Rapport technique: Banc optique pour asservissement de position d'un laser (projet 2021_K021)

Groupe 3 : Eddy Barlogis, Marie-Hélène Carron, Maël Hubert, Mohamed Meguebel

Du 25 janvier 2022 au 8 avril 2022

I. Contexte du projet et solution technique

1. Descriptif du projet

La société Solec a proposé à notre équipe, conformément à l'offre 2021_K021 qu'elle nous a proposée, de développer un dispositif permettant d'asservir en position un faisceau laser.
L'une des applications envisagées est l'observation astronomique par optique adaptative, en particulier la création d'étoiles artificielles. Dans ce contexte, l'atmosphère peut faire dévier le faisceau, ce qui constitue une perturbation. Celle-ci est alors corrigée par des miroirs de basculement contrôlés par un système d'asservissement.

L'objectif de la société Solec sera donc de proposer un produit permettant d'asservir un faisceau laser pointé sur un dispositif de photodétection via la commande de servomoteurs, chacun relié à un miroir de basculement. Un tel dispositif est représenté par la Figure I.1.1.

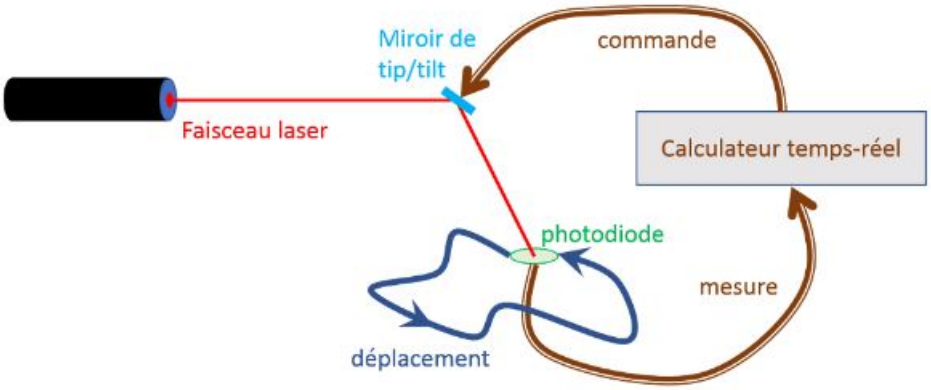


Figure I.1.1 : Schéma de principe du système d'asservissement souhaité par la société Solec, source : Caroline Kulcsar, professeure des universités, Institut d'Optique

2. Cahier des charges et contraintes matérielles

Le prototype que notre équipe doit développer est soumis à 3 contraintes :
- l'asservissement et la correction doivent être réalisés via un PID numérique
- les servomoteurs utilisés pour contrôler le mouvement des miroirs doivent être des servomoteurs classiques
- les coefficients du correcteur doivent pouvoir être modifiés en temps réel

De plus, les performances minimales attendues sont les suivantes :
- **Rapidité** : le système asservi doit permettre de suivre des mouvements de l'ordre de $10\,\mathrm{cm.s^{-1}}$
- **Précision** : l'erreur de pointage doit être la plus faible possible
- **Ergonomie** : l'interface Humain-Machine doit être facile à prendre en main, de préférence sous forme graphique

Par ailleurs, la société Solec nous conseille d'utiliser une liaison RS232 pour permettre les échanges entre l'interface et la partie matérielle ainsi qu'une carte Nucleo pour contrôler cette dernière.

3. Solution technique proposée

La solution technique proposée se décompose en deux pôles : la partie analogique et la partie numérique, chacune étant détaillée dans les sections dédiées. Ces deux pôles ont été développés simultanément pour permettre différents tests, aussi bien sur le fonctionnement des montages que sur le programme d'asservissement.

La partie analogique consiste en un montage optique comprenant entre autres un laser et une photodiode 4 quadrants ainsi qu'un montage électronique de photodétection relié à une carte Nucleo.
La partie numérique consiste en un programme d'asservissement codé sous Mbed et implémenté dans la carte Nucleo ainsi qu'une interface graphique sous Matlab.

La Figure I.3 résume cette solution technique sous forme de schéma-bloc et l'Annexe I.1 détaille le prototype sous forme de schéma fonctionnel.

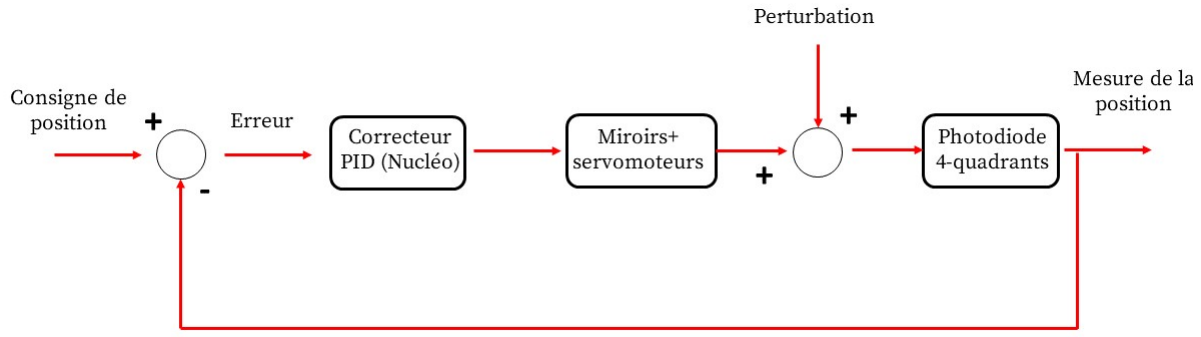


FIGURE I.3 : Schéma-bloc résumant la solution technique du projet

II. Partie analogique

L'idée générale de nos montages est résumée par la FIGURE II.0, le détail étant fourni dans les sous-sections suivantes. En outre, l'ANNEXE II.1 permet de visualiser le montage réel, les principaux éléments le constituant ayant été précisés.

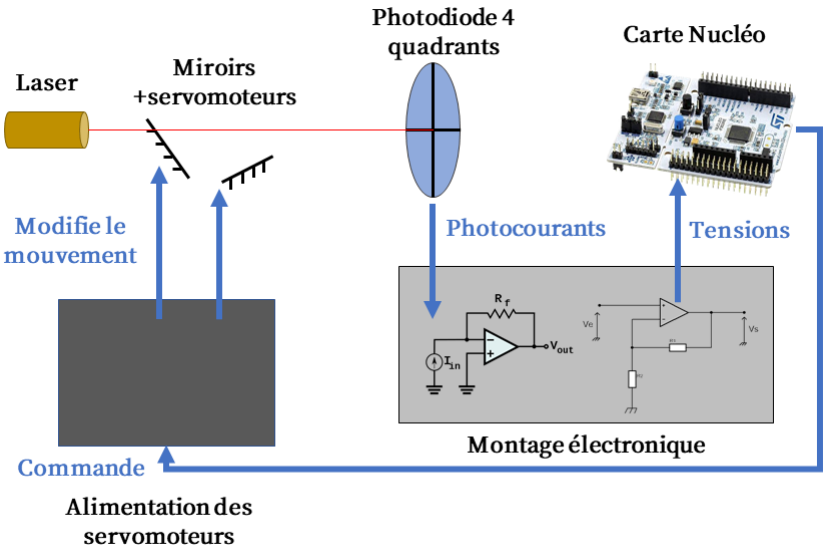


FIGURE II.0 : Schéma de principe de la partie analogique et liaison avec la carte NUCLEO

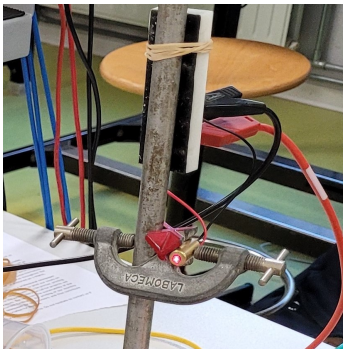
1. Montage optique

Ce montage consiste à permettre le positionnement du faisceau laser par rapport à un système de photodétection. Il est constitué d'un pointeur laser, d'un banc d'optique comprenant les miroirs et les servomoteurs associés et d'une photodiode 4 quadrants. Le choix des composants s'est porté sur :

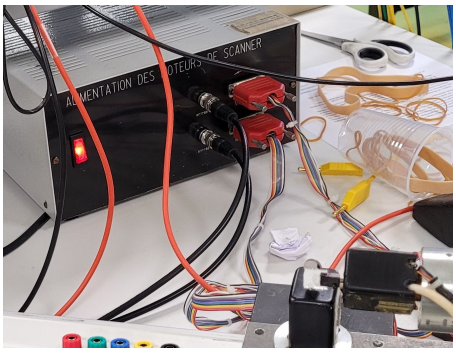
- un laser de Classe A **LFD650-0.4-12** de la marque Conrad, c'est un pointeur laser compact et présentant un très faible risque oculaire, c'est pourquoi nous avons opté pour ce modèle. Il sera alimenté avec une tension de 5 V.
- une photodiode 4 quadrants **SPOT-9DMI** de la marque OSI Optoelectronics. Le choix d'une telle photodiode s'explique par la volonté d'asservir les deux axes, X et Y, de manière indépendante, ce qui est possible en pratique en utilisant une photodiode 4 quadrants. Cette dernière a été entourée par un tube de papier cartonné afin d'éviter la saturation ou une quelconque perturbation provoquées par la lumière ambiante (voir FIGURE II.1.1.c).

