Intégration Numérique pour la Mesure et l'Analyse FROSt

- Enzo Sebiane -

Table des matières

1 Introduction **2**
1.1 Sujet d'étude . . . 2
1.2 Méthode FROSt (Frequency Resolved Optical Switching) . . . 2

2 Conception d'une Interface de Contrôle et d'Acquisition **3**
2.1 Le moteur KDC101 de Thorlabs . . . 4
2.2 Le spectromètre Mozza de Fastlite . . . 5
2.3 Liaison des modules et gestion des threads . . . 6

3 Conception et Mise en Œuvre d'une Interface GUI pour l'Analyse des Données FROSt **8**
3.1 Algorithme de Récupération et Prétraitement des Traces FROSt . . . 8
3.2 Interface Graphique . . . 9

4 Introduction aux Architectures de Réseaux Neuronaux **10**
4.1 Génération des données . . . 11
4.1.1 Architecture . . . 11
4.1.2 FROStNET . . . 12
4.2 Les différents modèles . . . 12
4.2.1 CNN1 Single/Dual output . . . 12
4.2.2 CNN2 Single/Dual output . . . 13
4.3 Entraînement du modèle et modélisation . . . 13

5 Utilisation de Luna pour la Modélisation et la Conception d'une IA **15**
5.1 Présentation de Luna . . . 16
5.1.1 Situation Problème . . . 16
5.2 Génération du Dataset . . . 16
5.3 Modélisation avec les LSTM . . . 17
5.4 Modélisation avec les Réseaux de Neurones Convolutifs (ConvNet) . . . 19

6 Développement d'une Interface Polyvalente sous PyMoDAQ **21**
6.1 Présentation de PyMoDAQ . . . 21
6.2 Adaptation des Plugins pour le Moteur et les Spectromètres . . . 21
6.2.1 Implémentation du Plugin pour le Moteur KDC101 . . . 21
6.2.2 Utilisation du Plugin pour les Spectromètres Ocean Optics . . . 23
6.2.3 Implémentation du Plugin pour les Spectromètres Avantes . . . 24
6.2.4 Implémentation du Plugin pour les Spectromètres Mozza de Fastlite . . . 25
6.3 Utilisation du Dashboard . . . 26

7 Conclusion **27**
7.1 Bilan . . . 27
7.2 Remerciements . . . 27

1 Introduction

Un domaine centrale de la recherche au sein de l'Institut Nationale de la Recherche Scientifique (INRS) est la recherche sur les pulsations ultra-brèves. Notamment en raison de la puissance crête à crête très importantes qu'elles peuvent fournir permettant de relever certaines dynamique non-linéaires inaccessible autrement. Pour citer seulement deux des applications ces impulsions sont utilisé pour l'ARPES (Angle-Resolved Photoemission Spectroscopy), qui examine les propriétés électroniques des matériaux avec une haute résolution temporelle, ou encore pour la génération de seconde harmonique (SHG), qui améliore la précision des mesures optiques non linéaires.

1.1 Sujet d'étude

Dans ce contexte, ma recherche se concentre sur la caractérisation de ces pulsations ultrabrèves à travers la méthode FROSt (Frequency-Resolved Optical Switching). Dans le cadre de la génération de pulsations laser ultra-brèves, la méthode de mesure FROSt a été développée par les laboratoires de l'INRS. Mon stage se présentait alors sous forme de plusieurs missions spécifiques :

- **Compréhension de la technique FROSt** : Approfondir les principes et les mécanismes de la méthode de mesure FROSt, afin de bien comprendre son fonctionnement et ses applications.
- **Familiarisation avec les outils de mesure** : Me familiariser avec les instruments et logiciels utilisés pour acquérir les spectres générés par la méthode FROSt, en apprenant à les utiliser de manière efficace.
- **Développement d'une interface graphique** : Créer une interface graphique permettant la configuration, la mise en place et l'acquisition des mesures FROSt, facilitant ainsi l'utilisation de la méthode par les chercheurs.
- **Exploration de l'intégration de l'IA** : Investiguer les possibilités offertes par l'intelligence artificielle pour améliorer la rapidité et l'efficacité des mesures FROSt, et tenter d'implémenter des solutions plus efficaces à certains calculs.

1.2 Méthode FROSt (Frequency Resolved Optical Switching)

Pour bien comprendre les travaux de recherche, une étude bibliographique s'est imposée. Deux méthodes sont alors importantes à relever :

La méthode de *Frequency-Resolved Optical Gating* (FROG) [1] qui est une technique avancée pour la caractérisation des impulsions optiques ultracourtes. Elle repose sur l'interaction non linéaire entre une impulsion de test et une impulsion de référence dans un cristal non linéaire, permettant de récupérer la forme temporelle de l'impulsion à partir de son spectre résolu en fréquence. La méthode FROG est capable de mesurer à la fois la phase et l'amplitude des impulsions optiques, en décomposant le spectre en spectrogrammes qui reflètent les variations de l'impulsion en fonction du décalage temporel entre les impulsions.

