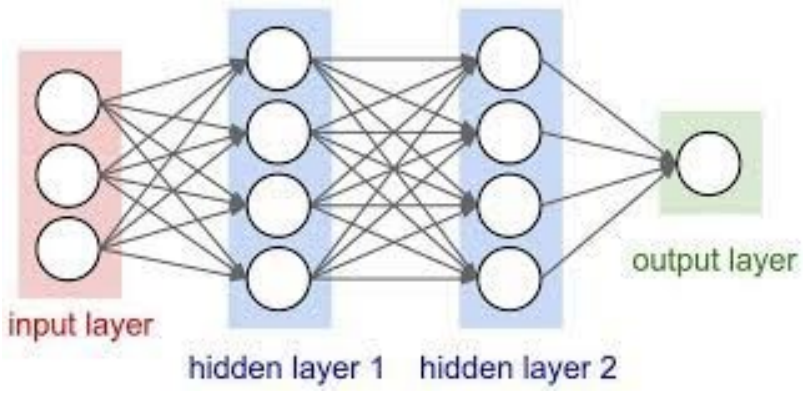


Remarque : ce genre de réseau de neurones peut différencier n'importe quoi. Mais vraiment ! Nous allons voir par la suite comment faire cela en programmant, mais cette méthode marche pour différencier n'importe quel objet quelle que soit sa nature.

Réseau de neurones à plusieurs couches :

Nous avons vu le cas d'un réseau de neurones à un unique neurone. Cependant cette représentation simplifiée ne marche que dans très peu de cas. En effet, avec un seul neurone les opérations que nous pouvons faire n'ont qu'un degré de linéarité, donc nous ne pouvons différencier les objets qu'avec une droite.

Pour remédier à ce problème on peut rajouter des neurones à notre réseau qui vont pouvoir rajouter des non-linéarités au problème et ainsi pouvoir résoudre (différentier) des problèmes plus compliqués.



(Source : bmc.com)

Pour comprendre le principe, chaque neurone de la couche **n** est relié à chaque neurone de la couche **n+1**. Ainsi chaque neurone de la couche des entrées est relié à chaque neurone de la 1ère couche.

Le principe mathématique est **le même** que pour le perceptron, cependant il y a une multitude de neurones réparties en différentes **couches.** Les opérations linéaires d'un neurone à l'autre sont les mêmes.

Chaque neurone de la couche **n** reçoit comme entrées, les résultats $p(z)$ de tous les neurones de la couche **n-1**. On appelle cette méthode la **forward propagation**.

Remarque : à partir de plusieurs couches, on travaille en représentation matricielle au niveau mathématiques pour simplifier les calculs.

Backward propagation :

La méthode de gradient est aussi appliquée pour un réseau de neurones à plusieurs couches. On appelle cette méthode plus généralement la **backward propagation**.

La backward propagation consiste à retracer comment la fonction coût évolue de la dernière couche jusqu'à la toute première couche. On calcule de la manière les gradients de chaque paramètre pour chaque couche pour revenir jusqu'au début.

Entrainement :

Plus le réseau a d'échantillons sur lequel il peut s'entrainer (calculer ses paramètres) mieux sera le modèle. En effet, une plus grande diversité d'objet du même type apportera plus de flexibilité au modèle lors de la phase de différenciation.

C'est pour cela que lors de l'entrainement d'un réseau de neurones, nous avons besoin d'une **base de données**, regroupant un grand (très grand) nombre d'échantillons servant à l'entrainement du réseau de neurones, donc au calcul de ses paramètres pour minimiser l'erreur.

Conclusion :

Normalement, plus il y a de couches plus le réseau va être performant. Cependant il prendra plus de temps à être « entrainé » en contrepartie (plus de calculs).

Plus un réseau de neurones est profond (beaucoup de couche), plus il a de chance de se perdre. On appelle ce phénomène le **vanishing gradient**.

(Voir des explications plus détaillées en annexe sur les réseaux de neurones)

Programmation – perceptron :

On va commencer par programmer un simple réseau de neurones de type perceptron. Pour cela on a besoin des bibliothèques numpy et mathplotlib.pyplot.

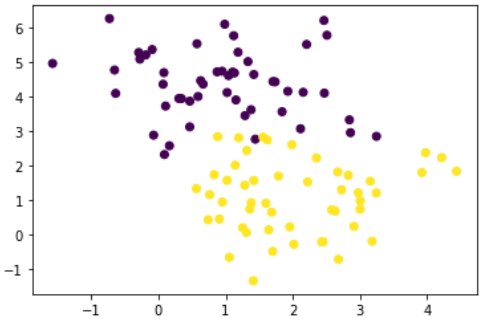
```
import numpy as np
import matplotlib.pyplot as plt
```

Cf les parties mathématiques en annexe, on peut créer les différentes fonctions de notre programme permettant les calculs. Notamment les calculs des gradients des différents paramètres (ici en représentation matricielle) :

```
#Fonction d'initialisation
def initialisation(X):
    W = np.random.randn(X.shape[1],1) #valeurs aléatoires aux composantes de W et à b
    b = np.random.randn(1)

    return (W, b)

#Fonction d'activation
def model(X, W, b):
    Z = X.dot(W) + b
    A = 1 / (1 + np.exp(-Z))
    return A

#Fonction coût
def log_loss(A, Y):
    return 1 / len(Y) * np.sum(-Y * np.log(A) - (1 - Y) * np.log(1 - A))


#Fonction gradients
def gradients(A, X, Y):
    dW = 1 / len(Y) * np.dot(X.T, A - Y)
    db = 1 / len(Y) * np.sum(A - Y)
    return (dW, db)

#Fonction de descente de gradient
def desc_gradient(dW, db, W, b, alpha):
    W = W - alpha * dW
    b = b - alpha * db
    return (W, b)
```

Pour notre exemple, on choisit d'importer des points, séparer en 2 familles qu'on va chercher à séparer :

```
X, Y = make_blobs(n_samples=100, n_features=2, centers=2,random_state=0)
Y = Y.reshape((Y.shape[0],1))
```



Avec X, la matrice contenant les positions des points, et Y la matrice contenant l'appartenance des points aux familles.

Par la suite on peut créer une fonction **neurone_artificiel()** prenant en entrée la matrice X, Y et le nombre d'itérations qu'on veut donner à notre modèle avant qu'il ne s'arrête. On construit donc le modèle dans cette fonction :

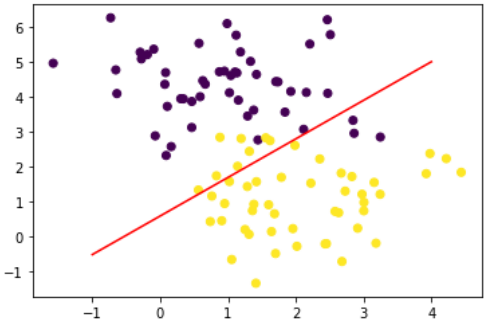
```
def neurone_artificiel(X, Y, alpha = 0.1, nb_iteration = 100):
    #initialisation W, b
    W, b = initialisation(X)

    Loss = [] #on regarde pour chaque itération, l'erreur pour suivre l'évolution

    #apprentissage
    for i in range(nb_iteration):
        A = model(X, W, b)
        Loss.append(log_loss(A, Y))
        dW, db = gradients(A, X, Y)
        W, b = desc_gradient(dW, db, W, b ,alpha)

    plt.plot(Loss)
    plt.show()

    Y_prediction = prediction(X, W, b)
    print(accuracy_score(Y,Y_prediction)) # on compare les données de références Y

    return (W, b)
```

On retrouve bien cette rétroaction lors du calcul du modèle avec la boucle for de la fonction. La fonction renvoie les paramètres finaux calculés après les n itérations données en entrées.

On peut par ailleurs utiliser une fonction **prediction()**, permettant de comparer les valeurs calculées avec le modèle à la fin avec les valeurs de références Y pour voir si le modèle est performant ou non.

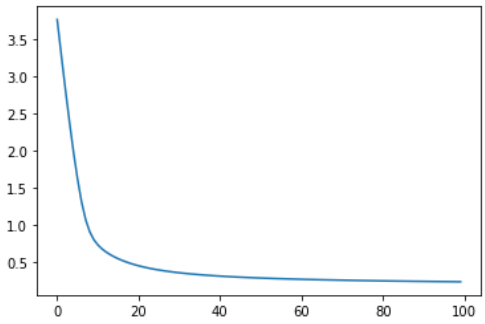
```
def prediction(X, W, b):
    A = model(X, W, b)
    print(A)   #on donne la probabilité des données
    return A >= 0.5  #on classe les données dans les 2 familles
```

On peut avec cet exemple obtenir par exemple la droite suivante :



On voit bien qu'il est impossible avec une seule droite de séparer entièrement les 2 familles. Ainsi, il y a bien une réelle limite au réseau d'un neurone unique. Pour résoudre ce problème il faudrait rajouter des neurones pour apporter des non-linéarités (« courber » la droite).

Remarque : on peut tracer l'erreur du modèle en fonction du nombre d'itérations pour voir la progression du modèle au fil des itérations :



On voit bien ici que le modèle réduit l'erreur puis commence à stagner car le modèle ne peut plus progresser.

Programmation – réseau à n neurones :

On se rend rapidement compte du côté mathématique, qu'il va être assez compliqué de programmer entièrement un réseau de neurones multicouches à la main.

Ainsi, on utilise plus généralement pour les réseaux de neurones, des bibliothèques qui intègrent entièrement les fonctions réalisées auparavant.

Nous avons utilisé pour ce projet l'une d'entre elle : **sklearn**,

Plus précisément la classe **MLPClassifier.**

```
from sklearn import datasets
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
```

Création de la base de données :

Cette IA a pour but de différencier les formes des objets qui arrivent sur le convoyeur (rond, triangle, carré, pentagone…). Il faut donc lors de son entrainement lui fournir des images similaires pour que le modèle puisse être performant pour ces objets.

On choisit donc de créer une base de données en partant de quelques images de références puis en créant une multitude d'images (par translation, rotation) pour diversifier les objets lors de l'entrainement.

```python
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np
from scipy import ndimage
import csv
import pandas as pd
import random as random
from tqdm import tqdm

# Ouverture de l'image
imageLue = Image.open("C:/Users/Travail/Desktop/1A IOGS/Projet IETI/programme/IA/base_donnees/formes/resize_size/h_resize.png")
# Convertion de l'image en nuance de gris
imageGris = imageLue.convert("L")
# Convertion de l'image en array numpy
image_array = np.array(imageGris)

# Seuillage de l'image pour n'avoir que la forme sur un fond noir
for i in range(len(image_array)):
    for j in range(len(image_array)):
        if image_array[i,j] < 170:
            image_array[i,j] = 0

taille_image_compress = 64
# Compression de l'image
imageConvertie = Image.fromarray(image_array)
imageCompress = imageConvertie.resize((taille_image_compress,taille_image_compress))
image_array_2 = np.array(imageCompress)

# Création des images par rotation et translation
nb_images = 10000
liste = np.zeros((nb_images,taille_image_compress**2))
for i in tqdm(range(nb_images)):

    rotation = random.randint(0,359)
    translation_x = random.randint(-20,20)
    translation_y = random.randint(-20,20)

    new_image_1 = ndimage.rotate(image_array_2, rotation, reshape=False, order=3, mode='constant')
    new_image_2 = ndimage.shift(image_array_2, (translation_x, translation_y), order=3,mode='constant')

    l = []
    for n in new_image_2:
            for item in n:
                l.append(item)
    liste[i,:] = l

# Écriture du fichier .csv
df = pd.DataFrame(liste)
df.to_csv("C:/Users/Travail/Desktop/1A IOGS/Projet IETI/programme/IA/base_donnees/formes/hexagone.csv")
```

Ici on a choisi de créer une base de données de **10000 images** par forme, que l'on regroupe dans un fichier csv (on y stocke les valeurs des pixels des images).

