Guide complet

Mise à jour du hardware et software de deux spectrofluorimètres

Auteur :
Youssef BOUSSALEM

Rôle :
Stagiaire, étudiant en 4ème année à l'INSA Strasbourg

Encadrant :
Lorry Engel

Rôle :
Ingénieur en électronique

IPCMS - Institut de Physique et Chimie des Matériaux de Strasbourg

1er août 2025

Table des matières

Chapitre 1

Documentation du programme de contrôle des spectrofluorimètres

Le fonctionnement du spectrofluorimètre repose sur une interaction continue entre un code Arduino et un programme Python. Le code Arduino tourne sur une carte Uno et assure le contrôle bas niveau des moteurs pas à pas via un Shield CNC. De son côté, le script Python s'exécute sur une Raspberry Pi et joue un rôle central de coordination : c'est lui qui envoie les ordres, récupère les données, et orchestre toute la séquence de mesure.

La communication entre les deux se fait par port série, avec un protocole simple. Le Python envoie des commandes sous forme de texte (par exemple pour déplacer un moteur ou initialiser une fente), et l'Arduino répond une fois l'opération terminée.

Le fichier main.py lit un fichier de configuration YAML qui décrit les mesures à réaliser avec leurs longueurs d'onde à scanner, les temps d'intégration et autre. Ensuite, il initialise les composants nécessaires (monochromateurs, détecteurs, fentes) et lance une boucle dans laquelle il déplace les moteurs, effectue les mesures, puis enregistre les résultats.

Le code Arduino (main.ino) lit les commandes série, les découpe, puis agit selon la demande. Il gère les mouvements des moteurs, la lecture des capteurs de position, et renvoie un accusé de réception à Python une fois l'action réalisée. C'est un système synchrone.

L'ensemble fonctionne comme un système maître-esclave : Python donne les ordres, Arduino exécute. C'est cette séparation des responsabilités qui rend l'ensemble modulaire et adaptable à différents types de spectrofluorimètres.

1.1 Programme Arduino

Setup

La fonction *setup()* initialise le système. Elle configure les broches des moteurs et des capteurs de fin de course, et établit la communication série à 115200 bauds avec le Raspberry Pi.

Elle appelle d'abord la fonction *initMotorPins()* pour préparer les broches *direction*, *step* et *enable* des moteurs, puis désactive les moteurs (enable = HIGH). Ensuite, elle configure les pins des capteurs comme entrées (INPUT_PULLUP), de plus configure aussi la pin pour le shutter.

Loop

La fonction loop() est une boucle infinie qui surveille en permanence s'il y a des données arrivant par le port série. Si une ligne de commande est reçue (terminée par un retour à la ligne), elle est stockée dans *line* et analysée.

Elle est alors transmise à la fonction *handleCommand()* qui va interpréter l'ordre, exécuter la commande et renvoyer un accusé de réception (Serial.println()).

findZero

Cette fonction est responsable de la mise à zéro d'un moteur. Elle sert à aligner mécaniquement le moteur avec une position de référence fixée par un capteur.

Lorsqu'elle est appelée, la fonction active la direction définie dans *motor.zeroDirection*, puis lance le moteur pas à pas tant que le capteur n'est pas encore déclenché. L'état du capteur est vérifié avec un *digitalRead()* sur la pin *motor.limitSwitchPin*. Le moteur continue de tourner tant que le capteur ne retourne pas la valeur attendue, définie dans *motor.zeroPinState*.

Un timeout de sécurité est mis en place : si la position zéro n'est pas atteinte après un certain temps (défini dans *ZERO_ TIMEOUT_ MS*), la fonction s'interrompt et écrit *ZERO,TIMEOUT* sur le port série. Cela évite de bloquer le moteur indéfiniment en cas de capteur défectueux ou mal câblé. Sinon, si tout se passe bien, elle affiche *ZERO,DONE*.

moveStepper

Cette fonction permet de faire tourner un moteur d'un nombre de pas donné, dans une direction précise, à une vitesse choisie (rapide ou lente).

La direction est d'abord envoyée à la pin *motor.dirPin* via *digitalWrite()*. Puis, une boucle for est lancée sur le nombre de pas souhaité. À chaque itération, un front montant puis descendant est appliqué sur la pin *motor.stepPin*, avec un *delayMicroseconds()* pour espacer les impulsions. Le temps d'attente est déterminé par la vitesse sélectionnée : *motor.slow_ stepSpeed* ou *motor.fast_ stepSpeed*.