La méthode FROSt (Frequency Resolved Optical Switching) [2] qui se distingue par son utilisation de l'absorption transitoire dans les solides comme commutateur optique ultrarapide, sans contraintes d'accord de phase. Elle utilise un montage de type pompe-sonde : une impulsion de pompe induit l'absorption transitoire, tandis que l'impulsion à caractériser sonde la transmissivité du matériau photoexcité. Ce dispositif permet de caractériser des impulsions ultra-broadband à faible intensité et dans une large gamme de longueurs d'onde, allant de 0,5 à 20 μm, grâce à l'utilisation de matériaux comme le séléniure de zinc (ZnSe). La ptychographie est employée pour récupérer les profils temporels des impulsions de sonde et du commutateur optique.

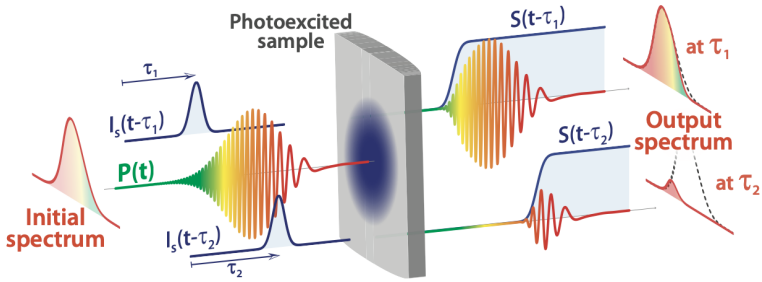


FIGURE 1 – Schéma de la méthode FROSt illustrant l'interaction entre l'impulsion de pompe et l'impulsion de sonde dans l'échantillon solide.

L'installation expérimentale pour la méthode FROSt est illustrée dans la Figure 2. Une impulsion de pompe et une impulsion de sonde sont focalisées sur un échantillon solide mince, avec un angle relatif de 3° entre les deux faisceaux. L'impulsion de sonde est focalisée à l'aide d'un miroir parabolique décalé, tandis que l'impulsion de pompe passe à travers un trou dans la parabole. La sonde est ensuite réimagée dans un spectromètre après avoir traversé l'échantillon. Un dispositif de retard ajuste le délai entre les impulsions. Des milieux dispersifs peuvent être ajoutés pour tester la robustesse de la technique en introduisant une dispersion de vitesse de groupe. La méthode FROSt permet la caractérisation d'impulsions avec des longueurs d'onde allant de 0,77 à 10 μm et des durées allant jusqu'à moins de deux cycles optiques, avec des énergies aussi faibles que quelques nanojoules.

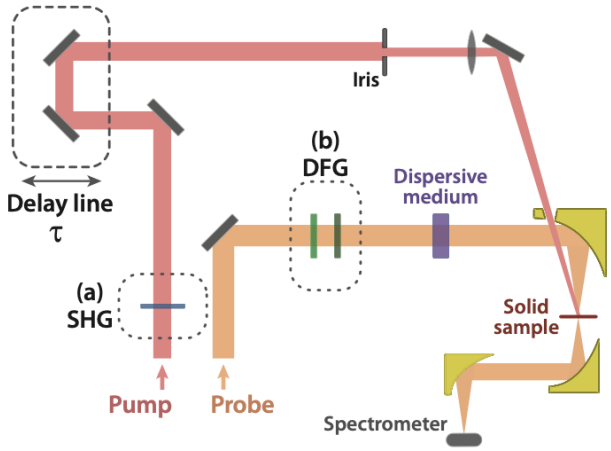


FIGURE 2 – Schéma expérimental pour la caractérisation des impulsions courtes avec la méthode FROSt.

2 Conception d'une Interface de Contrôle et d'Acquisition

Avant de discuter des modules d'acquisition, il a été nécessaire de me familiariser avec les bases du développement, telles que l'installation de PyCharm afin de créer un environnement virtuel pour exécuter mes codes, ainsi que l'utilisation de GitHub pour partager et sauvegarder efficacement mon travail. Une fois ces étapes préliminaires réalisées, j'ai pu me concentrer sur le matériel à ma disposition.

2.1 Le moteur KDC101 de Thorlabs

Le premier élément sur lequel je me suis concentré a été le moteur KDC101 de Thorlabs. Ce moteur, associé au dispositif Z825B, permet de contrôler des mouvements avec une précision de l'ordre de 10^{-4} mm. Bien qu'il soit possible de contrôler manuellement ce moteur, il est essentiel, dans le cadre d'une acquisition FROG ou FROSt, de pouvoir automatiser les mouvements pas à pas depuis un code.