Comme dit précédemment, plus on a de neurones plus les calculs sont long. C'est pourquoi les entrées de notre réseau de neurones (qui correspondent aux valeurs des pixels de l'images) doivent être minimisées.

On prend pour ce réseau de neurone, **4096 entrées** soit des images faisant **64*64 pixels**.

Calcul du modèle :

La base de données étant créée, on peut maintenant calculer notre modèle. Pour cela on importe tout d'abord la base de données dans le code puis on la sépare en 2 catégories :

- **Les données d'entrainement**

- **Les données de test**

Les données d'entrainement vont servir à entrainer le modèle comme leur nom l'indique. Alors que les données de test vont permettre de tester le modèle après calcul pour voir s'il est performant. Il faut absolument que les données de test soient différentes des données d'entrainement pour ne pas créer de biais lors de la différenciation.

```
pourcentage_dataset = 70
nb_pct = int(pourcentage_dataset*10000/100)

# Importation des base de données
data_rond = pd.read_csv('C:/Users/Travail/Desktop/1A IOGS/Projet IETI/programme/IA/base_donnees/formes/rond.csv').iloc[:nb_pct,1:].values
print("rond")
data_triangle = pd.read_csv('C:/Users/Travail/Desktop/1A IOGS/Projet IETI/programme/IA/base_donnees/formes/triangle.csv').iloc[:nb_pct,1:].values
print("triangle")
data_carre = pd.read_csv('C:/Users/Travail/Desktop/1A IOGS/Projet IETI/programme/IA/base_donnees/formes/carre.csv').iloc[:nb_pct,1:].values
print("carre")
data_pentagone = pd.read_csv('C:/Users/Travail/Desktop/1A IOGS/Projet IETI/programme/IA/base_donnees/formes/pentagone.csv').iloc[:nb_pct,1:].values
print("pentagone")
data_hexagone = pd.read_csv('C:/Users/Travail/Desktop/1A IOGS/Projet IETI/programme/IA/base_donnees/formes/hexagone.csv').iloc[:nb_pct,1:].values
print("hexagone")
data_etoile = pd.read_csv('C:/Users/Travail/Desktop/1A IOGS/Projet IETI/programme/IA/base_donnees/formes/etoile.csv').iloc[:nb_pct,1:].values


liste = [data_rond, data_triangle, data_carre, data_pentagone, data_hexagone, data_etoile]

# Récupération des données ( x--> données du modèle ; y--> données de vérification)
x = np.zeros((6*nb_pct,4096))
y = np.zeros(6*nb_pct)

for i in tqdm(range(6)):
    y[nb_pct*i:nb_pct*(i+1)] = i
    x[nb_pct*i:nb_pct*(i+1),:] = liste[i]

# Séparation des données d'entrainement et des données de test
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.5, shuffle=True)
```

Une fois les données séparées on peut commencer le calcul du modèle. Tout d'abord il faut l'initialiser avec ce que l'on appelle les hyperparamètres.

Ce sont tous les paramètres globaux qui sont à l'origine du modèle (que nous ne détaillerons pas ici) :

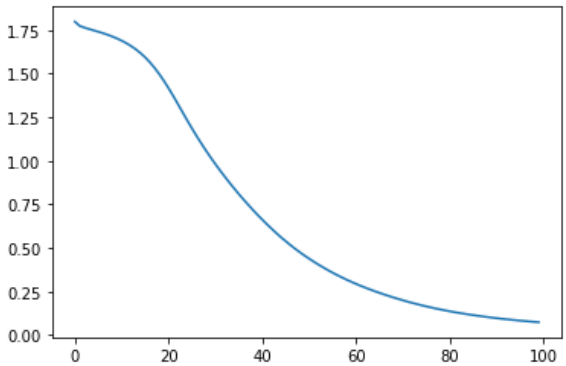
```
# Initialisation du modèle
print("Lancement du modèle")
mlp = MLPClassifier(hidden_layer_sizes=(256,128),
                    activation='logistic',
                    solver='sgd',
                    learning_rate='constant',
                    learning_rate_init=1e-3,
                    max_iter = 100,
                    random_state=0, tol=5e-5,
                    verbose=True,
                    momentum=0.9)
"""
hidden_layer_sizes : Nombre de couche et nombre de neurone par couche (10,10) --> 2 couches cachées de 10 neurone chacune
activation : Fonction d'activation pour connaitre la probabilité d'appartenance de la donnée à un groupe --> 'logistic' = sigmoïd
solver : Methode de determination de correction de l'erreur --> 'sgd' = descente du gradient
learning_rate : technique d'apprentissage
learning_rate_init : Le taux d'apprentissage initial utilisé. Contrôle la taille du pas dans la mise à jour des poids (paramètres)
max_iter : Nombre d'itération maximum avant l'arret de l'entrainement
random_state : Initialise aléatoirement la valeur de tous les paramètres du réseau de neurone (la seed)
tol : Tolérance de l'optimisation de l'erreur. Différence minimum de l'erreur entre 10 itérations avant que l'entrainement s'arrête.
verbose : Print l'avancement de l'entrainement du modèle
momentum : Coefficient d'amortissement de la fonction de descente de gradient
"""
```

Puis une fois initialisé, on peut entrainer le modèle avec la méthode **.fit()** :

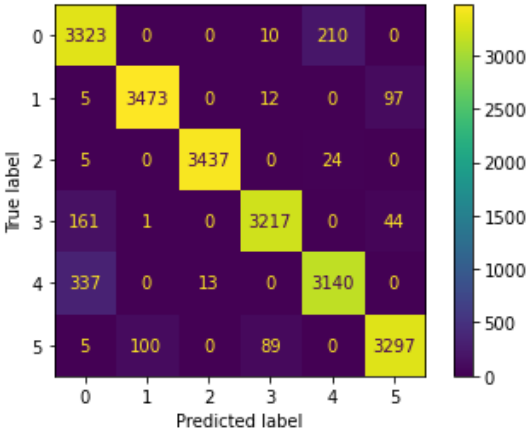
```
# Entrainement du modèle
print("Lancement de l'entrainement")
mlp.fit(x_train, y_train)

# Prédiction de la référence des données test
print("Lancement des prédictions")
predi=mlp.predict(x_test)

# Précision obtenue avec les données test en les comparant aux vraies classes
print(accuracy_score(y_test, predi))

print("Training set score: %f" % mlp.score(x_train, y_train))
print("Test set score: %f" % mlp.score(x_test, y_test))
```

Enfin, après calcul on réalise les prédictions sur les données de test puis on affiche la précision que l'on a sur ces données pour voir si le modèle est performant.

Après de nombreux essais sur les bons hyperparamètres à trouver. Nous obtenons un modèle qui permet de différencier les formes avec une précision de 95%. Voici la courbe de l'erreur en fonction des itérations :



On peut par ailleurs voir où le modèle à fait des erreurs avec cette représentation :



Chaque chiffre correspond à une forme (0 : rond, 1 : triangle, 2 : carré, 3 : pentagone, 4 : hexagone, 5 : étoile)

Predicted label = ce que le modèle prédit / **True label** = ce que l'objet est vraiment.

On voit bien que dans la plupart des cas le modèle trouve le bon résultat. Or par exemple, pour des objets comme des ronds, l'IA se trompe en pensant qu'il s'agit d'hexagone et inversement. Ce qui est plutôt logique dû à la forte similitude des formes et au peu de pixels que contiennent les images.

Implémentation de l'IA :

Une fois le modèle calculé, on peut récupérer les paramètres du modèle avec la méthode coefs_ et intercepts_ puis les sauvegarder dans des tableaux numpy :

```
parametres_poids = mlp.coefs_
parametres_bias = mlp.intercepts_

#Création d'un fichier .npy (fichier enregistrant un tableau numpy directement) pour stocker les paramètres
np.save("parametres_p.npy",parametres_poids,allow_pickle = True)
np.save("parametres_b.npy",parametres_bias,allow_pickle = True)
```

Le code qui est présent dans notre projet se décompose en 2 parties :

- L'initialisation du modèle
- La différenciation des images

Tout d'abord la partie initialisation consiste à juste récupérer les paramètres que l'on avait enregistrés, initialiser le modèle et lui attribuer ces paramètres :

```
# Importation des paramètres
A = np.load("parametres_poids.npy", allow_pickle = True)
A = [A[0],A[1],A[2]]

B = np.load("parametres_bias.npy", allow_pickle = True)
B = [B[0],B[1],B[2]]


# Création du modèle
mlp = MLPClassifier(hidden_layer_sizes=(256,128),
                    activation='logistic',
                    solver='sgd',
                    learning_rate='constant',
                    learning_rate_init=1e-4,
                    max_iter = 1, random_state=0,
                    tol=5e-5,
                    verbose=True,
                    momentum=0.9)

# Initialisation dess paramètres dans le modèle (obligatoire)
x_init = np.zeros((6,4096))
y_init = [1,2,3,4,5,6]

mlp.fit(x_init, y_init)

# On attribue les paramètres au modèle
mlp.intercepts_ = B
mlp.coefs_ = A
```

Puis, dans la boucle **while** où l'on récupère les images, on a ce code :

```
image = image[:,:,0:3]

imageConvertie_1 = Image.fromarray(image)

imageGris = imageConvertie_1.convert("L")
# Convertion de l'image en array numpy
image_array = np.array(imageGris)

# Seuillage de l'image pour n'avoir que la forme sur un fond noir
for i in range(512):
    for j in range(640):
        if image_array[i,j] < 120:
            image_array[i,j] = 0

# Compression de l'image
imageConvertie_2 = Image.fromarray(image_array)
imageCompress = imageConvertie_2.resize((64,64))
image_array_2 = np.array(imageCompress)

predi=mlp.predict(image_array_2)
```

Il permet de récupérer l'image dans le bon format, la convertir en teinte de gris, la compresser en 64*64 pixels, puis faire la prédiction sur la forme qu'il y a dans l'image avec la méthode **predict().**

La variable **predi** est un chiffre entre 0 et 5, renseignant donc sur la nature de la forme.

Le programme de la caméra

Le programme du fonctionnement de la caméra est déjà fourni. La caméra est une caméra RGB uEye de résolution 1280x1024. La caméra peut être pilotée depuis le logiciel uEye Cockpit mais on souhaite la contrôler sous Python.