La fonction ne vérifie pas l'état du capteur de fin de course durant le mouvement — elle s'exécute de façon purement aveugle. Ce comportement est volontaire : la sécurité repose sur le bon étalonnage initial et l'usage de limites logicielles côté Python. Une fois le déplacement terminé, elle écrit *MOVE,DONE* sur le port série pour informer Python que l'action est complétée.

handleCommand

Cette fonction est au centre de la logique de la boucle loop(). Cette fonction analyse les commandes reçues via la liaison série, identifie l'action à exécuter, et appelle la fonction correspondante (*moveStepper()* ou *findZero()*).

La commande est découpée avec la méthode *command.substring()* pour extraire les champs séparés par des virgules. Le premier champ indique l'action à effectuer (*MOVE, ZERO, SHUTTER* ou autre). Ensuite, la fonction recherche dans le tableau *motors[]* le moteur dont le nom correspond à celui fourni dans la commande.

Pour un *MOVE*, la fonction lit le nombre de pas et la direction (0 ou 1), puis appelle *moveStepper()* avec ces paramètres. Pour un *ZERO*, elle appelle simplement *findZero()* sur le moteur sélectionné. Si le nom du moteur est inconnu ou la commande mal formée, aucune action n'est entreprise et un message d'erreur est envoyé au programme python.

1.2 Programme Python

1.2.1 main.py

Ce script contient la logique principale du code python et est chargé de la communication entre le système et l'utilisateur.

createConfiguredMonochromators

C'est ici que l'on crée les objets représentant les monochromateurs d'excitation et d'émission. On lit le fichier de configuration YAML, on détecte pour chaque monochromateur s'il s'agit d'un modèle de type A ou B, puis on crée un objet MonochromatorA ou MonochromatorB en conséquence.

Pour un type A, l'objet est instancié avec uniquement la relation longueur d'onde - pas, le port série et les limites de pas. Pour un type B, en plus des paramètres précédents, on doit créer un objet Slit (fente) avec ses propres paramètres. Chaque objet ainsi créé est ensuite stocké dans un dictionnaire avec pour clé "excitation monochromator" ou "emission monochromator".

createMeasurements

Une fois les monochromateurs instanciés, cette fonction génère les objets de mesure. On lit ici un second fichier YAML (dans le dossier Measure Config) qui décrit les mesures à effectuer : type (émission, excitation, synchrone ou unique), temps d'intégration, résolution et plages de longueurs d'onde.

Chaque type de mesure est associé à une classe différente (EmScanMeasure, ExScanMeasure, etc.), et les objets sont instanciés avec les bons monochromateurs, les longueurs d'onde, et la configuration de balayage. Le dictionnaire retourné contient tous les scans à exécuter.

interactiveRun()

C'est la fonction principale du script, exécutée automatiquement à la fin. Voici son déroulement logique :

1. Lecture de la configuration des monochromateurs depuis le YAML config système.
2. Création des objets Monochromator via createConfiguredMonochromators.
3. Proposition à l'utilisateur d'initialiser les moteurs (via .initMotors()).
4. Affichage des fichiers YAML de configuration de mesure disponibles dans le dossier.
5. Sélection du fichier de mesure.
6. Création des objets Measure via createMesurements.
7. Affichage d'informations utiles : longueurs d'onde min/max, plage de résolution.
8. Confirmation utilisateur pour lancer ou annuler la mesure.
9. Exécution des mesures : pour chaque scan, appel à measure.measure() puis sauvegarde des résultats avec measure.saveResults().

1.2.2 Classe Wavelength

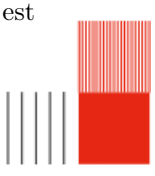
L'objet WaveLength représente une valeur physique de longueur d'onde exprimée en nanomètres (nm). Ce choix d'encapsulation permet une abstraction propre : on ne manipule jamais directement des float, mais des objets dédiés, ce qui ouvre la voie à des conversions et opérations intégrées sans alourdir le code principal.

Lors de son initialisation, on fournit une valeur numérique unique, stockée dans l'attribut self.value. Toutes les opérations numériques ou conversions sont alors basées sur cette valeur.

Conversions

Deux méthodes sont déjà disponibles pour convertir la longueur d'onde :

— to_waveNumber() transforme la longueur d'onde en nombre d'ondes via la formule : $\nu = \frac{10^4}{\lambda}$
avec λ en nm et le résultat en cm^{-1}. Elle intègre un contrôle d'erreur si la longueur d'onde est nulle.