Avant de s'attarder sur la création d'une interface graphique, j'ai d'abord voulu programmer ce contrôle. Pour ce faire, il m'a fallu comprendre l'utilisation des bibliothèques de liens dynamiques (DLLs). Ces DLLs contiennent des fonctions pré-compilées, offrant une interface permettant de contrôler le moteur sans nécessiter une compréhension approfondie des détails matériels sous-jacents.

Voici un exemple de début de code où, une fois les DLLs Kinesis de Thorlabs installées, elles sont importées :

```
>>
# Importer les bibliothèques Kinesis
dll_path = r'C:\\Program Files\\Thorlabs\\Kinesis'
if dll_path not in sys.path:
    sys.path.append(dll_path)

clr.AddReference('Thorlabs.MotionControl.DeviceManagerCLI')
clr.AddReference('Thorlabs.MotionControl.KCube.DCServoCLI')
from Thorlabs.MotionControl.KCube.DCServoCLI import KCubeDCServo
from Thorlabs.MotionControl.DeviceManagerCLI import DeviceManagerCLI
```

De cette manière, on importe la bibliothèque capable de gérer les dispositifs connectés (`DeviceManagerCLI`) ainsi que celle permettant les contrôles directs du moteur (`KCubeDCServo`).

Le schéma suivant résume le fonctionnement général de la commande de ce type de moteur :

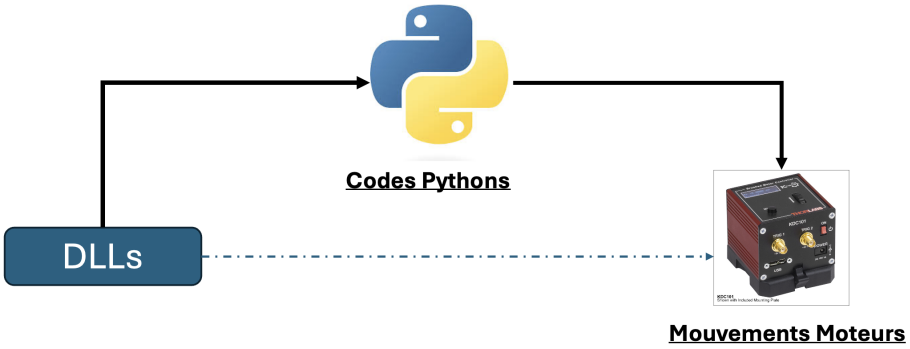


FIGURE 3 – Schéma illustrant le fonctionnement de Dlls pour le contrôle du moteur

Une fois ce travail effectué et après avoir réussi à piloter le moteur, il était nécessaire de commencer à penser à l'utilisateur, c'est-à-dire à créer une interface à partir de notre code initial.

Pour ce faire, j'ai utilisé `tkinter` :

```
>>
import tkinter as tk
from tkinter import messagebox
from Methods_gui import KCubeDCServoController
```

`tkinter` permet de créer une interface utilisateur de manière très simple en générant des cadres (`frames`), des boîtes de message (`messagebox`), des étiquettes (`labels`) ou encore des boutons, qui

appellent la classe créée précédemment pour contrôler le moteur. Honnêtement, une fois le code de backend bien structuré sous forme de classe, créer une interface revient essentiellement à rendre le code plus esthétique, ce qui s'est avéré relativement simple.

Ainsi, notre acquisition se rapproche de notre objectif :

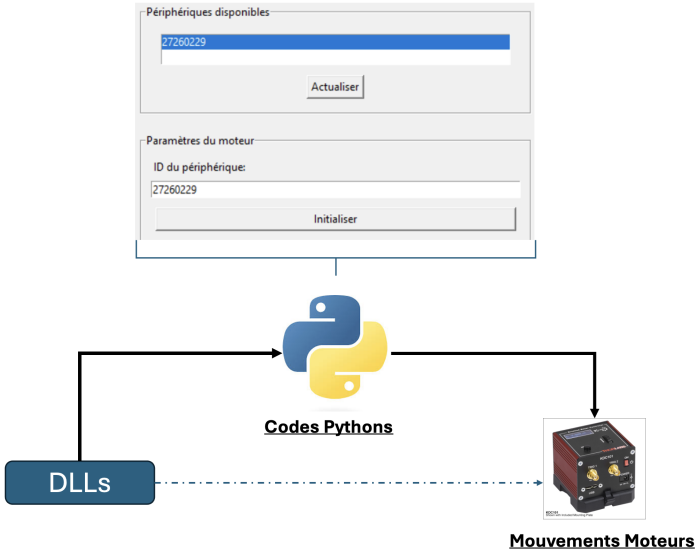


FIGURE 4 – Schéma illustrant le fonctionnement de Dlls pour le contrôle du moteur avec une interface créee sous tkinter

Avec le contrôle des mouvements du moteur bien maîtrisé, nous pouvions désormais commencer à nous intéresser au spectromètre.