Figure 27 : Caméra Ueye 1280x1024

En particulier le programme de la caméra fourni, contient une boucle **while** (tant que la caméra est ouverte). On récupère la donnée sous la forme d'une array propre à la bibliothèque ueye puis on la reformate en array numpy pour la donner à OpenCv. On récupère alors la frame recadrée à 50% que l'on peut traiter. A la fréquence de la caméra, on récupère en continu la frame.

```
# Continuous image display
while(nRet == ueye.IS_SUCCESS):

    # In order to display the image in an OpenCV window we need to...
    # ...extract the data of our image memory
    array = ueye.get_data(pcImageMemory, width, height, nBitsPerPixel, pitch, copy=False)

    # bytes_per_pixel = int(nBitsPerPixel / 8)

    # ...reshape it in an numpy array...
    frame = np.reshape(array,(height.value, width.value, bytes_per_pixel))

    # ...resize t                      he image by a half
    frame = cv2.resize(frame,(0,0),fx=0.5, fy=0.5)
```

Figure 28 : Programme de la caméra

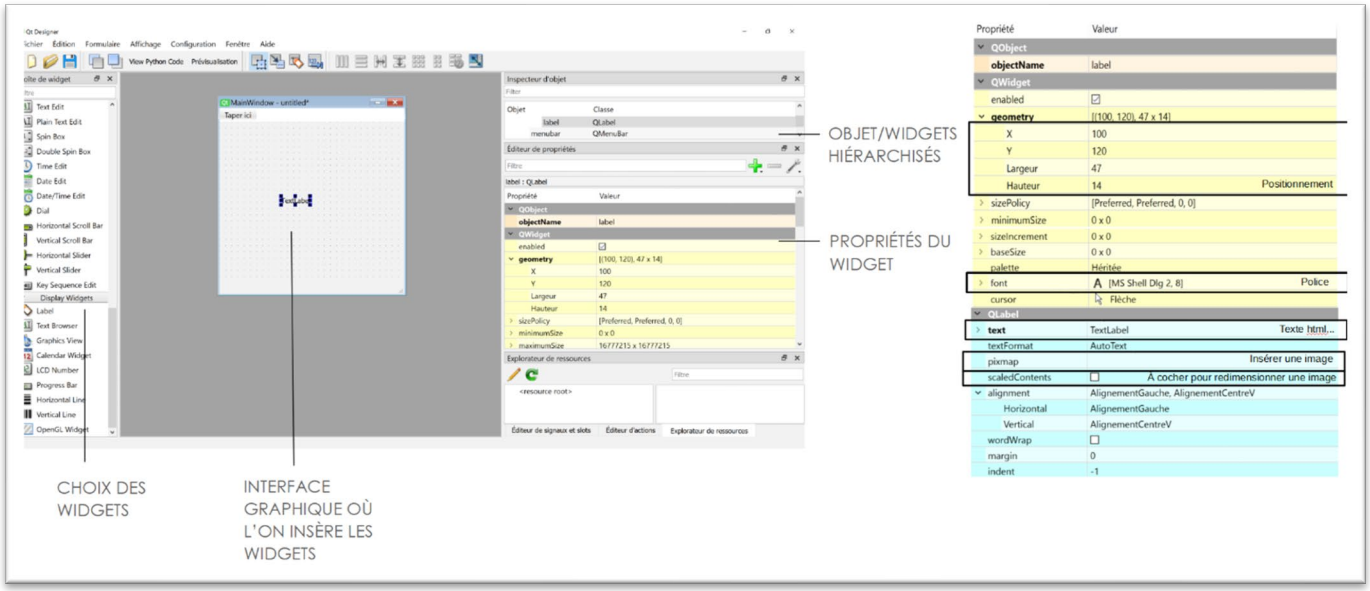
L'interface graphique :

Pour réaliser l'interface graphique, on a utilisé la bibliothèque **PyQt5** associée à l'application **QtDesigner**. QtDesigner permet de créer graphiquement l'IHM en statique, pour pouvoir ensuite l'importer dans python, et associer aux widgets (boutons, combobox, …) des actions.

Le design avec QtDesigner

Qt Designer est un outil qui permet de designer et construire une interface graphique (GUI pour Graphical User Interface).

On ouvre QtDesigner avec le logo :

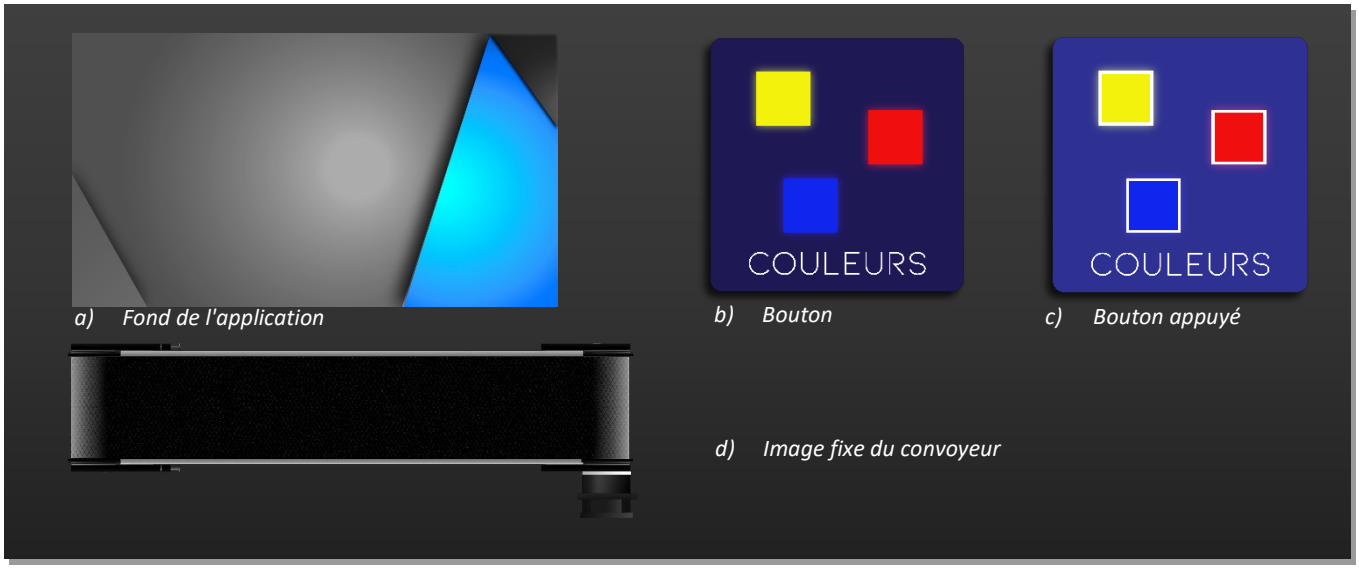


On clique sur créer, et l'interface de QtDesigner se présente comme ci-contre :

WIDGET : élément graphique implémenté dans une interface graphique qui permet d'interagir avec l'utilisateur (boutons…)

On a implémenté en particulier un « StackedWidget » qui permet d'insérer différentes pages (indicées dans le programme). On a également utilisé de nombreux « PushButton » qui ont un état appuyé ou non, auquel on associera une méthode. Dans la hiérarchisation, des widgets, il faut faire attention de leur donner un bon nom, puisqu'ils constitueront le nom des variables des objets dans la programmation en python.

Avec Illustrator, on crée un fond vectorialisé, et avec Photoshop, on crée des boutons que l'on va animer par la suite :



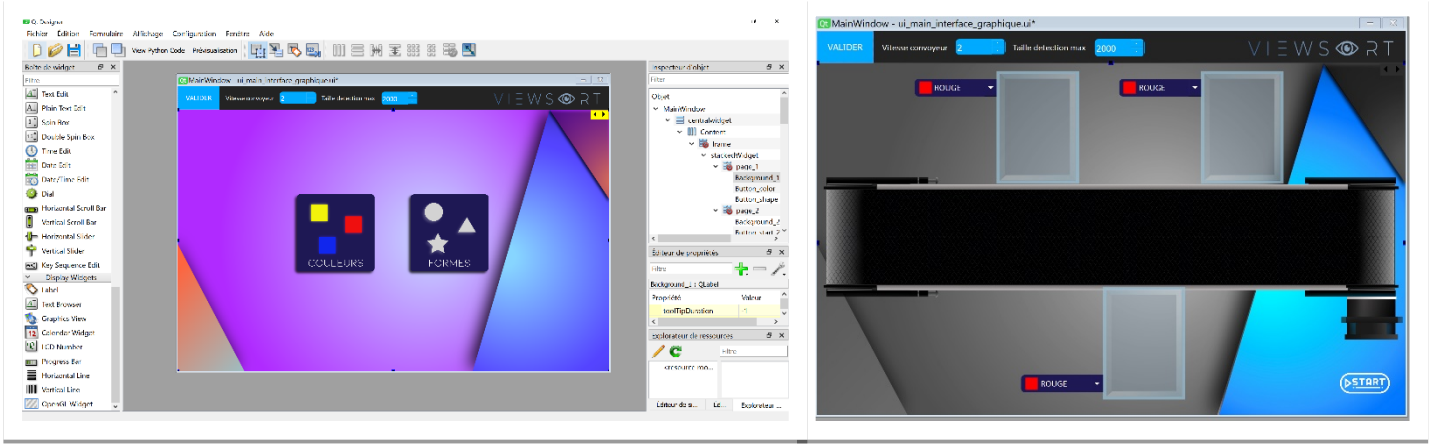
a) *Fond de l'application*

b) *Bouton*

c) *Bouton appuyé*

d) *Image fixe du convoyeur*

On crée pour différentes pages du stackedWidget, le design de l'interface graphique, en insérant les éléments créés.



On insère dans un même et unique dossier tous les éléments graphiques au format .png pour avoir un fond transparent, avec pour chaque bouton, l'image appuyée et non appuyée du bouton graphique.

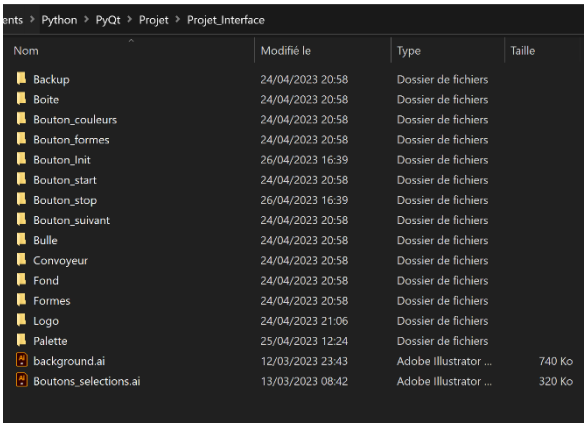


Figure 29 : Tous les éléments graphiques de l'interface

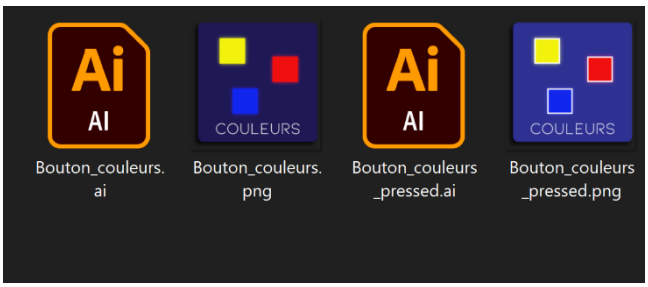
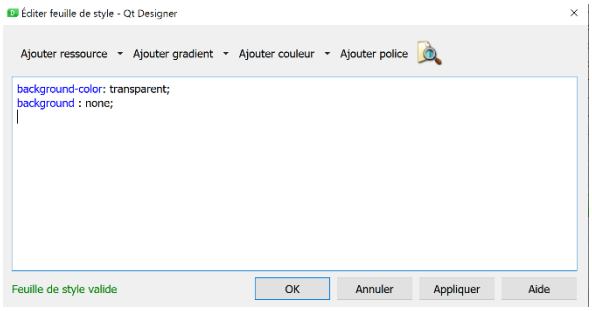


Figure 30 : Bouton avec les images des 2 états

Pour améliorer, l'apparence d'un widget, on créer une feuille de style pour chaque élément graphique (clic droit sur l'objet, modifier la feuille de style) :

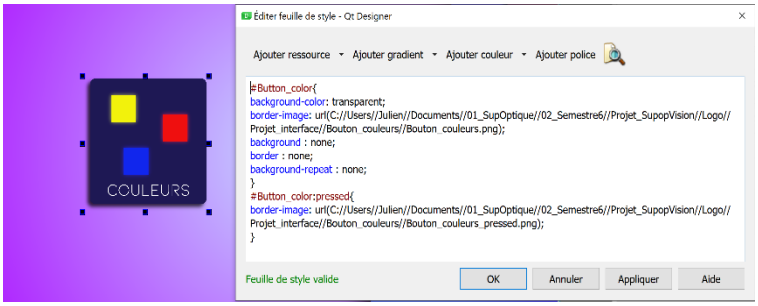
Par exemple pour un fond transparent :

Remarque : une image fixe, est placée avec le widget « label », qui sert aux images et au texte. On ajoute l'image dans la section bleue avec « PixelMap »



Pour un bouton :

On récupère le chemin de l'image et on précise l'état « pressed », pour préciser l'image utilisée quand le bouton est appuyé. On rajoutera également l'état « hover », lorsque la souris survole l'image du bouton, son image changera et sera éclaircie.