La photodiode possède 5 pattes, l'une des pattes correspond à la masse et les 4 autres sont chacune reliée à un quadrant. Si une partie du faisceau est positionnée sur un quadrant, il apparaîtra un photocourant (un par patte) qui pourra être exploité en étant converti en une tension pour la carte NUCLEO. Cette conversion est l'objet de la sous-section suivante.

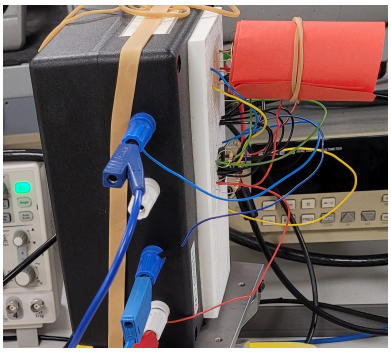
A chaque utilisation de ce montage, il faut bien noter la position d entre le dernier miroir du banc et la photodiode, cette position sera utilisée comme paramètre du programme d'asservissement. De plus, il faut bien positionner le faisceau laser en face de la photodiode afin que celui-ci ne soit pas complètement en dehors du détecteur dès le départ.



a. Pointeur laser



b. Miroirs de basculement et alimentation des servomoteurs



c. Photodiode 4-quadrants

FIGURE II.1.1 : Photographies du montage optique

Ceci passe notamment par un positionnement approprié du banc ainsi que de nombreuses translations verticales et horizontales du boîtier associé à la photodiode. Le positionnement du faisceau sur la photodiode étant particulièrement fastidieux, celui-ci peut être très long à mettre en place pour un expérimentateur peu habitué à ce type de montages. Une solution peut être l'ajout d'un générateur créant un offset analogique pour le laser, lequel permettrait alors de centrer le faisceau sur la photodiode.

Les servomoteurs sont commandés par une alimentation pour laquelle chaque câble est associé à un axe (le câble du haut pour l'axe X et le câble du bas pour l'axe Y, ces câbles représentés sur la photographie de la FIGURE II.1.1.b). C'est cette alimentation qui est effectivement commandée par la carte NUCLEO.

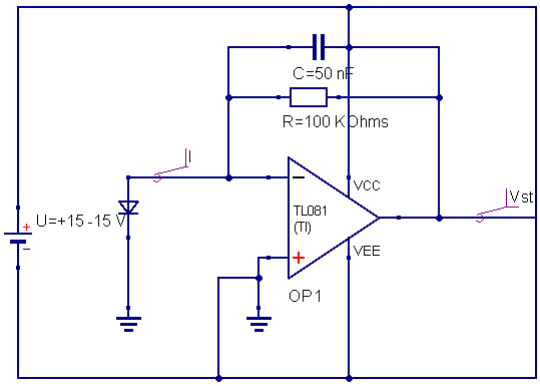
Pour asservir la position du faisceau, il faut pouvoir convertir une variation angulaire des miroirs en variation de tension. On peut déterminer les constantes K_x et K_y respectivement les constantes de conversion associées à l'axe X et à l'axe Y. Pour ce faire, on prend une feuille de papier millimétré et on mesure le déplacement maximal atteignable en X ou en Y puis l'écart en tension maximal correspondant.
On peut alors définir les constantes K_x et K_y, définies dans le programme d'asservissement, par :

$$K_x = \frac{\theta_{\mathrm{X,max}} - \theta_{\mathrm{X,min}}}{U_{\mathrm{X,max}} - U_{\mathrm{X,min}}} = 29.47\,\mathrm{rad.V^{-1}}$$

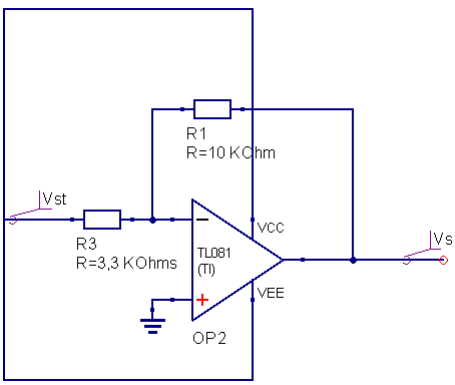
$$K_y = \frac{\theta_{\mathrm{Y,max}} - \theta_{\mathrm{Y,min}}}{U_{\mathrm{Y,max}} - U_{\mathrm{Y,min}}} = 42.12\,\mathrm{rad.V^{-1}}$$

2. Montage électronique

Ce montage a deux objectifs : permettre d'acquérir le signal des 4 quadrants de la photodiode et relier les 4 tensions associées à la carte NUCLEO. Pour ce faire, deux sous-montages sont réalisés : un montage de photodétection transimpédance décrit par la FIGURE II.2.a et un montage inverseur décrit par la FIGURE II.2.b.
Un schéma du montage électronique complet pour un photocourant (les branchements associés aux trois autres photocourants étant identiques) est également disponible en ANNEXE II.2.



a. Montage transimpédance



b. Montage inverseur

FIGURE II.2 : Schémas du montage électronique pour un photocourant réalisés sur QUCS

a. Montage transimpédance

Le montage transimpédance réalisé permet d'assurer la photodétection des 4 signaux provenant chacun d'un quadrant de la photodiode. Celui-ci permet de convertir le photocourant de chaque patte de la photodiode en une tension proportionnelle à ce photocourant selon l'équation :

$$V_{\mathrm{st_i}}(\omega) = \frac{R_{t_i} I_{\mathrm{phot_i}}}{1 + j R_{t_i} C_i \omega},\ i \in \{1,2,3,4\}$$

où $V_{\mathrm{st_i}}$, R_{t_i}, C_i et $I_{\mathrm{phot_i}}$ sont respectivement la tension de sortie, la résistance, la capacité du condensateur et le photocourant d'entrée du montage.

En ajoutant des condensateurs en parallèle des résistances, nous filtrons la fréquence de 50 Hz associée à l'éclairage ambiant via un filtre passe-bas du 1er ordre.
Pour faciliter la réalisation du montage des Amplificateurs Linéaires Intégrés (ou ALI), en particulier au niveau des masses et de l'alimentation du composant, nous avons utilisé un composant **TL-084** contenant 4 ALI dans un seul boîtier. Nous avons choisi les valeurs $R_{t_i} = 100\,\mathrm{k\Omega},\ i \in \{1,2,3,4\}$ pour les résistances et $C_i = 50\,\mathrm{nF},\ i \in \{1,2,3,4\}$ pour les capacités des condensateurs, donnant une fréquence de coupure $f_c = 1/2\pi R_{t_i} C_i = 32\,\mathrm{Hz} < 50\,\mathrm{Hz}$.

Nous mesurons en sortie de ce montage des tensions négatives et de valeur absolue maximale $V_{\mathrm{st_i}} \approx 1.1\,\mathrm{V}$ pour un faisceau laser de spot entièrement contenu sur le quadrant associé, ce qui n'est pas adapté à la carte NUCLEO qui n'accepte que des tensions positives comprises entre 0 et 3.3 V. Obtenir une tension adaptée à la carte est l'objet du second montage.

b. Montage inverseur

Le montage inverseur réalisé permet l'obtention en sortie d'une tension de signe opposé à celui de la tension en entrée avec une amplification directement reliée au choix des résistances constituant ce montage. Cet effet peut être décrit par l'équation :

$$V_{\mathrm{s_i}} = -\frac{R_{2_i}}{R_{1_i}} V_{\mathrm{st_i}}\ i \in \{1,2,3,4\}$$

où $V_{\mathrm{s_i}}$, $V_{\mathrm{s_i}}$, R_{2_i} et R_{1_i} sont respectivement les tensions d'entrée et de sortie et les résistances du montage inverseur.

Nous avons choisi les valeurs $R_{1_i} = 1\,\mathrm{k\Omega}$ et $R_{2_i} = 3\,\mathrm{k\Omega},\ i \in \{1,2,3,4\}$ pour amplifier la tension d'entrée en la multipliant par 3. Nous utiliserons également et pour les mêmes raisons un composant TL-084 pour réaliser les 4 fonctions ALI. La tension de sortie obtenue a pour valeur maximale $V_{\mathrm{s_i}} = 3.25\,\mathrm{V}$ et est bien positive, elle est donc adaptée à la carte NUCLÉO.

c. Carte NUCLEO

Afin d'avoir accès aux signaux de la photodiode, nous connectons la carte NUCLEO aux sorties du montage inverseur en reliant ces sorties aux broches PA_0, PA_1, PB_0 et PC_1 (broches acceptant des entrées de type AnalogIn, ce qui est le cas ici).
Pour pouvoir piloter le mouvement des miroirs, nous connectons les broches PA_4 et PA_5 (de type AnalogOut) à l'alimentation des servomoteurs. L'ensemble de ces broches sera utilisé dans le programme d'asservissement réalisé sous Mbed.

III. Partie numérique

1. Programme d'asservissement

Dans cette sous-section, le programme d'asservissement par un PID numérique réalisé sous Mbed est détaillé et expliqué fonction par fonction. Celui-ci est donné en ANNEXE III.1.
La FIGURE III.1 donne l'architecture générale du programme.