— to_frequency() convertit en fréquence (Hz) via la relation : $f = \frac{c}{\lambda}$
où la vitesse de la lumière c est prise comme constante en m/s. Ici aussi, un ValueError est levé si $\lambda = 0$ pour éviter toute division par zéro.

Même si ces fonctionnalités ne sont pas encore exploitées dans le code principal, elles permettent déjà de visualiser ou stocker les données dans d'autres unités sans avoir à dupliquer de logique ailleurs.

1.2.3 Classe WLRange

La classe WLRange regroupe un intervalle de longueurs d'onde : minimum, maximum, et pas (wLStep). Tous trois sont des objets WaveLength.

Cette classe permet de décrire proprement les gammes de balayage utilisées dans les mesures (émission, excitation, etc.), en évitant les erreurs de type ou d'unité. Elle vérifie aussi que wLMin est bien inférieur à wLMax, ce qui protège des erreurs de configuration en amont.

1.2.4 Classe IntegrationTime

La classe IntegrationTime encapsule simplement un temps d'intégration, en millisecondes. Elle sert à garantir que les valeurs utilisées soient toujours positives et raisonnables (entre 0 et 1 000 000 ms), en centralisant les contrôles dans un objet dédié.

Elle offre une méthode to_seconds() pour convertir en secondes pour l'affichage et pour les fonctions de temporisation.

1.2.5 Classe Slit

La classe Slit, utilisée exclusivement dans les monochromateurs de type B, représente la fente réglable électroniquement en termes de résolution spectrale. Elle encapsule à la fois la conversion entre pas moteur et largeur de fente (exprimée ici en longueur d'onde effective), le dialogue série avec l'Arduino, et les contrôles d'intégrité liés aux déplacements. Voici une explication détaillée de ses attributs et méthodes.

Instanciation

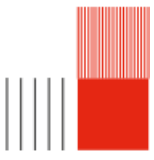
Le constructeur __init__() initialise les caractéristiques d'une ou plusieurs fentes. Il prend notamment un numberOfSlits (3), le nombre de fentes dans le monochromateur, une valeur de longueur d'onde courante (objet WaveLength), un décalage (offsetWL) et un coefficient pour la conversion pas - valeur physique (coefWL). On peut également fournir une position actuelle (step) et les bornes (minStep, maxStep). L'objet série serialConnection représente le lien de communication avec le microcontrôleur. Une fois l'objet créé, la méthode updateWaveLength() permet de calculer la valeur actuelle de la fente à partir du pas courant.

Conversion pas-longueur d'onde

Les méthodes getValueFromStep() et getStepFromValue() sont des fonctions de conversion. La première convertit un nombre de pas en une longueur d'onde équivalente (selon la calibration spécifique à chaque fente), tandis que la seconde fait l'opération inverse. Ces méthodes permettent une grande souplesse de manipulation : on peut raisonner en longueur d'onde plutôt qu'en position moteur, tout en gardant un pilotage précis en pas moteur en arrière-plan.

Trouver la position zéro

La méthode findZero() permet de retrouver la position zéro des fentes, ce qui correspond à une calibration physique du système. Pour chaque fente (généralement trois), la méthode envoie une commande série de type ZERO,SLITx au microcontrôleur, puis attend une réponse. Si la réponse est "ZERO,DONE", le pas est mis à 0 et la valeur réinitialisée à offsetWL. En cas d'échec ou de dépassement de temps, une



erreur est levée. Ce processus est indispensable pour garantir une référence stable avant tout déplacement.

Déplacement

Les méthodes moveToValue() et moveToPercentage() permettent de déplacer les fentes à une position donnée, soit en longueur d'onde, soit en pourcentage de l'ouverture maximale. Dans les deux cas, on calcule d'abord le nombre de pas à effectuer, la direction du déplacement, puis on envoie la commande MOVE,SLITx,nombre_de_pas,direction à chaque fente. On attend ensuite une réponse "MOVE,DONE" pour valider le mouvement. Une fois le déplacement terminé, la position (step) et la valeur associée (WaveLength) sont mises à jour.

1.2.6 Classe Monochromator

Les classes Monochromator, MonochromatorA et MonochromatorB modélisent le comportement de deux types distincts de monochromateurs contrôlés via port série. Ces objets pilotent les déplacements moteurs pour sélectionner des longueurs d'onde précises, et dans le cas du type B, gèrent également les fentes motorisées. Toute la structure repose sur une classe mère abstraite (Monochromator) qui formalise l'interface commune, ensuite spécialisée dans MonochromatorA et MonochromatorB.