Par exemple pour un PushButton :

Pour spécifier dans QtDesigner, l'image dans son état haut ou bas

Finalement, le fichier enregistré a pour extension « .ui » et peut être importé dans Python par 2 méthodes :

```
#Méthode 1
#Dans une invite de commande dans le dossier
#où le .ui avec QtDesigner est enregistré, on tape :
pyuic5 -x mon_app.ui -o interface.py

#Méthode 2
#Charger le .ui créér avec QtDesigner directement avec Python
class Ui(QtWidgets.QMainWindow):
    def __init__(self):
        super(Ui, self).__init__()
        uic.loadUi('mon_app.ui', self) # Charger le fichier .ui
        self.show() # Afficher à l'écran l'interface graphique
```

On utilisera la méthode 1 par la suite qui compile un programme python en présentant 1 classe Ui_MainWindow correspondant à l'application graphique, instanciée dans le « main » à la fin du programme et constituée de 2 méthodes :

- setupUi : toute l'interface graphique est formée des noms associés aux widgets particuliers (PushButon, combobox, labels, frame…)

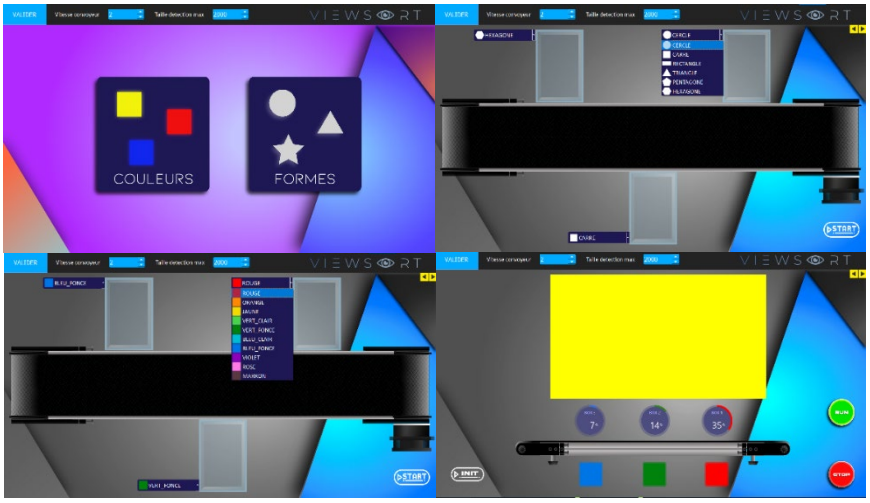
- retranslateUi (qui associe aux items les noms à afficher sur l'écran en particulier pour les combobox)

```
# Form implementation generated from reading ui file 'ui_main7.ui'
#
# Created by: PyQt5 UI code generator 5.15.4
#
# WARNING: Any manual changes made to this file will be lost when pyuic5 is
# run again.  Do not edit this file unless you know what you are doing.


from PyQt5 import QtCore, QtGui, QtWidgets


class Ui_MainWindow(object):
    def setupUi(self, MainWindow):
        MainWindow.setObjectName("MainWindow")
        MainWindow.resize(720, 480)
        MainWindow.setStyleSheet("background-color: rgb(45, 45, 45);")
        self.centralwidget = QtWidgets.QWidget(MainWindow)
        self.centralwidget.setObjectName("centralwidget")
        self.verticalLayout = QtWidgets.QVBoxLayout(self.centralwidget)
        self.verticalLayout.setContentsMargins(0, 0, 0, 0)
        self.verticalLayout.setSpacing(0)
        self.verticalLayout.setObjectName("verticalLayout")
        self.Top_Bar = QtWidgets.QFrame(self.centralwidget)
        self.Top_Bar.setMaximumSize(QtCore.QSize(16777215, 40))
        self.Top_Bar.setStyleSheet("background-color: rgb(35, 35, 35);")
        self.Top_Bar.setFrameShape(QtWidgets.QFrame.NoFrame)
        self.Top_Bar.setFrameShadow(QtWidgets.QFrame.Raised)
        self.Top_Bar.setObjectName("Top_Bar")
        self.horizontalLayout = QtWidgets.QHBoxLayout(self.Top_Bar)
        self.horizontalLayout.setContentsMargins(0, 0, 0, 0)
        self.horizontalLayout.setSpacing(0)
        self.horizontalLayout.setObjectName("horizontalLayout")
        self.frame_toggle = QtWidgets.QFrame(self.Top_Bar)
        self.frame_toggle.setMaximumSize(QtCore.QSize(70, 40))
        self.frame_toggle.setStyleSheet("background-color: rgb(85, 170, 255);")
        self.frame_toggle.setFrameShape(QtWidgets.QFrame.StyledPanel)
        self.frame_toggle.setFrameShadow(QtWidgets.QFrame.Raised)
        self.frame_toggle.setObjectName("frame_toggle")
        self.verticalLayout_2 = QtWidgets.QVBoxLayout(self.frame_toggle)
        self.verticalLayout_2.setContentsMargins(0, 0, 0, 0)
        self.verticalLayout_2.setSpacing(0)
        self.verticalLayout_2.setObjectName("verticalLayout_2")
        self.Button_menu = QtWidgets.QPushButton(self.frame_toggle)
        sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Expanding, QtWidgets.QSizePolicy.Expanding)
        sizePolicy.setHorizontalStretch(0)
        sizePolicy.setVerticalStretch(0)
        sizePolicy.setHeightForWidth(self.Button_menu.sizePolicy().hasHeightForWidth())
        self.Button_menu.setSizePolicy(sizePolicy)
        self.Button_menu.setStyleSheet("#Button_menu{\n"
"background-color: rgb(0, 170, 255);\n"
```

Figure 31 : Code compilé en Python à partir de QtDesigner

La programmation Python de l'init de l'interface

On peut améliorer et coder de nouveaux éléments comme ceux précédemment créer dans la méthode d'initialisation de l'interface graphique setupUI. On redéfinira également en fonction de la taille de l'écran, les coordonnées des Widgets. Cette première partie du code constitue l'initialisation graphique :



L'association de méthodes aux événements

Le cas des PushButton :

Il faut faire fonctionner l'interface graphique, pour chaque boutton, en particulier, il faut assigner une action. On créer une méthode dans la classe de l'interface graphique qui sera réalisée, à chaque fois que le pushButton sera actionnée :

1) On crée la variable du bouton nommée ici « Button_color », qui sera affichée page_1 :

2) On dispose les coordonnées du bouton puis son style avec la méthode setStyleSheet comme vue précédemment dans QtDesigner

```
#Bouton de couleurs
self.Button_color = QtWidgets.QPushButton(self.page_1)
self.Button_color.setGeometry(QtCore.QRect(L_WINDOW//4-L_WINDOW//20, H_WINDOW//5, H_WINDOW//2, H_WINDOW//2))
self.Button_color.setStyleSheet("#Button_color{\n"
                                 "background-color: transparent;\n"
                                 "border-image: url(.//Projet_interface//Bouton_couleurs//Bouton_couleurs.png);\n"
                                 "background : none;\n"
                                 "border : none;\n"
                                 "background-repeat : none;\n"
                                 "}\n"
                                 "#Button_color:pressed{\n"
                                 "border-image: url(.//Projet_interface//Bouton_couleurs//Bouton_couleurs_pressed.png);\n"
                                 "}"
                                 "#Button_color:hover{\n"
                                 "border-image: url(.//Projet_interface//Bouton_couleurs//Bouton_couleurs_pressed.png);\n"
                                 "}")
self.Button_color.setText("")
self.Button_color.setObjectName("Button_color")
```

3) Le bouton est associé à une méthode lorsqu'il est pressé, ici **choice_color** :

```
self.Button_color.clicked.connect(self.choice_color)
```

On retiendra pour avoir une action avec un PushButton :

```
self.Bouton.clicked.connect(self.action_bouton)
```

Puis après la méthode setupUi, on place les méthodes associées aux actions des widgets. Par exemple, en cliquant sur le bouton pour le tri des couleurs, le choix de la couleur se met à « True » et on passe à la page suivante :

```
def choice_color(self):
    """Le booléen global choice est à 1, si on fait le choix de détection de couleur

    """
    global choice
    choice = True
    self.go_to_second()
```

Les autres PushButton :

Les boutons vont servir à changer la plupart du temps l'état de variables globales qui seront communiquées à la Nucléo pour faire arrêter/fonctionner les moteurs, réinitialiser la position des servomoteurs, ou bien donner la liste des couleurs/formes choisies par l'utilisateur :

```
# ================================================================================
# VARIABLES GLOBALES ENTRE LES PROGRAMMES
# ================================================================================

#Globales booléens
global choice # True couleur, false forme
choice = True
global marche # Valeurs on ou off des moteurs, de l'application
marche = 0
global servo_retour # Valeurs retour des servomoteurs (1 = retour en position initiale)
retour = 0

#Globales pour OpenCv et la détection
global taille_max # Taille max de détection pour OpenCv (Surface px²)
global vitesse # Vitesse du moteur A CHANGER


#Globales pour le tri dans les 3 boîtes
global color # Liste des 3 couleurs à récuperer pour les 3 boîtes à trier
color = ['ROUGE', 'VERT_FONCE', 'BLEU_FONCE']
global shape # Liste des 3 formes à récuperer pour les 3 boîtes à trier
shape = ['CERCLE', 'CARRE', 'HEXAGONE']
global pourcentage_box # Pourcentage de remplissage des boîtes (liste) pour les 3 boîtes
```

Les boutons pour tourner les pages : permettent de changer l'indice du stackedWidget :

```
# ================================================================================
# Changer de page
# ================================================================================

    def go_to_first(self):
        """Se déplacer à la page 1 du stackedWidget

        """
        self.stackedWidget.setCurrentIndex(0)

    def go_to_second(self):
        """Se déplacer à la page 2 du stackedWidget

        """
        self.stackedWidget.setCurrentIndex(1)

    def go_to_third(self):
        """Se déplacer à la page 3 du stackedWidget

        """
        self.stackedWidget.setCurrentIndex(2)

    def go_to_fourth(self):
        """Se déplacer à la page 4 du stackedWidget

        """
        self.stackedWidget.setCurrentIndex(3)
```