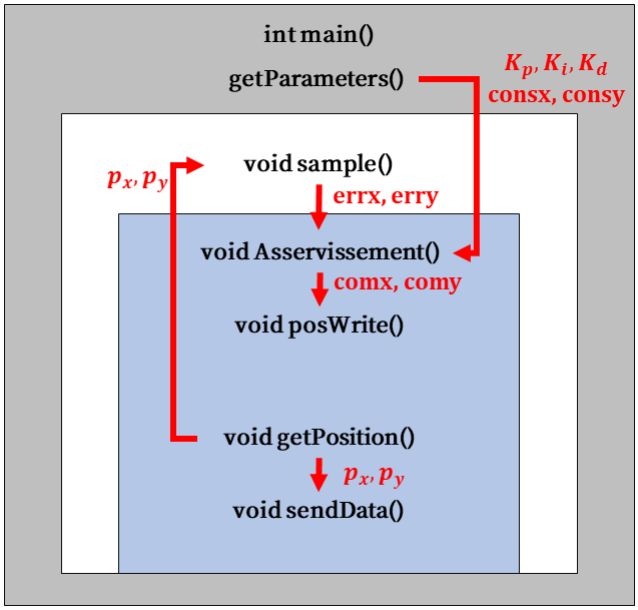


FIGURE III.1 : Schéma de la structure du programme d'asservissement codé sous Mbed

a. Définition des paramètres

Il s'agit de définir les constantes du système ainsi que leur valeur numérique.
- **r** : rayon de la photodiode (en m)
- d : distance entre le dernier miroir et la photodiode (en m)
- K_x, K_y : terme de conversion entre l'angle imposé par les miroirs et la tension pour chaque axe (en rad.V^{-1})
- Te : période d'échantillonnage (en s)

b. Définition des entrées et sorties

Les variables globales d'entrées-sorties, toutes reliées à une broche de la carte NUCLEO, permettent de faire le lien entre les parties analogique et numérique du système. Elles sont définies ci-dessous.

- **AnalogIn measureζ, ($\zeta \in \{A, B, C, D\}$)** : entrée de type AnalogIn, il s'agit de la valeur de la tension associée au signal de l'un des quadrants après un passage dans les montages transimpédance et inverseur.
- **AnalogOut commandX/commandY** : sortie de type AnalogOut, il s'agit du signal de commande de l'alimentation des servomoteurs (un signal par axe en prenant pour référence que le câble du haut de l'alimentation est affecté par commandX). Ce sont ces sorties qui permettent, après application du programme d'asservissement, d'effectivement corriger la position du faisceau en modifiant le mouvement des miroirs.

c. Fonction d'interruption série

La fonction **void getParameters()** permet d'acquérir via MATLAB les valeurs de la consigne pour chaque axe consx et consy ainsi que les valeurs des constantes du correcteur (K_p, K_i et K_d).

d. Fonctions secondaires

- **void getPosition()** : cette fonction lit les signaux provenant de chaque quadrant de la photodiode après leur passage dans les montages transimpédance et inverseur.
  Chaque signal est représenté par une variable avec un nom de lettre (A,B,C et D) et chaque variable provient de la lecture de la tension sur l'une des broches PA_0, PA_1, PB_0 ou PC_1. Une fois ces données acquises, elles sont transférées à MATLAB par la fonction void sendData().

  Les variables locales tempx et tempy traduisent la position du faisceau sur la photodiode sur les axes X et Y respectivement et sont définies pour que les axes de la photodiode soient alignés sur ceux du déplacement.

Pour éviter de calibrer la photodiode et d'établir la conversion tension de sortie ↔ position du faisceau sur la photodiode, une expression normalisée de la position définit tempx et tempy.
Le paramètre r est le rayon de la photodiode et permet de dimensionner la variable.

Pour éviter la propagation des erreurs, les variables de position tempx et tempy ne sont affectées aux valeurs px et py que si ces dernières ne sont pas des NaN. Le seuillage *(A+B+C+D>0.1)* permet de ne pas considérer une position en dessous d'une tension seuil, ce qui permet de limiter l'effet du bruit.

— **void Asservissement()** : cette fonction utilise l'erreur réactualisée par la fonction void sample() aux rangs n et $n-1$ pour chaque axe (via les tableaux errx et erry) ainsi que les valeurs de commande comx et comy et de correction K_p, K_i et K_d acquises via la fonction void getParameters().

L'expression des correcteurs proportionnel, intégral et dérivé numériques est ensuite implémentée et la commande sur chaque axe est réactualisée à la valeur de la somme de chaque terme de correction u_p, u_i et u_d. L'action intégrale possède un terme de protection anti-windup (ou anti-emballement) dont le seuil a été choisi arbitrairement à $V < 1.3$V avec l'expression " *((uix[1] > 0) - (uix[1] <= 0))*fminf(fabs(uix[1]),1.3)*" pour l'axe X.
Les équations de ces correcteurs peuvent donc se récrire sous la forme :

$$u_p(k) = K_p e(k) \text{ pour le proportionnel,}$$
$$u_i(k) = \min(u_i(k-1), u^{\max}) + T_e K_i e(k) \text{ pour l'intégral avec anti-windup,}$$
$$u_d(k) = \frac{K_d}{T_e}(e(k) - e(k-1)) \text{ pour le dérivé}$$

— **void posWrite()** : cette fonction écrit sur les broches PA_4 et PA_5 contrôlant les servomoteurs les commandes comx et comy corrigées obtenues à partir de la fonction void Asservissement().
Ces commandes sont divisées par 3.3 pour avoir un nombre compris entre 0 et 1, les tensions pouvant aller de 0 à 3.25 V et un offset de 0.5 a été rajouté pour faire en sorte que, si le faisceau est à peu près centré sur la photodiode, on ne soit pas limité à un seul quadrant.

— **void sendData()** : cette fonction utilise les variables correspondant à la position du faisceau sur chaque quadrant et actualisées par la fonction void getPosition() pour les transmettre et les afficher sur MATLAB.

e. Fonctions d'interruption et principale

— **void sample()** : cette fonction est la fonction d'interruption du ticker. Une fois les données sur la position et les variables px et py définies par la fonction void getPosition(), le tableau de l'erreur de chaque axe est réactualisé en utilisant l'écart entre la position px/py et la consigne consx/consy puis à partir de cette erreur, la fonction void Asservissement() réactualise la commande comx/comy.
Cette commande est ensuite imposée aux servomoteurs par la fonction void posWrite() et les données de position envoyées à MATLAB via la fonction void sendData() (pour cette dernière fonction, on veillera à choisir une période d'échantillonnage suffisamment faible, donc une fréquence d'échantillonnage suffisamment élevée pour respecter la vitesse de transmission de la liaison RS232).

— **int main()** : cette fonction est la fonction principale. Elle permet la réception des paramètres donnés par la fonction void getParameters() par interruption et d'appliquer la fonction void sample() (donc l'ensemble de l'asservissement) à intervalle régulier avec la période d'échantillonnage Te.
La ligne "*matlab.baud(115200);*" permet de contrôler la vitesse de transmission de l'information de la liaison RS232.

2. Interface graphique

Avec pour objectif de proposer une interface homme-machine simple d'utilisation pour des étudiant.e.s ingénieur.e.s, nous proposons ici une interface graphique réalisée sous MATLAB.

Une esquisse a été initiée sous PYTHON avec la bibliothèque TKINTER mais pour assurer une meilleure coordination avec les essais préexistants réalisés avec la carte NUCLEO, nous avons décidé de basculer sur MATLAB. A noter par ailleurs que la programmation de l'interface sous MATLAB est plus intuitive de prime abord que sous PYTHON. Dans tous les cas, de nombreuses sources sont accessibles en ligne en libre accès.
Pour l'essentiel, nous nous sommes appuyés sur deux tutoriels dont les liens respectifs sont accessibles en cliquant sur le texte en gras associé

— **petit tutoriel** pour comprendre les grandes lignes de fonctionnement de l'interface sous MATLAB, facilement adaptable à notre problématique.

— **tutoriel très complet** quant à l'utilisation de TKINTER pour réaliser une interface graphique sous PYTHON, un peu trop dense vis-à-vis des ambitions du projet. Le tutoriel n'explique pas par ailleurs la connexion avec ARDUINO.

a. Création de l'interface

Par souci de simplicité, nous avons utilisé le module GUIDE déjà présent sur MATLAB même si celui-ci est semble-t-il voué à être remplacé par APPDESIGNER.

Pour lancer GUIDE, il suffit d'écrire dans la Command Windows l'instruction guide. La fenêtre de la FIGURE III.2.a.1 devrait alors s'ouvrir.