La classe MonochromatorA hérite de la classe mère et correspond à un modèle de monochromateur où la relation entre le nombre de pas et la longueur d'onde est linéaire. Elle implémente donc les deux méthodes de conversion en utilisant une simple équation affine : $\lambda = \lambda_0 + \alpha \cdot k_{pas}$ pour aller du pas à la longueur d'onde, et l'inverse pour la commande. Elle redéfinit aussi la fonction initMotors(), qui attend le signal d'initialisation du microcontrôleur avant de lancer le findZero().

La classe MonochromatorB représente quant à elle les modèles plus complexes, notamment les Fluorolog, où la conversion entre pas et longueur d'onde est non-linéaire. Ici, la relation est de la forme $\lambda = \lambda_0 + \alpha \cdot \sin(\beta \cdot k_{pas})$ et l'inverse utilise asin(), ce qui reflète la géométrie du mécanisme optique. En plus des méthodes classiques héritées de la classe mère, elle prend en charge un objet Slit représentant les trois fentes motorisées. C'est lors de la création de l'objet que la connexion série est transmise à l'objet Slit, assurant un canal de communication unique pour l'ensemble du module. Sa méthode initMotors() appelle donc à la fois findZero() pour le moteur principal et findZero() pour les fentes. Elle peut aussi modifier dynamiquement la résolution via la méthode setResolution(), qui agit directement sur les fentes.

1.2.7 Classe Signal

La classe Signal, définie dans le fichier spectro_signal.py, représente une courbe de mesure. Elle encapsule à la fois un nom pour identifier la nature du signal (par exemple "émission", "référence", etc.) et une liste de valeurs numériques représentant l'intensité mesurée à différentes longueurs d'onde.

Si aucune donnée n'est fournie à la création, la liste est simplement vide. Cela permet de construire progressivement un signal, soit en l'alimentant au fur et à mesure des acquisitions, soit en important directement une liste existante.

La classe offre plusieurs méthodes pour enrichir ou modifier le signal. add_signal() permet d'ajouter une liste entière de valeurs à la suite des données existantes. append_signal() ajoute une valeur unique, ce qui est utile pour une construction point par point lors d'un balayage. clear_signal() réinitialise complètement le signal.

1.2.8 Classe Measure

Les classes définies dans le fichier measure.py forment le cœur de la logique de mesure du spectrofluorimètre. L'architecture est pensée de manière modulaire autour d'une classe de base abstraite Measure, qui définit les fondations communes à toutes les mesures possibles, puis étendue par plusieurs sous-classes qui implémentent chacune un mode de mesure spécifique.

La classe Measure n'est pas destinée à être utilisée directement. Elle encapsule les éléments communs à toute mesure : les plages de longueurs d'onde (excitatrice et émissive), le temps d'intégration, les deux monochromateurs utilisés (un pour l'excitation, l'autre pour l'émission), et enfin les signaux mesurés et de référence. Le constructeur valide notamment que les deux monochromateurs ont bien les rôles attendus (ex et em) pour éviter des erreurs de configuration. Quelques méthodes abstraites (measure, plot, saveResults) obligent les sous-classes à fournir leur propre implémentation de la logique.

La classe uniqueWLMeasure représente une mesure à deux longueurs d'onde fixes (excitatrice et émissive). Lors de l'appel à la méthode measure, chaque monochromateur se déplace vers sa longueur d'onde cible, puis les shutters s'ouvrent pour permettre la mesure du signal et de la référence. Une fois cela effectué, les shutters se ferment. La méthode saveResults enregistre les valeurs obtenues dans un fichier CSV, incluant date, heure, et noms des longueurs d'onde.

Les autres sous-classes étendent la logique à des balayages spectroscopiques. synchroScanMeasure permet de déplacer simultanément les deux longueurs d'onde avec un décalage constant entre excitation et émission, ce qui est typiquement utilisé pour suivre des comportements dépendants du déplacement spectral. ExScanMeasure fixe l'émission et balaye l'excitation, tandis que EmScanMeasure fait l'inverse. Ces classes déclenchent les mesures, et enregistrent les résultats de manière similaire à uniqueWLMeasure.

Cette structure permet d'étendre facilement le système à de nouveaux types de mesures en dérivant Measure sans devoir réécrire la logique de bas niveau comme l'ouverture des shutters ou la gestion des signaux.

Bibliographie