Les boutons du choix de la méthode de tri couleur/forme : permettent de définir la variable globale booléenne choice (True : choix couleur / False : choix forme).

```
# ========================================================================
# Choix couleur-forme
# ========================================================================

    def choice_color(self):
        """Le booléen global choice est à 1, si on fait le choix de détection de couleur

        """
        global choice
        choice = True
        self.go_to_second()

    def choice_shape(self):
        """Le booléen global choice est à 0, si on fait le choix de détection de formes

        """
        global choice
        choice = False
        self.go_to_fourth()
```

Les boutons de contrôle des moteurs de la nucléo pour le démarrage et l'arrêt : permettent de définir la variable globale booléenne marche (True : moteur à CC on / False : moteur à CC off).

```
def stop(self):
    """
    Arrête le process : pourcentage stop, moteurs off grâce à marche = 0

    """
    global marche
    print('stop')
    marche = 0
    ser.write(bytes('s', 'utf-8'))
    print("Moteurs à l'arrêt")

def start(self):
    """
    Démarre le process : moteurs on grâce à marche = 1

    """
    print('run')
    global marche, servo_retour
    marche = 1
    ser.write(bytes('r', 'utf-8'))
    servo_retour = 0 #Les servo ne sont plus forcés en position minimale
    print("Moteurs en route")
```

Le bouton de réinitialisation : permet de remettre à zéro l'état de tri, et les compteurs, et de faire revenir les servomoteurs en position initiale.

```
def initServoPressed(self):
    """
    Retour des servomoteurs en position initiale, et pourcentages à 0%
    """
    global servo_retour
    servo_retour = 1
    global pourcentage_box


    config.remplissage_color = {'ROUGE': 0, 'ORANGE': 0, 'JAUNE': 0, 'VERT_CLAIR': 0,
                'VERT_FONCE' : 0, 'BLEU_CLAIR': 0, 'BLEU_FONCE': 0,
                 'VIOLET': 0, "ROSE" : 0, "MARRON" : 0, "INDEFINI" : 0}
    config.remplissage_shape = {'TRIANGLE': 0, 'CARRE': 0, 'RECTANGLE': 0,
               'PENTAGONE': 0, 'HEXAGONE' : 0, 'ETOILE': 0, 'CERCLE': 0}

    #Remplit le pourcentage initialiser à 0 sur les roues chromatiques de %
    pourcentage_box = [0, 0, 0]
    self.set_pourcentage(pourcentage_box)
```

Application continue avec émission de signaux réguliers pour rafraîchir l'application

L'application une fois initialisée et liée aux méthodes des boutons est dans un état « statique ». Il a fallu, avoir une application temps-réel capable de se réaffranchir régulièrement, pour mettre à jour graphiquement l'état du tri en précisant les pourcentages de remplissage des 3 boîtes de tri, et d'envoyer s nécessaire des données en continue.

On a mis au point un WorkerThread qui permet d'initialiser un signal qui va être émis ensuite de manière continue dans une boucle **while**. Le signal est associé à une méthode qui est ainsi répétée périodiquement (ici toutes les 5sec, et dans notre cas toutes les 0.01sec).

```python
import sys
import time
from PyQt5 import QtWidgets, QtCore

class WorkerThread(QtCore.QObject):
    signalExample = QtCore.pyqtSignal(str, int)# signal initialisé qui permet l'envoie d'une str et d'un int

    def __init__(self):
        super().__init__()

    @QtCore.pyqtSlot()
    def run(self):
        while True: #Boucle while, qui permet l'émission continue du signal
            # Long running task ...
            self.signalExample.emit("leet", 1337) # Emission toutes les 5sec du mot "leet" et int 1337
            time.sleep(5)# Temps de rafraichissement de 5sec

class Main(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()
        self.worker = WorkerThread()
        self.workerThread = QtCore.QThread()
        self.workerThread.started.connect(self.worker.run)
        self.worker.signalExample.connect(self.signalExample)  # Connecttion du signal/slots
        self.worker.moveToThread(self.workerThread)
        self.workerThread.start()

    def signalExample(self, text, number):#L'émission du signal est associée à la méthode permettant l'afficha
        print(text)
        print(number)

if __name__ == '__main__':
    app = QtWidgets.QApplication([])
    gui = Main()
    sys.exit(app.exec_())
```

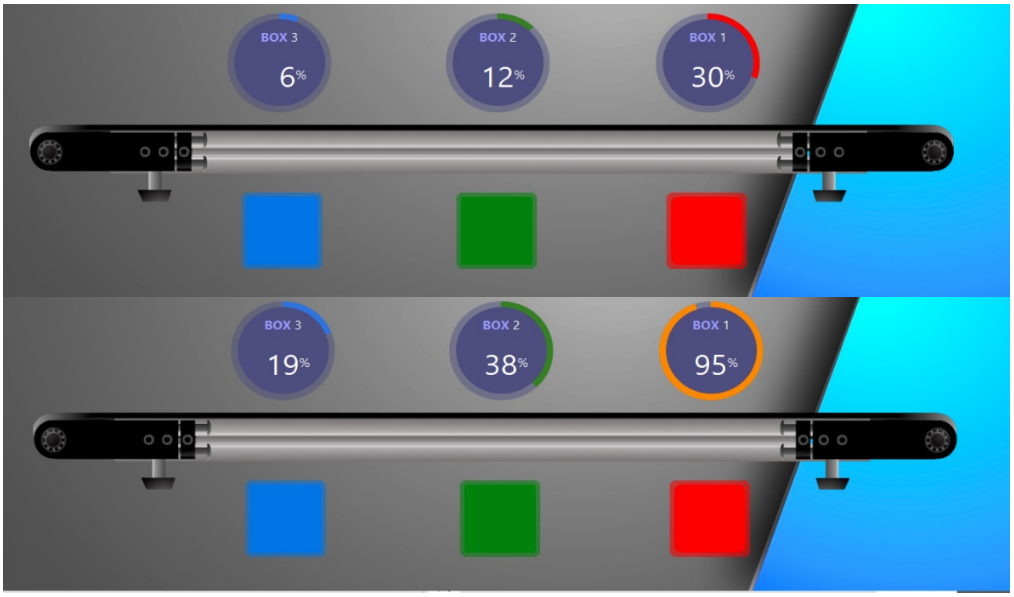
Dans notre cas, on veut rafraichir régulièrement les pourcentages de remplissage, on a utilisé le signal :

```python
signalContinu = QtCore.pyqtSignal(str) #Init fonction émise à période régulière
```

Auquel la fonction répétée périodiquement est :

```python
    def signalContinu(self, text):
        """Méthode associée au signal que l'on émet à période régulière.
        A chaque période, on refresh le pourcentage et on le remplie graphiquement

        Parameters
        ----------
        text : texte à afficher à période régulière dans la console

        """

        print(text)
        global marche
        print(marche)
        self.changeTestpourcentage()

        self.set_pourcentage([pourcentage_box[0], pourcentage_box[1], pourcentage_box[2]])
```

Dans la fonction du signal on peut donc mettre d'autres méthodes qui vont être appelées périodiquement comme **set_pourcentage()** qui vient récupérer les pourcentages en globales comptés selon le programme de détection de couleurs/formes et les renouveler graphiquement en redessinant la roue chromatique régulièrement :



A 100% de remplissage de l'une des boîtes on fournit « marche = 0 », arrêtant les moteurs.

Pourcentages réels :

On améliore alors la page de fonctionnement du tapis. Les pourcentages ne sont plus changés par la méthode d'incrémentation (changeTestPourcentage()) mais par les quantités physiques détectées. Soit avec pourcentage_box = [pourcentage_box[0], pourcentage_box[1], pourcentage_box[2]] qui changent selon le dictionnaire remplissage mis à jour dans la détection de couleurs et de formes.

```
def signalContinu(self, text, image):
    """Méthode associée au signal que l'on émet à période régulière.
    A chaque période, on refresh le pourcentage et on le remplie graphiquement

    Parameters
    ----------
    text : texte à afficher à période régulière dans la console

    """

    print(text)
    print(marche)
    # self.changeTestpourcentage()


    self.set_pourcentage([pourcentage_box[0], pourcentage_box[1], pourcentage_box[2]])

    self.label_video.setPixmap(QtGui.QPixmap.fromImage(image))

    config.current_slider = self.slider.value()
    # print(current_slider)
```

```
pourcentage_box[0] = config.remplissage_color[config.color]
```

Config.remplissage_color est le dictionnaire qui compte les cubes de couleurs détectés (vu en partie détection de couleurs) et **config.color** la couleur détectée à l'instant t de la détection. Ils sont placés dans le dossier config, pour les avoir en variables globales entre les modules.

Pourcentage_box[i] lit le compteur de la couleur associé à la boite i.

Flux vidéo dans PyQt :

On ajoute le flux vidéo dans PyQt en plaçant un label (comme les images on met un fond pour le label pour afficher l'image). Cependant, on rafraîchit l'image régulièrement, ce qui fait voir à l'œil un flux caméra.

Comme pour une image on définit un label dans le setup avec sa position… :

```
#Photo acquise en continue
self.label_video = QtWidgets.QLabel(self.page_3)
self.label_video.setGeometry(QtCore.QRect(L_WINDOW//5+L_WINDOW//7-L_WINDOW//105, H_WINDOW//50, L_WINDOW//3+L_WINDOW//60,
self.label_video.setText("")
self.label_video.setPixmap(QtGui.QPixmap("./bleu.png"))
self.label_video.setObjectName("label_video")
```

```
self.label_video.raise_()
```

Dans la méthode de rafraichissement, on place l'image a envoyé régulièrement et on appelle pour changer l'image :

```
self.label_video.setPixmap(QtGui.QPixmap.fromImage(image))
```

Dans la boucle **while** du thread, l'image récupérée de la caméra frame doit ainsi être actualisée :

```
img_convoyeur = frame
```

```
else:
    img_gray_conv = cv2.cvtColor(img_convoyeur, cv2.COLOR_BGR2GRAY)

    img_gray_conv = cv2.blur(img_gray_conv, (4, 4))

    # seuil = cv2.getTrackbarPos("seuil", "Seuillage")
    seuil = config.current_slider

    # Seuillage binaire tel que niveau_de_gris > 40 on met en blanc=255 les formes
    ret, thresh = cv2.threshold(
        img_gray_conv, seuil, 255, cv2.THRESH_BINARY)

    rgbImage = cv2.cvtColor(thresh, cv2.COLOR_BGR2RGB)

h, w, ch = rgbImage.shape
bytesPerLine = ch * w

convertToQtFormat = QtGui.QImage(rgbImage.data, w, h, bytesPerLine, QtGui.QImage.Format_RGB888)

p = convertToQtFormat.scaled(L_WINDOW//2, H_WINDOW//2, QtCore.Qt.KeepAspectRatio)

# Boucle continue de refresh de l'interface
self.signalContinu.emit("Acquisition", p)
```

Dans le cas du choix de la détection de formes, on refait les traitements de binarisation de l'image qu'on veut afficher, puis il faut convertir l'objet OpenCv en image adaptée au format Qt. L'image au format Qt est ici appelée p. On envoie régulièrement p avec **signalContinu.emit**(« str », p).

Le curseur (config.current_slider) permet de changer le seuil de binérisation (formes) ou le seuil de saturation (couleurs)

Voici la fenêtre finale de l'application dans le cas de la détection de couleur :

Figure 32 : Acquisition vidéo dans le cas de la détection de couleurs

Intégration

Association des 3 briques de base

Cette partie a été difficile car chaque programme avait été divisé en module indépendants, chacun fonctionnant sur sa propre boucle, et de plus avec une fréquence de rafraîchissement différente. Nous avons dû finalement insérer l'entièreté du programme dans celui de l'interface graphique, pour faire fonctionner en particulier la caméra, sans avoir 2 boucles **while** en parallèles. Tout fonctionne sur le thread qui constitue une routine en parallèle qui rafraîchit les images de l'interface graphique, envoie les informations aux servomoteurs et récupère l'image de la caméra.