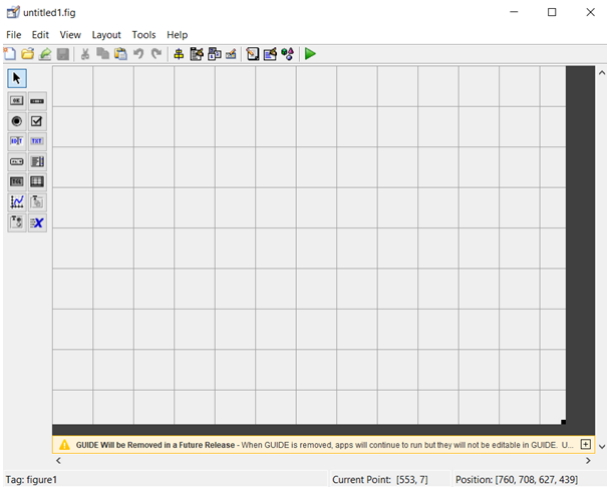


FIGURE III.2.a.1 : Fenêtre du module GUIDE de MATLAB

Il est possible de régler la taille de l'interface en faisant glisser le coin droit de celle-ci. Nous pouvons ensuite insérer un certain nombre d'icônes associés à différentes fonctions : graphe, bouton, etc. En vue des attendus de notre interface, nous avons utilisé les icônes : **Push Button** pour lancer la simulation, **Edit Text** pour entrer les valeurs utilisateur, **Static Text** pour indiquer à l'utilisateur les entrées à sélectionner et **Axes** pour les tensions des quadrants et la position du laser.

Après avoir positionné les différentes icônes à notre convenance, nous obtenons le début d'interface de la FIGURE III.2.a.2.

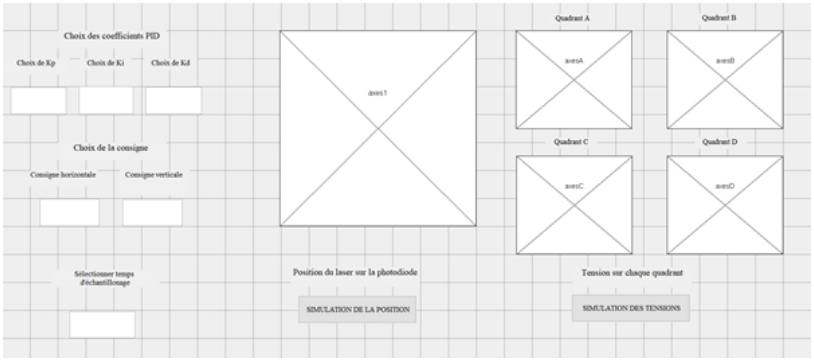


FIGURE III.2.a.2 : Positionnement des icônes sur l'interface

Après l'avoir enregistré, une série de code préremplie devrait apparaître sur MATLAB. En sélectionnant sur RUN sur la page de l'interface, son allure fonctionnelle se génère.

b. Code de l'interface

Tout d'abord, revenons un instant sur le menu de l'interface où figurent les icônes sélectionnées. Pour pouvoir programmer leur fonction, il faut en les sélectionnant par un clic droit venir changer leur **tag** et y indiquer un nom que nous pourrons retrouver sur les lignes de code générées automatiquement.
Par exemple, pour le bouton SIMULATION nous avons inscrit le nom "simulate". Il n'est pas nécessaire de le faire pour les icônes **Static Text** car elles ne servent qu'à indiquer les valeurs à entrer par l'utilisateur.
Ceci fait, nous pouvons commencer à travailler sur le code.

La plupart des lignes de code (en anglais) obtenues automatiquement sont indiquées en commentaire ou ne sont pas à toucher par l'utilisateur. Il suffira de modifier à sa guise les fonctions issues des différentes icônes.
Après avoir complété le code (c.f. les commentaires) il suffit de sélectionner **Run**. S'il s'agit de récupérer celui d'un tiers, il faudra également téléverser les fichiers .m, .fig et .asv.

c. Description de l'interface et fonctionnalités

La dernière version de l'interface est illustrée par la FIGURE III.2.c. Dans cette version, il est possible de choisir la valeur des coefficients du correcteur K_p, K_i et K_d ainsi que les consignes et la durée de l'échantillonnage souhaitées.

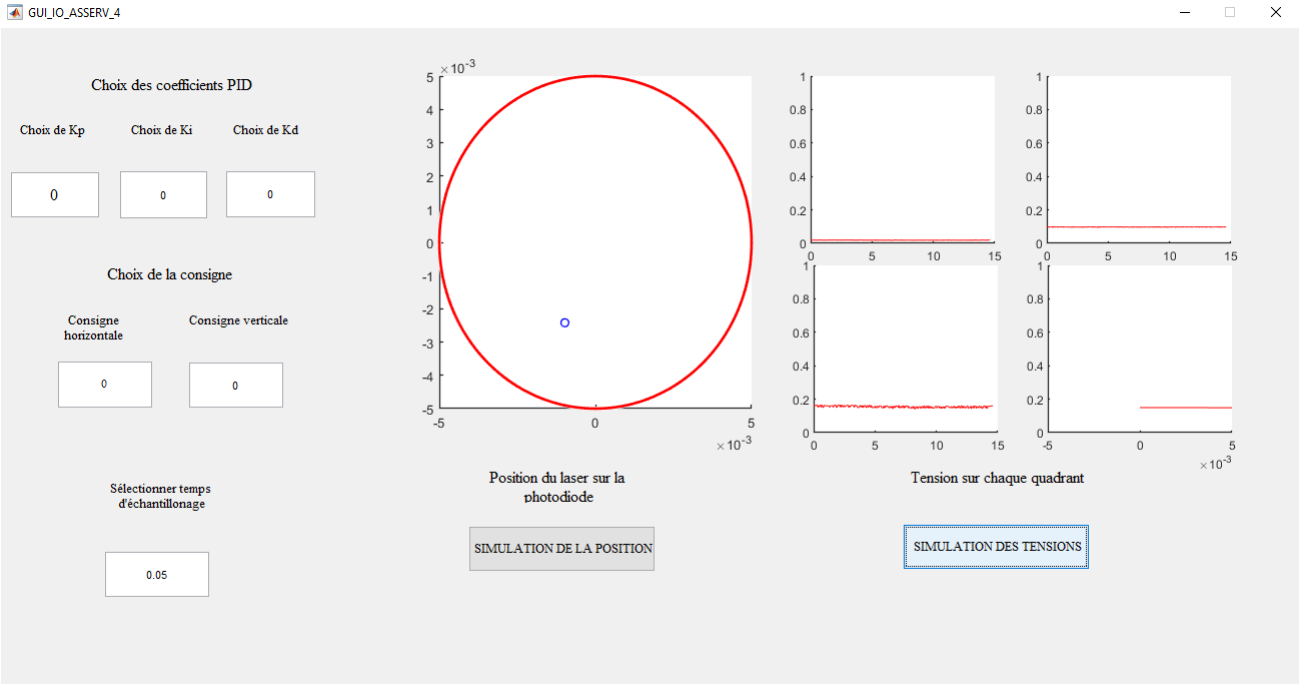


FIGURE III.2.c : Affichage de l'interface graphique pour une photodiode en l'absence totale d'éclairement laser

La position sur la photodiode, délimitée par un cercle rouge, est indiquée par le point cerclé de bleu et est accessible en cliquant sur le bouton SIMULATION DE LA POSITION, cette position s'actualise quasiment en temps réel.
Les tensions associées à chaque quadrant sont également accessibles via le bouton SIMULATION DES TENSIONS et s'affichent chacune sur un graphe avec une évolution quasiment en temps réel et suivant bien les variations d'éclairement de la photodiode.

Toutefois, l'interface n'est pas totalement satisfaisante car :

- le graphe de position de la photodiode devrait indiquer une cinquantaine de points et non un seul
- même si les 4 tensions sont récupérées et lues par MATLAB, l'un des graphes n'affiche pas les valeurs souhaitées (soit une ligne horizontale, soit aucune courbe), ce qui est visible sur le graphe inférieur droit de la FIGURE III.2.c.

IV. Bilan du projet

1. Déroulement du projet

a. Retroplanning

Le tableau de cette sous-section permet de préciser les actions menées pour réussir le projet.
Par souci de simplification, les membres de l'équipe ayant travaillé sur chaque tâche sont mentionnés par leurs initiales, c'est-à-dire **EB** pour Eddy Barlogis, **MHC** pour Marie-Hélène Carron, **MH** pour Maël Hubert et **MM** pour Mohamed Meguebel.

	Partie analogique	**Partie numérique**
Séance 0	Choix des composants optiques (**MHC, MM**) Prise en main du montage (**EB, MH**)	×
Séance 1	Premiers montages de photodétection avec 2 photodiodes (**MHC, MM**)	Première version du programme d'asservissement sous Mbed(**MH**)
Séance 2	Mise en place du montage transimpédance avec la photodiode 4-quadrants (**EB, MM**)	Amélioration du code Mbed (**MH**) Travail sur l'interface graphique (**MHC**)
Séance 3	Amélioration du montage électronique, caractérisation des miroirs (angle/tension) et correction de la saturation (**EB, MM**)	Tests du code Mbed (**MH**) Travail sur l'interface graphique et suivi de formations à l'intefaçage (**MHC, MM**)
Séance 4	Réalisation du montage inverseur (**EB, MHC**)	Amélioration du code Mbed et (**MH**) visualisation des positions sous Matlab Travail sur l'interface graphique (**MM**)
Séance 5	×	Tests du code Mbed (**MH, MM, EB, MHC**) Travail sur l'interface graphique (**MM**)
Séance 6	×	Tests via l'interface(**MH, MM, MHC**) Amélioration de l'interface (**MM, MH**)

b. Remarques, problèmes rencontrés

Lors de la première séance de la phase 3 (Séance 1), nous avons câblé un montage de photodétection composé de deux photodiodes, deux ALI et deux résistances afin d'effectuer les premiers tests du programme d'asservissement. Cette solution était provisoire car nous n'avions alors plus accès à une photodiode quatre-quadrants.