De plus, il a fallu faire communiquer la Nucléo avec Python pour échanger les données, en particulier lors de l'envoi des informations aux servomoteurs lorsque la couleur/forme est détectée (Python->Nucléo), et lors de la détection par la cellule photo détectrice TOR (Nucléo -> Python).

Emission/réception de signaux entre Python et la Nucléo

L'envoi et la réception de signaux entre la Nucléo et Python se font par le bais d'une liaison série. Il suffit alors d'écrire « write » pour envoyer un message et « read » pour réceptionner un message.

Prenons le cas du démarrage du moteur envoyé par Python et la détection par la cellule TOR envoyée par la Nucléo.

Envoi d'une donnée depuis Python et réception par la Nucléo :

1- Initialiser une liaison série côté Python

On établit dans Python, une liaison série avec la bibliothèque serial. On complète le **serial_com** par le numéro de COM du PC. Ensuite on réalise une liaison à 115 200 bauds :

```
# ================================================================================
# Création de la connection entre la nucléo et l'ordinateur
# ================================================================================

import serial
import serial.tools.list_ports

serial_com = 5
serial_com_name = 'COM'+str(serial_com)

# Ouvre le serial port
ser = serial.Serial(serial_com_name, baudrate=115200)
```

2- Envoyer des données à la Nucléo

Dans le cas du démarrage du moteur, l'appui sur le bouton start fait passer marche à 1. On envoie la donnée à la Nucléo en écrivant :

```
ser.write(bytes('r', 'utf-8'))
```

Write permet d'envoyer des str ou des char. On utilisera uniquement des char à envoyer. On utilisera 'r' pour run envoyer à la Nucléo ce qui correspond à 1 bit.

```
def start(self):
    """
    Démarre le process : moteurs on grâce à marche = 1

    """
    print('run')
    global marche, servo_retour
    marche = 1
    ser.write(bytes('r', 'utf-8'))
    servo_retour = 0 #Les servo ne sont plus forcés en position minimale
    print("Moteurs en route")
```

3- Réception de la donnée côté Nucléo

Par exemple, nous avons mis en place un bouton arrêt/marche pour arrêter ou faire fonctionner le convoyeur. Cela se traduit dans le code de la Nucléo par la réception du caractère 'r' pour la mise en marche au sein de la fonction de réception (voir partie suivante la configuration de la liaison série Nucléo-PC par le port USB en C++)

```
// Information de lancement global du convoyeur
if(data == 'r')
{
    run = 1;
}

// Information d'arret/relance global du convoyeur
if(data == 's')
{
    run = 0;
}
```

Puis on retrouve cette donnée dans une boucle **if** qui permet d'arrêter ou de faire fonctionner le convoyeur dans le **while** du main :

```
if(run == 1)
{
    enable=1;
    led_2 = 1;
    led = 0;
}
if(run == 0)
{
    enable=0; // Mise en marche enable et reset obligatoire
    led_2 = 0;
    led = 1;
}
```

Remarque : la condition **run = 1** est aussi présente dans la fonction de décrémentation. Cela permet de ne pas décrémenter dans le vide sachant que le convoyeur est à l'arrêt.

Envoi d'une donnée depuis la Nucléo et réception par Python :

Dans le cas de la cellule détectrice TOR, lorsqu'un objet passe, alors la variable **detecteur = 1**. La Nucléo doit signaler à Python le passage de l'objet et ainsi envoyer le char « d ».

1- Initialiser une liaison série côté Nucléo

Comme dit précédemment la Nucléo est connecté avec le PC, est des informations peuvent être échangées entre les 2. Pour initialiser la liaison série à 115 200 bauds, il faut renseigner ces lignes de code avant le main :

```
// Connection au programme python pour l'envoie et la reception de données
UnbufferedSerial      my_pc(USBTX, USBRX);
char data;
void reception_pc_nucleo(void);
```

my_pc est l'objet associé à la transmission (TX) et réception (RX) par le port USB.

On définit la fonction **reception_pc_nucleo.**

Puis dans l'initialisation dans le main avant la boucle while, on attache la fonction de réception à my_pc. (pour la partie précédente pour la réception de données)

```
//CONNECTION PYTHON
my_pc.baud(115200);
my_pc.attach(&reception_pc_nucleo, UnbufferedSerial::RxIrq);
```

2- Envoyer des données à Python

Pour envoyer le caractère « d » à Pyhton lors de la détection (le détecteur TOR fait passer sa variable à detecteur = 1), on écrit **my_pc.write(&message, 1)**, où 1 signifie l'envoie d'un seul bit = 1 caractère.

```
my_pc.write(&message,1);
```

```
if(detecteur == 1 && delay==0) // si le capteur a detecté un objet et si le délai de detection est à
{
    char message = 'd';
    my_pc.write(&message,1);// on envoie l'info au pc comme quoi on a eu une detection donc il faut prendre une photo
    delay = 100;// réduit ma frame
}
```

3- Réception des données dans Python

On applique **ser.readline(1)** afin de lire l'unique caractère envoyé par la Nucléo. Il est vide lorsque la détection n'a pas lieu et passe à « d ». Cependant le caractère est envoyé en ASCII après avoir été converti en binaire dans l'envoi de la liaison série. Il faut donc décoder le bit avec **decode().** On réceptionne alors la valeur « d » côté Python.

```
detecteur = ser.readline(1)
detecteur_value = detecteur.decode()
# on reçoit "d" = détection par la Nucléo lorsque l'objet passe devant le détecteur
```

Emission/réception de signaux entre l'interface graphique et Python

Les informations en particulier de remplissage (compteur pour les pourcentages) et la forme renvoyée, sont pour plus d'aisance partagées en variables globales inter modules à l'aide d'un fichier config. Ce fichier config détient toutes les variables qui changent souvent d'un programme à l'autre, et que l'on veut récupérer de n'importe quel module en l'important. Cependant, comme finalement on a regroupé tous les modules en un unique module dans l'interface graphique suite au problème des boucles **while** en parallèle, et que ces variables sont retournées, ce fichier ne serait plus nécessaire mais il faudra effectuer tous les changements pour rester local au module.

Variables dans le fichier config initialisées :

```
# choice = True # True couleur, false forme
# choice_IA = False # Activation ou non de l'IA

detecteur = ""

remplissage_color = {'ROUGE': 0, 'ORANGE': 0, 'JAUNE': 0, 'VERT_CLAIR': 0,
            'VERT_FONCE' : 0, 'BLEU_CLAIR': 0, 'BLEU_FONCE': 0,
             'VIOLET': 0, "ROSE" : 0, "MARRON" : 0, "INDEFINI" : 0}
color = 0

remplissage_shape = {'TRIANGLE': 0, 'CARRE': 0, 'RECTANGLE': 0,
          'PENTAGONE': 0, 'HEXAGONE' : 0, 'ETOILE': 0, 'CERCLE': 0}
shape = 0

current_slider = 120

choice = True
```

Bilan

Partie technique/Avancement final

Nous restons un petit peu sur notre faim pour ce projet. En effet, chaque brique de notre projet fonctionne parfaitement. Nous avons le plan précis en tête du rendu final du projet. Cependant, par manque de temps (pas d'ambition) nous n'avons pas un résultat final pour notre système qui fonctionne.

Nous avons quelques problèmes concernant le traitement de l'image capturée par la caméra par le programme informatique (OpenCV ou IA) au niveau de l'intégration et mise en commun des modules. Cependant, avec une webcam, le programme marche parfaitement ou encore lancé seul avec la caméra uEye. Le programme de traitement d'image fonctionne, mais il est impossible à l'heure actuelle pour nous de fournir l'image au programme pour qu'il détermine la caractéristique de l'objet.

De plus, le programme des tickers pour faire fonctionner les 3 servomoteurs indépendamment, fonctionne seul en envoyant au clavier le moteur à pousser. Mais intégrer aux autres modules, là où pourtant la variable envoie le bon moteur, la réception n'engendre pas la poussée du bon servomoteur.

Cf. fonctionnement en vidéo en envoyant du pc le servomoteur à pousser

---

La prolongation possible serait d'observer au niveau du thread (boucle **while**) dans l'interface graphique ce qui ne fonctionne pas. Il faudrait également regarder au niveau du **delay** du photodétecteur qui mis à nul engendre un flux continu (ce qu'on voulait et là le programme fonctionne et même la bonne couleur est détectée mais il détecte infiniment la forme au passage du cube sur 10ms). Il faudrait également plus d'informations sur le traitement multi-tâche et le parallélisme de tâches en informatique.

Cependant, nous avons réalisé ce projet dans l'objectif non pas de l'accomplir mais d'aller au-delà pour en apprendre davantage en informatique et en électronique (nous en sommes très fiers). Nous avons appris l'utilisation plus poussée d'OpenCv pour allumer une webcam et faire de la détection de formes, réaliser une introduction au deep learning qui nous passionne et développer une interface graphique plus sophistiquée qu'avec Tkinter avec de simples boutons. On n'avait jamais réalisé de gros programmes qui fonctionnent en continu avec une boucle **while** entre différents systèmes, ce qui au niveau des interactions, des temps de pause et de la fréquence de rafraîchissement nous a posé des problèmes. Nous aurions souhaité aller au bout mais quelques séances en plus auraient été suffisantes.

Retour d'expérience de l'équipe

Il était au début compliqué de se répartir les tâches car comme nous étions un petit groupe, nous avions besoin de l'aide de l'ensemble du groupe pour avancer sur les différentes tâches.

Par exemple, comprendre le fonctionnement des servomoteurs et du moteur pas à pas et réussir à les faire fonctionner avec la Nucléo, a nécessité la réflexion du groupe entier sur plusieurs séances.

Cependant, après s'être lancé et avoir compris la plupart des objectifs que nous nous étions fixés, les différents membres du groupe se sont focalisés sur des parties plus spécifiques. Par exemple, l'interface utilisateur et la détection de formes et couleurs ont été faites pour la majeur partie par Julien, alors que l'intelligence artificielle et le contrôle des servomoteurs avec les tickers a été fait majoritairement par Thomas. La partie électronique du départ a été réalisée ensemble.

Nous avons développé de nombreuses compétences grâce à ce projet dans le domaine de l'informatique ou de l'électronique. La capacité à réussir à faire communiquer des langages entre eux en est un exemple.

Par ailleurs nous avons renforcé nos compétences en soudure tout au long de ce projet en créant de nombreux adaptateurs pour nos branchements.