Lors de la Séance 3, puisque Matlab affichait des valeurs de position inadéquates (à savoir une succession de caractères plus ou moins aléatoires), la question de la possible saturation de la photodiode a été soulevée et corrigée. Cependant, les valeurs de position étaient toujours inadéquates car les tensions délivrées par le montage transimpédance sont négatives, ce que la carte Nucleo n'a pas su gérer, c'est pourquoi un montage inverseur a été mis en place au cours de la Séance 4 ainsi que lors de sessions de travail supplémentaires.

L'interface graphique, d'abord pensée sur Matlab a été développée sur Python à la suite d'une redistribution des tâches. Compte-tenu de la difficulté de réaliser certaines fonctionnalités sous Python, l'interface finale a été développée sous Matlab.

Malgré de nombreuses améliorations du programme d'asservissement, seul l'axe Y semble correctement asservi, l'axe X étant oscillant.

2. Validation du cahier des charges

a. Respect des contraintes matérielles

Notre prototype **tient compte de toutes les contraintes matérielles** :

- l'asservissement et la correction sont réalisés via un PID numérique programmé sous Mbed
- les servomoteurs utilisés sont des servomoteurs classiques et c'est l'alimentation de ces derniers qui permet la correction
- les coefficients du correcteur peuvent être modifiés en temps réel grâce à la liaison série avec MATLAB et via l'inferface graphique

b. Respect des exigences

Notre prototype **ne valide pas toutes les exigences du cahier des charges** :

- **Rapidité et Précision** : **non validées** lorsqu'on génère une perturbation, et ce, quel que soit le choix des paramètres du correcteur, seul l'axe Y se stabilise.
  La visualisation de la position du faisceau nous montre en effet qu'une oscillation est présente selon l'axe X. Il est possible qu'une erreur de gain angle/tension pour cet axe soit responsable de cette oscillation, néanmoins, par manque de temps, il n'a pas été possible de déterminer la cause du problème et encore moins de corriger cet effet.

  A cause de cette oscillation, la précision de notre dispositif ne peut être déterminée, de même que la rapidité ne peut être déterminée puisque l'asservissement n'est que partiel.
- **Ergonomie** : **partiellement validée** l'interface graphique développée permet un contrôle simplifié de l'asservissement du faisceau quasiment en temps réel, bien que l'affichage de la position et de l'une des tensions ne soit pas optimal

3. Enrichissement personnel, remarques sur le projet

Eddy :

Ce projet a été très intéressant car il nous a permis d'utiliser nos connaissances et compétences dans différents domaines (automatique, programmation, électronique) afin d'arriver à un but concret. Travailler en équipe sur un projet plus ambitieux que d'habitude était une bonne expérience de découverte de ce qui pourra nous attendre en entreprise. En effet nous avons pu apprendre à répartir des tâches afin de respecter le calendrier, améliorer notre communication au sein de l'équipe, et développer notre autonomie afin de résoudre les divers problèmes que nous avons pu rencontrer.

Marie-Hélène :

Connaissant mon profil très voire trop théoricien, ce projet a été l'occasion de me confronter à des problèmes concrets et souvent imprévus. Par exemple, les problèmes liés au développement de l'interface graphique ou la difficulté de câbler des montages inverseurs sur une plaque comportant une grande quantité de composants sans faire de court-circuit m'ont sortie de ma routine et cela m'a fait progresser.

Par ailleurs, c'est en comprenant la logique du code d'asservissement que j'ai à la fois compris la logique de l'asservissement numérique et plus généralement les grands fondamentaux de la programmation sur Mbed.

Je dirais donc qu'au-delà d'avoir passé des moments enrichissants auprès des autres membres de l'équipe (que je connaissais déjà bien), j'ai amélioré légèrement mes compétences en électronique et largement ma compréhension de la programmation d'un asservissement numérique.

Maël :

Même si j'ai touché un peu à tout, je me suis surtout intéressé au code et même si je n'ai pas appris tant que ça dans ce domaine, c'était intéressant de mettre en pratique mes connaissances dans un contexte plus général.

En revanche, c'était un projet ambitieux et il nous a manqué du temps pour le mener totalement à bien. C'est un peu dommage. Nous étions vraiment la tête dans le guidon et manquions parfois du recul nécessaire. Avec ce recul, certaines choses nous paraissent évidentes, ce qui n'était pas le cas quand nous travaillions dessus.

Mohamed :

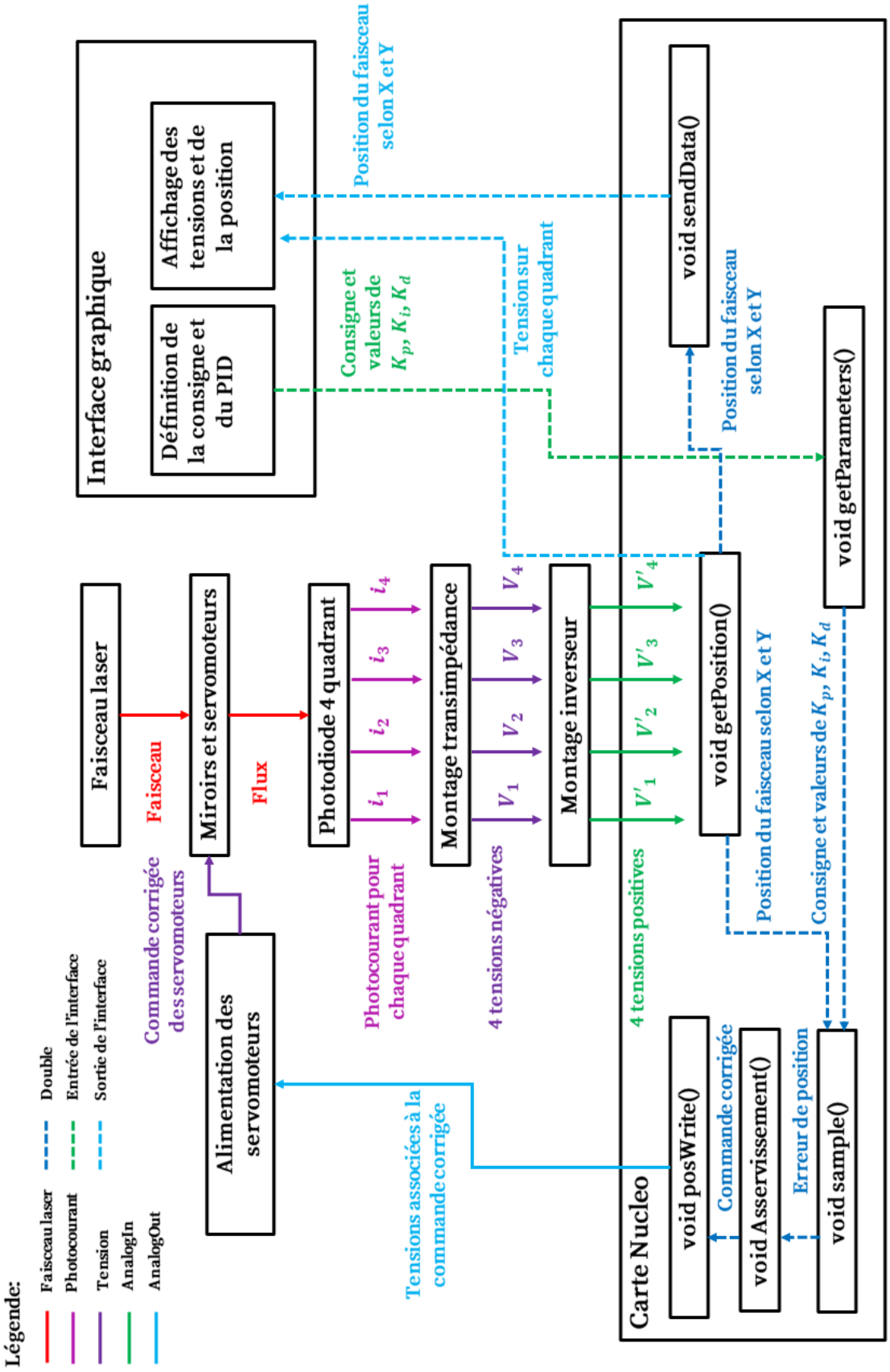
Le choix du sujet découle essentiellement d'un désir d'allier les savoir-faire acquis en électronique mais également en optique. Le début du projet fut ambitieux en vertu du fait que l'équipe s'est lancée dans la conception du système d'asservissement avec deux axes et par utilisation de la photodiode quatre-quadrants. Cela ne nous a néanmoins pas découragé et nous avons vite su nous positionner les au sein du groupe en fonction de nos compétences individuelles.

Me considérant plutôt comme équilibré avec une appétence plus forte pour la théorie, je me suis proposé en tant que relai entre les différents sous-groupes. J'ai donc beaucoup discuté à la fois avec le pôle électronique pour la conception du système de détection (montage amplificateur, prise de l'information, etc.) mais également avec le pôle programmation pour l'asservissement PID (détermination des constantes de conversion, communication carte-MATLAB, etc.).
Il est clair que ceci m'a permis de développer mes compétences dans chacun des domaines mais aussi et surtout d'y voir leur caractère transverse et l'importance de savoir basculer de l'un à l'autre. Ces premières semaines de travail ne furent cependant pas embûches, accompagnées par exemple de bugs inexpliqués dans le code ou encore de circuits électroniques fastidieux à réaliser.

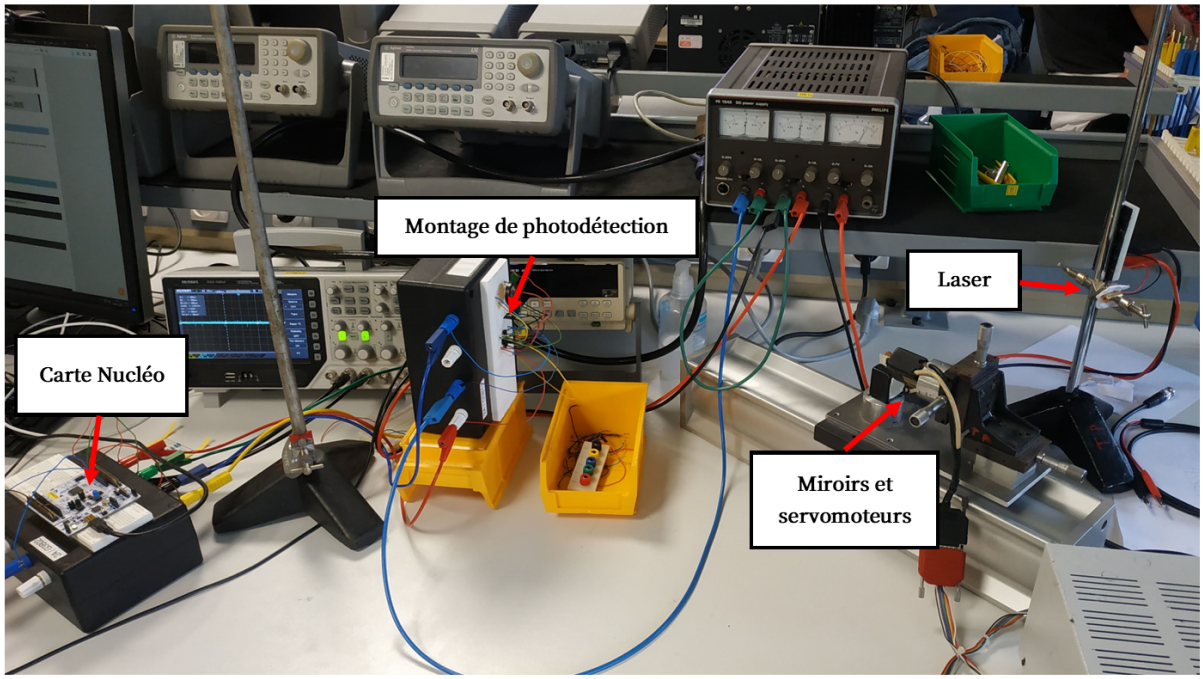
En outre, suite à un changement dans la répartition du projet, j'ai été amené à me concentrer sur une partie qui m'était sur le moment inconnue : la réalisation d'une interface. Plus familier avec PYTHON, j'ai pris la décision de m'y essayer à l'aide de la bibliothèque TKINTER. D'abord enthousiaste, je me suis vite heurté à certaines difficultés, notamment la prise d'information continue avec la carte NUCLEO. En ayant ensuite discuté plus longuement avec le pôle programmation, j'ai repris du début sous MATLAB cette fois. Plus concluant cette fois, l'interface est fonctionnelle pour l'essentiel. Il reste que le rendu n'est pas aussi satisfaisant

Je peux donc conclure avec le fait que ce PROTIS m'a apporté d'importants enseignements techniques - en électronique et en informatique - mais également managériales et d'organisation. En effet, j'ai appris à accepter l'arrivée des problématiques et difficultés sous ses différents plans.

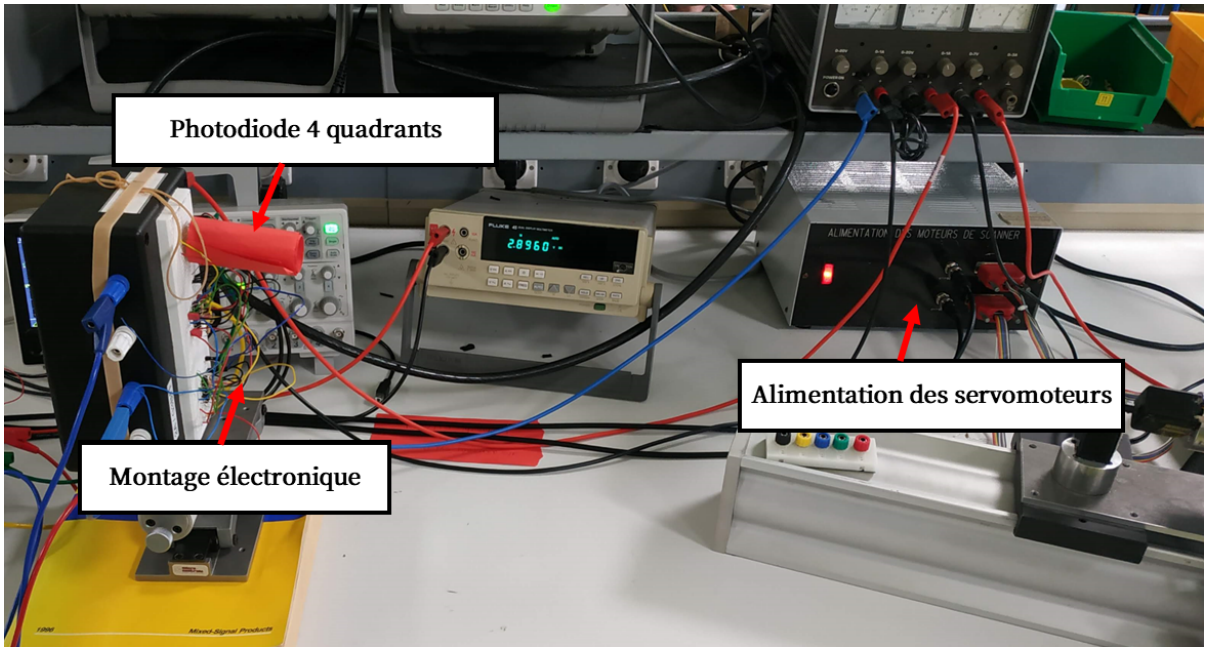
Annexe I.1 : Schéma fonctionnel du protoype développé



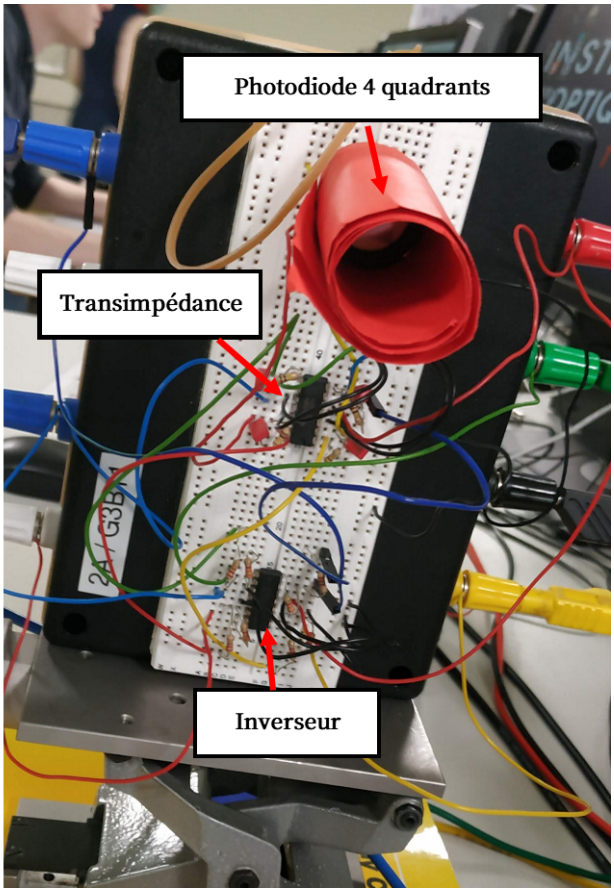
Annexe II.1 : Photographies du montage réel