Structure du projet

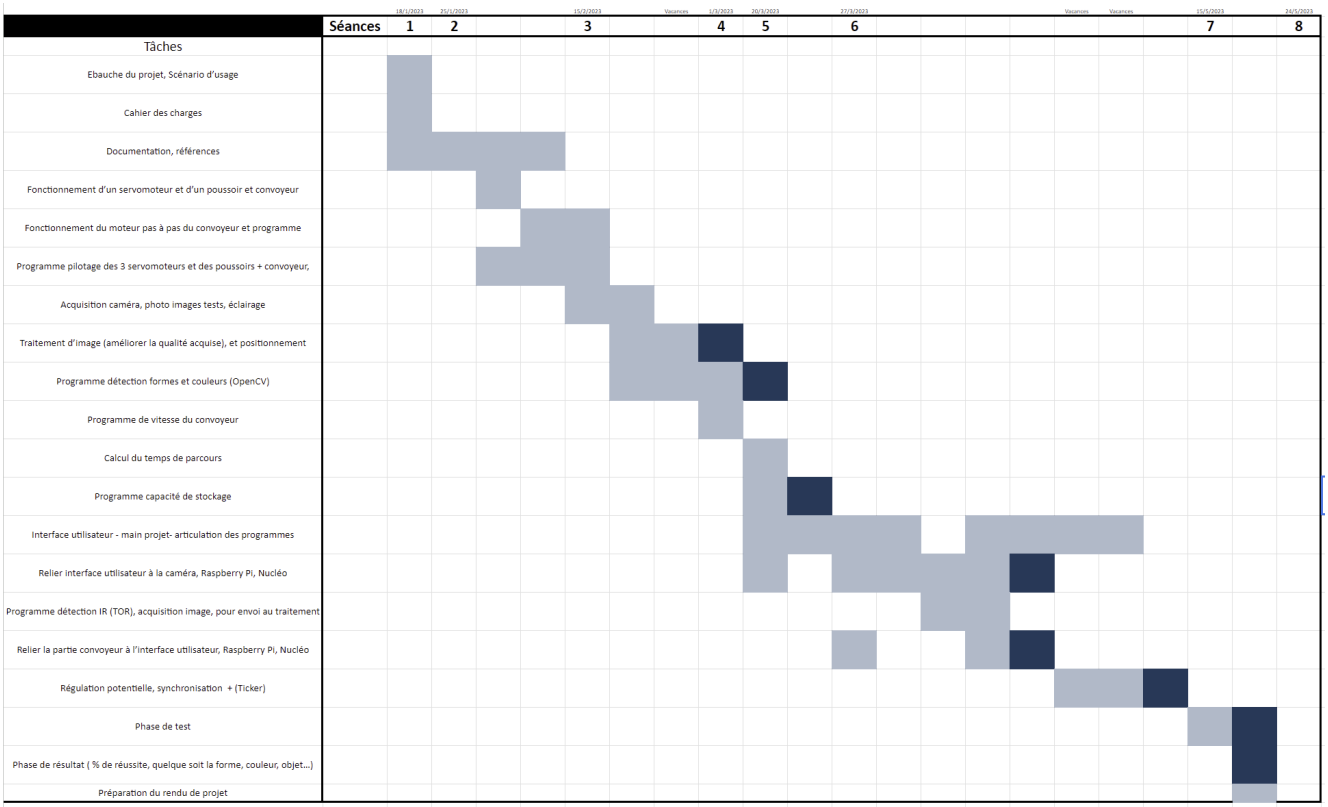
Le projet va se segmenter en différentes parties. Plusieurs étapes doivent être respectées pour le bon déroulement du projet. Nous notons ici les grandes phases du projet :

- Pilotage des moteurs et du convoyeur avec la carte Nucléo.
- Acquisition des images avec la caméra et débuter le traitement d'image.
- Création de l'interface utilisateur.
- Relier la carte Raspberry Pi à la carte Nucléo et mettre en marche le système.

Diagramme de Gantt

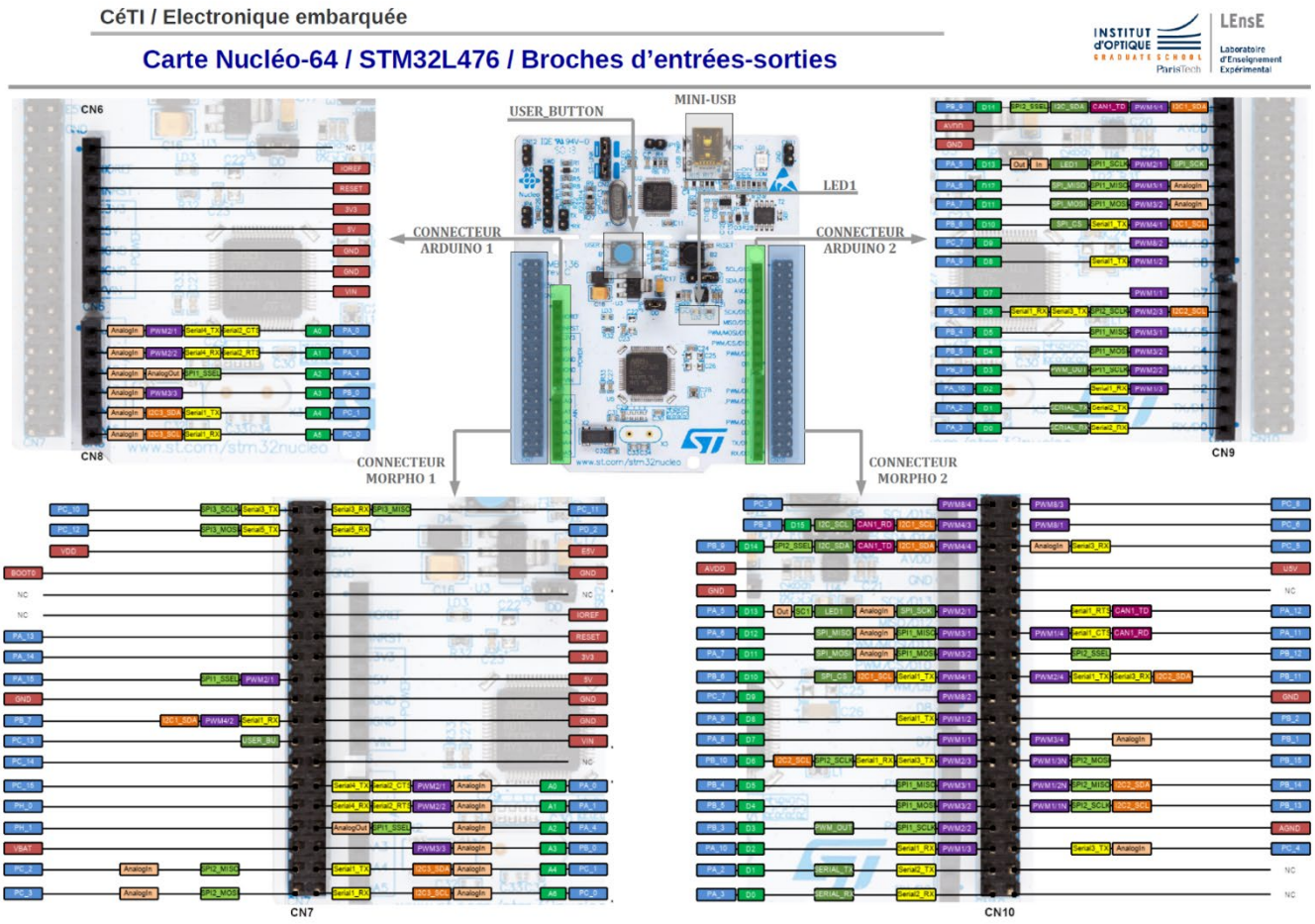
Nous répartissons les différentes tâches du projet dans un diagramme de Gantt pour segmenter le projet et repartir les tâches parmi les membres du groupe.

Toutes les cases en bleu claire ont été réalisées. Cependant, quelques cases sont restées foncées, car la partie n'était pas finalisée à 100% pour avoir un pourcentage de réussite du fonctionnement optimal.



Annexes

Entrées et sorties de la carte Nucléo (source : LEnsE)



Intelligence artificielle

DEEP LEARNING

5- => Méthode de Descente de Gradient :

va permettre d'ajuster les paramètres w_i et b de façon à minimiser les erreurs donc minimiser la fonction coût.

On calcule donc le gradient de la fonction coût :

On cherche le minimum.

On pose la formule suivante :

$$w_{t+1} = w_t - \alpha \frac{\partial \mathcal{L}}{\partial w_t}$$

$\alpha > 0$

- C'est une sorte de méthode de Newton mais pour n dimensions avec $n \in \mathbb{N}^*$

=> Cependant la descente de gradient ne peut pas différencier les minimums locaux et globaux.

Pour un perceptron on a :

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial a} \frac{\partial a}{\partial z} \frac{\partial z}{\partial w_1} = -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} - a^{(i)} \right) x_1^{(i)}$$

avec : $\frac{\partial \mathcal{L}}{\partial a} = -\frac{1}{m} \sum_{i=1}^{m} \frac{y^{(i)}}{a^{(i)}} - \frac{1-y^{(i)}}{1-a^{(i)}}$; $\frac{\partial a}{\partial z} = a(1-a)$; $\frac{\partial z}{\partial w_1} = x_1$

puis de même : $\frac{\partial \mathcal{L}}{\partial w_2} = -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} - a^{(i)} \right) x_2^{(i)}$

$$\frac{\partial \mathcal{L}}{\partial b} = -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} - a^{(i)} \right)$$

2

6- => Avant de coder il faut comprendre qu'en Deep Learning on a beaucoup de données. Ainsi il faut vectoriser les données pour gagner beaucoup de temps :

On note $X = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} \\ x_1^{(2)} & x_2^{(2)} \\ \vdots & \vdots \\ x_1^{(m)} & x_2^{(m)} \end{pmatrix}$ et $Y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y_1^{(m)} \end{pmatrix}$

avec $W = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix}$ et $B = \begin{pmatrix} b \\ \vdots \\ b \end{pmatrix}$

donc on a : $z^{(i)} = w_1 x_1^{(i)} + w_2 x_2^{(i)} + b$ donne :

$$\boxed{Z = X \cdot W + B}$$

Ensuite on note : $A = \begin{pmatrix} a^{(1)} \\ \vdots \\ a^{(m)} \end{pmatrix} = \sigma(Z)$ car $a^{(i)} = \sigma(z^{(i)})$

=> Pour la Descente de Gradient :

$$\boxed{W = W - \alpha \frac{\partial \mathcal{L}}{\partial W}} \qquad \boxed{B = B - \alpha \frac{\partial \mathcal{L}}{\partial B}}$$

=> Pour les différents Gradients :

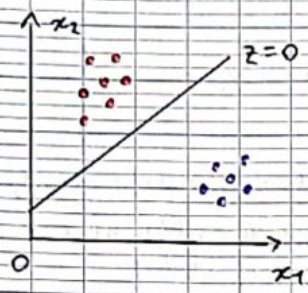
$$\boxed{\frac{\partial \mathcal{L}}{\partial W} = \frac{1}{m} X^t \cdot (A - Y)} \qquad \boxed{\frac{\partial \mathcal{L}}{\partial B} = \frac{1}{m} \sum (A - Y)}$$

=> Pour la fonction coût :

$$\boxed{\mathcal{L} = -\frac{1}{m} \sum Y \log(A) + (1-Y) \log(1-A)}$$

DEEP LEARNING

=> Traçant la frontière de décision ; elle représente les points où $z = 0$ donc ceux ayant une probabilité de 0,5 (le zéro de la sigmoïde)



On a donc $w_1 x_1 + w_2 x_2 + b = 0$

donc : $x_2 = \frac{-w_1 x_1 - b}{w_2}$

ce qui permet de tracer la frontière de décision sur un graphe après avoir calculé le modèle.

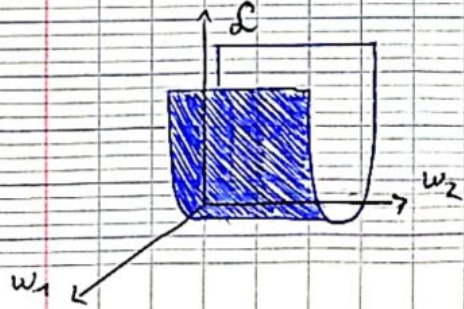
=> Remarque : Pour la méthode de descente de gradient, il faut toujours normaliser les données d'entrée.