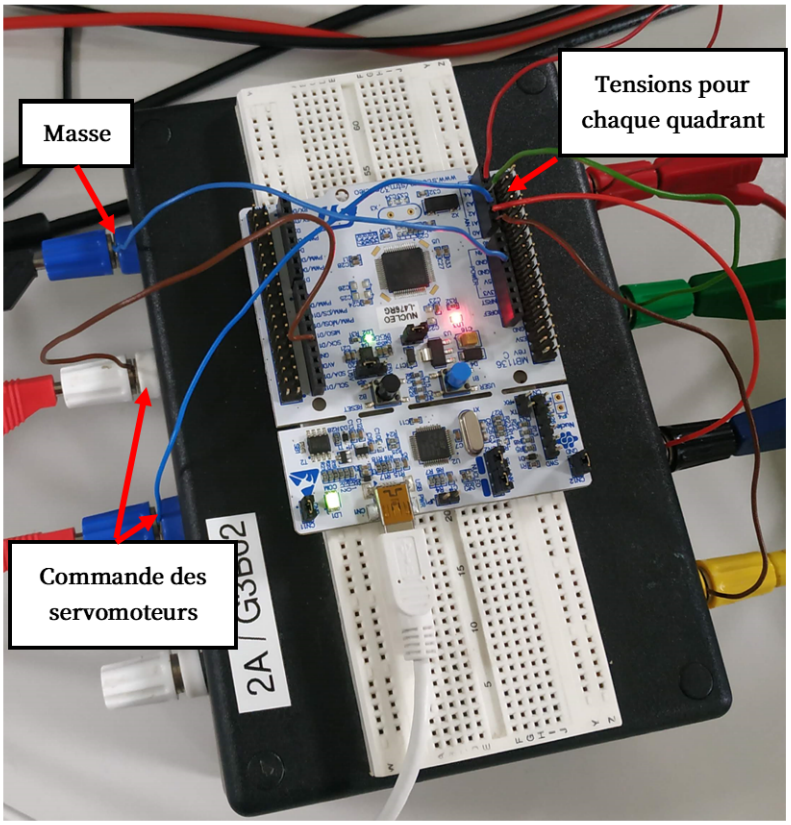
Annexe II.1.a : Protoype complet, première illustration



Annexe II.1.b : Protoype complet, deuxième illustration

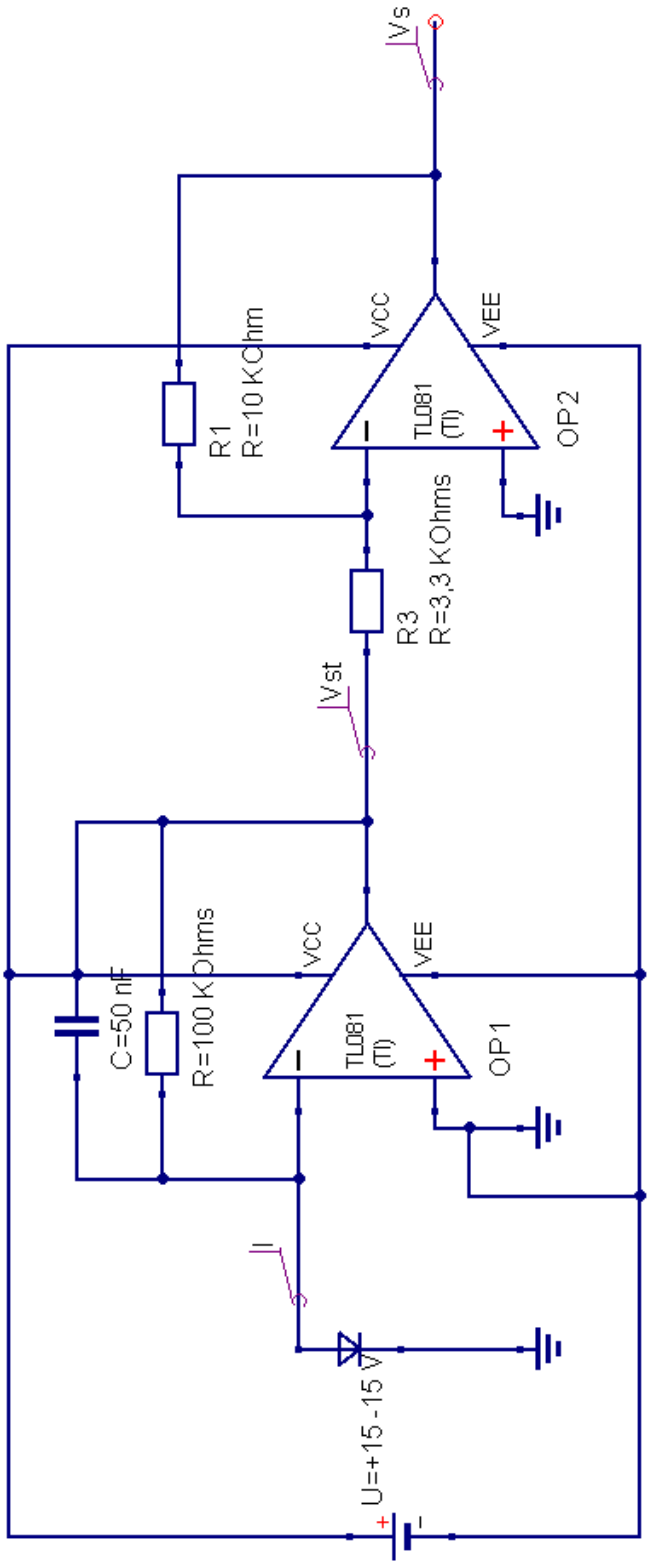


Annexe II.1.c : Montage de photodétection



Annexe II.1.d : Carte Nucleo

Annexe II.2 : Schéma du montage électronique pour un photocourant réalisé sur Qucs



Annexe III.1 : Programme d'asservissement sous Mbed

```
/* Déclaration des ressources externes */
#include "mbed.h"


/* Déclaration des constantes */
#define r 0.005 // Rayon photodiode (m)
#define d 0.45 // Distance caractéristique (m)
#define Kx 29.47 // Conversion miroirs angle -> tension (rad/V)
#define Ky 42.12
#define Te 0.05 // Période d'échantillonage (s)


/* Déclaration des entrées/sorties */
AnalogIn measureA(PA_0); // Signaux photodiode
AnalogIn measureB(PA_1);
AnalogIn measureC(PB_0);
AnalogIn measureD(PC_1);
AnalogOut commandX(PA_4); // Commandes des miroirs, X = cable haut
AnalogOut commandY(PA_5);

Serial matlab(USBTX, USBRX);
Ticker sampling;


/* Déclaration de fonctions */
void sample(void);

void getParameters(void);

void getPosition(void);

void Asservissement(void);

void posWrite(void);

void sendData(void);


/* Déclaration des variables globales */
double errx[2] = {0,0}; // Erreurs à k et k-1 (V)
double erry[2] = {0,0};

double px = 0; // Positions (m)
double py = 0;

double consx = 0; // Consignes (m)
double consy = 0;

double comx = 0; // Commandes (V)
double comy = 0;

double Kp = 1; // Paramètres d'asservissement
double Ki = 1;
double Kd = 0;

double uix[2] = {0,0}; // Commandes intégrales k et k-1 (V)
double uiy[2] = {0,0};

double A,B,C,D; // Tensions des quadrants

int bool_graph = false; // Tensions ABCD (true) ou position (px,py) (false)
```

Annexe III.1.1.a : Programme d'asservissement réalisé sous Mbed, paramètres et variables

```
/* Fonction principale */
int main(){
    matlab.baud(115200);
    matlab.attach(&getParameters); // Réception par interruption (pas scrutation!)
    sampling.attach(&sample,Te); // Action à intervalle Te régulier

    while (1) {

    }
}


/* Fonction d'interruption du ticker */
void sample() {

    getPosition();

    errx[1] = errx[0];
    erry[1] = erry[0];
    errx[0] = d*Kx*(px - consx); // A tester
    erry[0] = d*Ky*(py - consy);

    Asservissement(); // Kp,Ki,Kd,errx,erry,
    posWrite();

    sendData(); // Attention avec Te trop bas!
}


/* Fonction d'interruption série */
void getParameters() {

    matlab.scanf("%lf,%lf,%lf,%lf,%lf,%d",&Kp,&Ki,&Kd,&consx,&consy,&bool_graph); //
Pas besoin de vérifier matlab.readable()
}
```

ANNEXE III.1.1.b : Programme d'asservissement réalisé sous Mbed, fonctions principale et d'interruption

```
/* Fonctions secondaires */


void getPosition() {
    double A,B,C,D,tempx,tempy;

    A = measureA.read();
    B = measureB.read();
    C = measureC.read();
    D = measureD.read();

    tempx = r*((A+D)-(C+B))/(A+B+C+D);  // Axes photodiode alignés avec axes de
déplacement
    tempy = r*((A+B)-(C+D))/(A+B+C+D);  // Normalisation par le signal global

    if ((!isnan(tempx))&&(!isnan(tempy))&&(A+B+C+D>0.1)) { // Ignorer les NaNs +
Seuillage position
        px = tempx;
        py = tempy;
    }
}


void Asservissement() {
    double upx, upy, udx, udy;

    upx = Kp*errx[0]; // Action proportionelle
    upy = Kp*erry[0];

    uix[1] = uix[0]; // Action intégrale + anti-windup à arbitrairement 1.3V < 1.65V
    uiy[1] = uiy[0];
    uix[0] = ((uix[1] > 0) - (uix[1] <= 0))*fminf(fabs(uix[1]),1.3) + Te*Ki*errx[0];
    uiy[0] = ((uiy[1] > 0) - (uiy[1] <= 0))*fminf(fabs(uiy[1]),1.3) + Te*Ki*erry[0];

    udx = (Kd/Te)*(errx[0] - errx[1]); // Action dérivée
    udy = (Kd/Te)*(erry[0] - erry[1]);

    comx = upx + uix[0] + udx;
    comy = upy + uiy[0] + udy;
}


void posWrite() {

    commandX.write(0.5 + comx/3.3); // [0,1] <- [-1.65,1.65]
    commandY.write(0.5 + comy/3.3);
}


void sendData() {

    if (matlab.writeable()) {
        if (bool_graph == 1) {
            matlab.printf("%lf,%lf,%lf,%lf\r\n",A,B,C,D);
        } else {
            matlab.printf("%lf,%lf\r\n",px,py);
        }
    }
}
```