↳ En effet la fonction sigmoïde a un $\exp(-z)$ ce qui provoque des problèmes de trop grands nombres

ex : si on a $x_1 : [0;1]$ et $x_2 : [0;10]$

avec $z(x_1; x_2) = w_1 x_1 + w_2 x_2 + b$. w_1 et w_2 n'ont pas le même impact sur l'activation

Ici w_2 va avoir beaucoup plus d'impact. Si on trace la fonction coût en fonction de w_1 et w_2 on va avoir quelque chose comme :



La fonction coût est "compressée"
Cela complique la descente de gradient.

Ainsi pour normaliser, on peut utiliser la normalisation MinMax :

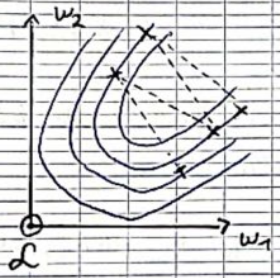
$$X = \frac{X - X_{min}}{X_{max} - X_{min}}$$

avec X_{min} et X_{max} les valeurs max et min que peut atteindre la donnée avant la normalisation

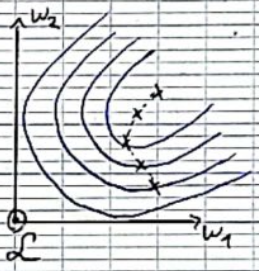
9- => Réglage des hyper-paramètres :

On a α qui va correspondre au pas entre chaque itération. En effet si α est trop il se peut que le modèle ne converge jamais :

α grand



α petit

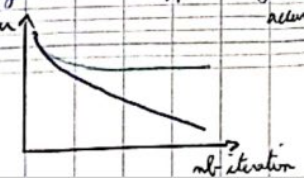


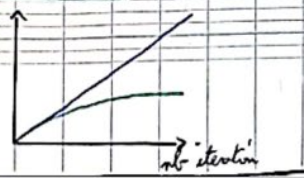
10- Diagnostique d'over-fitting :

Si on met à disposition du modèle, des données "d'entrainement" et de "test". Les données d'entrainement permettent au modèle de trouver les bons paramètres pour minimiser l'erreur de la fonction coût.

Les données test permettent de tester le modèle sur de nouvelles données.

=> Il peut arriver que le modèle soit en over-fitting, c'est à dire qu'il ne généralise plus et se "spécialise" sur les données d'entrainement. Et stagne sur l'apprentissage des données test :





DEEP LEARNING

Comment régler le problème ?

⇒ Avoir plus de données pour pouvoir généraliser, on réduire le nombre de paramètre.
De plus si on a plus de variables d'entrée que de donnée, on obtient un phénomène qu'on appelle : le fléau de la dimension.

↓

espace remplit principalement de vide où le modèle fait ce qu'il veut.

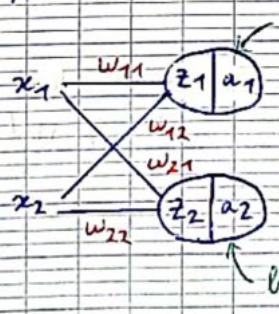
⇒ De plus le modèle, pour l'instant, n'est constitué que d'un neurone. Il faut rajouter d'autres neurones pour complexifier la différenciation

II - Réseau de neurone (2 couches) :

1=

⇒ On passe à un réseau de neurone pour s'affranchir du côté linéaire qu'a un seul neurone pour pouvoir résoudre des problèmes plus complexes. (non linéaire)

On part de cette structure :



$$\begin{cases} z_1 = w_{11} x_1 + w_{12} x_2 + b_1 \\ a_1 = \dfrac{1}{1+e^{-z_1}} \end{cases}$$

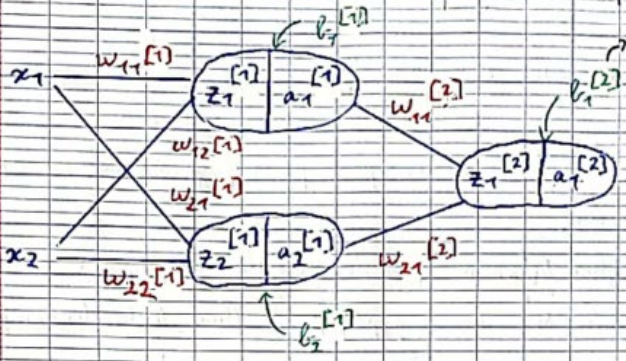
$$\begin{cases} z_2 = w_{21} x_1 + w_{22} x_2 + b_2 \\ a_2 = \dfrac{1}{1+e^{-z_2}} \end{cases}$$

qui est un réseau de neurone à 1 couche.

w_{xy} = paramètre associé au neurone x et provenant de l'entrée y.

On note $\begin{cases} n^{[x]} : \text{le nombre de neurone sur la couche } x \\ m : \text{le nombre d'itération} \end{cases}$

On peut par la suite rajouter autant de couche que l'on veut :



$$\begin{cases} z_1^{[2]} = w_{11}^{[2]} a_1^{[1]} + w_{12}^{[2]} a_2^{[1]} + b_1^{[2]} \\ a_1^{[2]} = \dfrac{1}{1+e^{-z_1^{[2]}}} \end{cases}$$

qui est un réseau de neurone à 2 couches.

=> On peut rajouter autant de couche que l'on veut et autant de neurone par couche que l'on veut. (Deep Neural Network)

2- On va vectoriser les equations pour n neurones et n couches.

On note $W^{[1]} = \begin{pmatrix} w_{11}^{[1]} & w_{12}^{[1]} \\ w_{21}^{[1]} & w_{22}^{[1]} \end{pmatrix}$; $B^{[1]} = \begin{pmatrix} b_1^{[1]} \\ b_2^{[1]} \end{pmatrix}$

Puis on prend X^t du perceptron pour devenir X ici :

On a donc : $\boxed{Z^{[1]} = W^{[1]} X + B^{[1]}} = \begin{pmatrix} z_1^{[1](1)} & \dots & z_1^{[1](m)} \\ z_2^{[1](1)} & \dots & z_2^{[1](m)} \end{pmatrix}$

et on pose : $Y = \begin{pmatrix} y^{(1)} & \dots & y^{(m)} \end{pmatrix}$

Remarque : Si on veut ajouter un neurone sur la couche, il suffit juste de rajouter une ligne sur $B^{[1]}$ et $W^{[1]}$

Puis on a $\boxed{A^{[1]} = \dfrac{1}{1+e^{-Z^{[1]}}}}$

=> Ensuite on peut realiser la même chose vers les autres couches en prenant les activations des couches comme entrées.

On appelle cela la : forward propagation.

Dimensions : $X \in \mathbb{R}^{n^{[0]} \times m}$ $\quad Y \in \mathbb{R}^{1 \times m}$

$W^{[c]} \in \mathbb{R}^{n^{[c]} \times n^{[c-1]}}$

$B^{[c]} \in \mathbb{R}^{n^{[c]} \times 1}$

$Z^{[c]}, A^{[c]} \in \mathbb{R}^{n^{[c]} \times m}$

pour la couche de neurone c.
pour m itérations

DEEP LEARNING

3- Entrainement du réseau de neurone :

=> On va appliquer la back propagation, qui va consister à faire les mêmes étapes que pour un seul neurone :

Fonction Coût → Calcule des gradients → MAJ de W et b.

La back-propagation consiste à retracer comment la fonction coût évolue de la dernière couche du réseau jusqu'à la toute première.

=> On peut donc calculer les gradients (par exemple de l'exemple à 2 couches) :

On pose $dZ2 = \frac{\partial \mathcal{L}}{\partial A^{[2]}} \frac{\partial A^{[2]}}{\partial Z^{[2]}}$ et $dZ1 = dZ2 \frac{\partial Z^{[2]}}{\partial A^{[1]}} \frac{\partial A^{[1]}}{\partial Z^{[1]}}$

On cherche les gradients pour réaliser la méthode de descente de gradient :

$$\frac{\partial \mathcal{L}}{\partial W^{[2]}} = dZ2 \frac{\partial Z^{[2]}}{\partial W^{[2]}} \qquad \frac{\partial \mathcal{L}}{\partial W^{[1]}} = dZ1 \frac{\partial Z^{[1]}}{\partial W^{[1]}}$$

$$\frac{\partial \mathcal{L}}{\partial B^{[2]}} = dZ2 \frac{\partial Z^{[2]}}{\partial B^{[2]}} \qquad \frac{\partial \mathcal{L}}{\partial B^{[1]}} = dZ1 \frac{\partial Z^{[1]}}{\partial B^{[1]}}$$

On trouve les résultats suivants : $\boxed{dZ2 = (A^{[2]} - Y)}$ $(n^{[2]}; m)$

$(n^{[2]}; n^{[1]})$ $\boxed{\frac{\partial \mathcal{L}}{\partial W^{[2]}} = \frac{1}{m} dZ2 \cdot A^{[1]t}}$

$(n^{[2]}; 1)$ $\boxed{\frac{\partial \mathcal{L}}{\partial B^{[2]}} = \frac{1}{m} \sum_{\text{axe 1}} dZ2}$

puis : $\boxed{dZ1 = W^{[2]t} \cdot dZ2 \times A^{[1]}(1 - A^{[1]})}$ $(n^{[1]}; m)$

$\boxed{\frac{\partial \mathcal{L}}{\partial W^{[1]}} = \frac{1}{m} dZ1 \cdot X^{t}}$ $(n^{[1]}; n^{[0]})$

$\boxed{\frac{\partial \mathcal{L}}{\partial B^{[1]}} = \frac{1}{m} \sum_{\text{axe 1}} dZ1}$ $(n^{[2]}; 1)$

III- Réseau de neurone (n couches) :

=> Concernant les paramètres des autres couches on a :

$$W^{[c]} \in \mathbb{R}^{n^{[c]} \times n^{[c-1]}}$$
$$B^{[c]} \in \mathbb{R}^{n^{[c]} \times 1}$$

=> Concernant la forward propagation on a :

$$Z^{[c]} = W^{[c]} \cdot A^{[c-1]} + B^{[c]}$$

$$A^{[c]} = \frac{1}{1 + e^{-Z^{[c]}}}$$

=> Concernant la back propagation on a :

$$\frac{\partial \mathcal{L}}{\partial W^{[c]}} = \frac{1}{m} dZ^{[c]} \cdot A^{[c-1]^T}$$

$$\frac{\partial \mathcal{L}}{\partial B^{[c]}} = \frac{1}{m} \sum_{\text{axe 1}} dZ^{[c]}$$

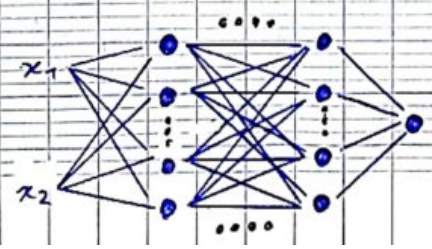
$$dZ^{[c-1]} = W^{[c]^T} \cdot dZ^{[c]} \times A^{[c-1]}\left(1 - A^{[c-1]}\right)$$

et pour la couche finale :

$$dZ^{[cf]} = A^{[cf]} - Y$$

Remarque : Plus un réseau de neurone est profond plus il a de chance de se perdre dans son apprentissage

=> vanishing gradient.



Remarque : Normalement, plus il y a de couches plus le réseau va être performant. Or cela prend plus de temps à être entraîné en contrepartie.