ANNEXE III.1.1.c : Programme d'asservissement réalisé sous Mbed, fonctions secondaires

Annexe III.2 : Code de l'interface sur Matlab commenté

```
function varargout = GUI_IO_ASSERV_4(varargin)
% GUI_IO_ASSERV_4 MATLAB code for GUI_IO_ASSERV_4.fig
%      GUI_IO_ASSERV_4, by itself, creates a new GUI_IO_ASSERV_4 or raises the existing
%      singleton*.
%
%      H = GUI_IO_ASSERV_4 returns the handle to a new GUI_IO_ASSERV_4 or the handle to
%      the existing singleton*.
%
%      GUI_IO_ASSERV_4('CALLBACK',hObject,eventData,handles,...) calls the local
%      function named CALLBACK in GUI_IO_ASSERV_4.M with the given input arguments.
%
%      GUI_IO_ASSERV_4('Property','Value',...) creates a new GUI_IO_ASSERV_4 or raises
the
%      existing singleton*.  Starting from the left, property value pairs are
%      applied to the GUI before GUI_IO_ASSERV_4_OpeningFcn gets called.  An
%      unrecognized property name or invalid value makes property application
%      stop.  All inputs are passed to GUI_IO_ASSERV_4_OpeningFcn via varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help GUI_IO_ASSERV_4

% Last Modified by GUIDE v2.5 06-Apr-2022 17:21:44

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @GUI_IO_ASSERV_4_OpeningFcn, ...
                   'gui_OutputFcn',  @GUI_IO_ASSERV_4_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before GUI_IO_ASSERV_4 is made visible.
function GUI_IO_ASSERV_4_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to GUI_IO_ASSERV_4 (see VARARGIN)

% Choose default command line output for GUI_IO_ASSERV_4
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes GUI_IO_ASSERV_4 wait for user response (see UIRESUME)
% uiwait(handles.figure1);

% /!\ A ETE AJOUTE /!\

global nucleo % prend la variable "nucleo" comme "global" à utiliser le long du code
```

```
nucleo = serialport('COM8', 115200); % il faudra choisir le bon port. Le deuxième
paramètre correspond au BaudRate usuel



% --- Outputs from this function are returned to the command line.
function varargout = GUI_IO_ASSERV_4_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;



function choix_Kp_Callback(hObject, eventdata, handles)
% hObject    handle to choix_Kp (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of choix_Kp as text
%        str2double(get(hObject,'String')) returns contents of choix_Kp as a double



% /!\ A ETE AJOUTE /!\

handles.dataKp = get(hObject,'String'); % récupère au format String la donnée
utilisateur
handles.KpSamples = str2double(handles.dataKp); % convertit la donnée utilisateur au
format Double à renvoyer sur la carte Nucleo
guidata(hObject, handles);

% même chose pour les autres handles.data de la suite du code


% --- Executes during object creation, after setting all properties.
function choix_Kp_CreateFcn(hObject, eventdata, handles)
% hObject    handle to choix_Kp (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function choix_Ki_Callback(hObject, eventdata, handles)
% hObject    handle to choix_Ki (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of choix_Ki as text
%        str2double(get(hObject,'String')) returns contents of choix_Ki as a double

% % /!\ A ETE AJOUTE /!\

handles.dataKi = get(hObject,'String');
handles.KiSamples = str2double(handles.dataKi);
guidata(hObject, handles);
```

```
% --- Executes during object creation, after setting all properties.
function choix_Ki_CreateFcn(hObject, eventdata, handles)
% hObject    handle to choix_Ki (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function choix_Kd_Callback(hObject, eventdata, handles)
% hObject    handle to choix_Kd (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of choix_Kd as text
%        str2double(get(hObject,'String')) returns contents of choix_Kd as a double

% % /!\ A ETE AJOUTE /!\

handles.dataKd = get(hObject,'String');
handles.KdSamples = str2double(handles.dataKd);
guidata(hObject, handles);


% --- Executes during object creation, after setting all properties.
function choix_Kd_CreateFcn(hObject, eventdata, handles)
% hObject    handle to choix_Kd (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in simulate_position.
function simulate_position_Callback(hObject, eventdata, handles)
% hObject    handle to simulate_position (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% /!\ A ETE AJOUTE /!\

% Pour renvoyer sur la carte Nucléo

str =
sprintf("%f,%f,%f,%f,%f,0",handles.KpSamples,handles.KiSamples,handles.KdSamples,handle
s.cxSamples,handles.cySamples);
% renvoie les donnes utilisateurs sur la carte Nucleo. Le 0 constitue un
% booléen qui différencie entre les deux types de simulation
global nucleo % il faut remettre "nucleo" en variable globale dans la fonction du
bouton
writeline(nucleo,str) % écrit sur la carte Nucleo les valeurs utilisateur, ensuite
séparées par la carte.

% Pour la position du laser
```

```
cla(handles.axes1) % permet de mettre le tracé sur le graphe dont le tag est
(changeable sur la figure) axes1
viscircles(handles.axes1,[0 0],0.005) % trace le cercle sur lequel on représente la
position du laser
ph = animatedline(handles.axes1,'Marker','o','Color','b'); % permet d'animer la
position du laser en temps réel
hold on
xlim([-0.005 0.005])
ylim([-0.005 0.005])
k = 0;
while true % boucle infinie
    data = readline(nucleo); % lit la tension, donc la position du laser sur la
photodiode
    if mod(k,50) == 0
        clearpoints(ph) % pour éviter d'avoir un grand nombre de lignes qui décrivaient
la position du laser
    end
    try
        data = double(split(data,","))'; % permet de séparer les valeurs de positions
en x et y fournit par la carte Nucleo
        clearpoints(ph)
        addpoints(ph,data(1),data(2));
        drawnow limitrate;
        k = k+1;
    end
end


function consx_Callback(hObject, eventdata, handles)
% hObject    handle to consx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of consx as text
%        str2double(get(hObject,'String')) returns contents of consx as a double

% % /!\ A ETE AJOUTE /!\

handles.datacx = get(hObject,'String');
handles.cxSamples = str2double(handles.datacx);
guidata(hObject, handles);


% --- Executes during object creation, after setting all properties.
function consx_CreateFcn(hObject, eventdata, handles)
% hObject    handle to consx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function consy_Callback(hObject, eventdata, handles)
% hObject    handle to consy (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of consy as text
%        str2double(get(hObject,'String')) returns contents of consy as a double

% /!\ A ETE AJOUTE /!\
```

```
handles.datacy = get(hObject,'String');
handles.cySamples = str2double(handles.datacy);
guidata(hObject, handles);


% --- Executes during object creation, after setting all properties.
function consy_CreateFcn(hObject, eventdata, handles)
% hObject    handle to consy (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function Te_Callback(hObject, eventdata, handles)
% hObject    handle to Te (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of Te as text
%        str2double(get(hObject,'String')) returns contents of Te as a double

% /!\ A ETE AJOUTE /!\

handles.dataTe = get(hObject,'String');
handles.TeSamples = str2double(handles.dataTe);
guidata(hObject, handles);


% --- Executes during object creation, after setting all properties.
function Te_CreateFcn(hObject, eventdata, handles)
% hObject    handle to Te (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in simulate_tensions.
function simulate_tensions_Callback(hObject, eventdata, handles)
% hObject    handle to simulate_tensions (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Pour renvoyer sur la carte Nucléo

str =
sprintf("%f,%f,%f,%f,%f,1",handles.KpSamples,handles.KiSamples,handles.KdSamples,handle
s.cxSamples,handles.cySamples);
global nucleo
writeline(nucleo,str)
% renvoie les donnes utilisateurs sur la carte Nucleo. Le 1 constitue un
% booléen qui différencie entre les deux types de simulation

% Pour les graphes des tensions
```

```
cla(handles.axesA) % permet de mettre le tracé sur le graphe dont le tag est
(changeable sur la figure) axesA
cla(handles.axesB)
cla(handles.axesC)
cla(handles.axesD)
A = animatedline(handles.axesA,'Color','r'); % pour l'animation des tensions, les avoir
en temps réel
ylim(handles.axesA,[0 1]) % le premier paramètre est pour mettre tracé sur le graphe
dont le tag est (changeable sur la figure) axesA, le second correspond au fait qu'on
prend les tensions normalisées
B = animatedline(handles.axesB,'Color','r');
ylim(handles.axesB,[0 1])
D = animatedline(handles.axesC,'Color','r');
ylim(handles.axesC,[0 1])
C = animatedline(handles.axesD,'Color','r');
ylim(handles.axesD,[0 1])

k = 0;
while true
    data = readline(nucleo);
    try
        data = double(split(data,","))';
        addpoints(A,k*handles.TeSamples,data(1));
        addpoints(B,k*handles.TeSamples,data(2));
        addpoints(C,k*handles.TeSamples,data(3));
        addpoints(D,k*handles.TeSamples,data(4));
        drawnow limitrate;
    end
    k = k+1;
end
```