2.2 Le spectromètre Mozza de Fastlite

Le spectromètre étudié dans ce projet était le spectromètre Mozza de Fastlite, performant dans la plage de 2000 nm à 5000 nm. Fastlite est une entreprise spécialisée dans la conception d'instruments pour la manipulation et la caractérisation d'impulsions laser ultra-courtes.

Dans un premier temps, il était essentiel de faire fonctionner ce spectromètre. J'ai donc installé le logiciel Mozza (`.exe`) fourni par Fastlite. À cette étape, aucune difficulté particulière n'a été rencontrée : il était possible d'acquérir des signaux de bruit, d'observer leur évolution et d'accéder à plusieurs paramètres tels que la fréquence de déclenchement (*trigger frequency*), le temps d'acquisition (*acquisition time*), ou encore le pas de longueur d'onde (*wavelength step*).

Le problème majeur était que, avec un fichier `.exe`, je n'avais aucun accès au code source. Or, dans le cadre de mon projet d'acquisition de traces, il était essentiel de pouvoir synchroniser les mouvements pas à pas du moteur avec l'acquisition des spectres par le Mozza.

Après plusieurs échanges de courriels avec Fastlite, j'ai pu obtenir un fichier `.whl` me permettant de récupérer la bibliothèque (DLLs) `libmozza`.

Plus précisément, une fois le dossier `libmozza` partagé par Fastlite copié dans mon environnement de travail, j'ai exécuté la commande suivante :

```
>> pip install libmozza-1.0.2-py3-none-any.whl
```

Ainsi, les DLLs étaient installées dans mon interpréteur Python.

Ensuite, il fallait non seulement acquérir les données spectrales qui nous intéressaient, mais aussi les stocker et les afficher. De la même manière que pour le moteur, nous pouvions commencer notre code par la commande suivante :

```
>> from libmozza.mozza import MozzaUSB, MozzaError
```

Grâce au logiciel `depends.exe`, j'ai pu observer les fonctions "cachées" dans la bibliothèque `libmozza`. Après quelques essais et en utilisant des fonctions telles que `mozza.get_serials()` qui permet d'obtenir dynamiquement le numéro de série du spectromètre (ce qui offre une grande modularité), j'ai pu créer l'axe des abscisses avec la fonction `mozza.set_wavenumber_array()` et acquérir mon spectre avec `mozza.process_spectrum()`. Finalement, j'ai pu afficher simplement le spectre à l'aide de `matplotlib`, obtenant ainsi :

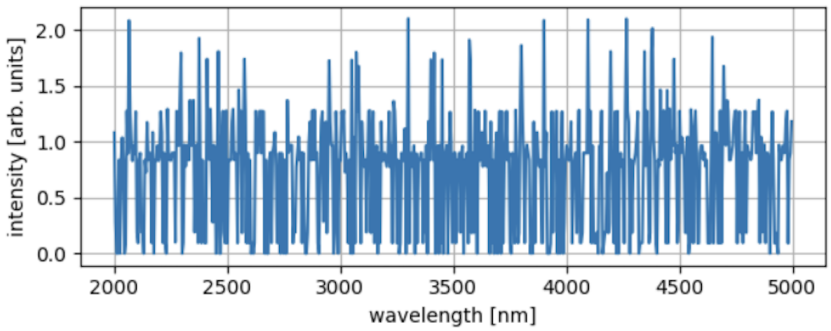


FIGURE 5 – Affichage des rawdata récupéré via libmozza et affiché sous matplotlib après une acquisition

Une fois cela réussi il était facile d'y ajouter des paramètres d'acquisitions permettant de faire varier le résultat de la même manière que dans le .exe mis a notre disposition par Fastlite

2.3 Liaison des modules et gestion des threads

Cette section présente le cœur du travail réalisé, car en cas de réussite de la mise en relation du moteur et du spectromètre, nous obtiendrions notre premier prototype d'acquisition de traces FROSt.

Cependant, ce ne fut pas sans difficultés. Tout d'abord, il a fallu structurer de manière optimale les codes du KDC101 et du Mozza sous forme de classes, chacune appelant des méthodes dans des scripts bien définis. Cette fois-ci, il ne suffisait pas que les codes fonctionnent individuellement, il fallait les appeler pour les faire fonctionner ensemble.

Une étape importante était de lier les deux classes. La première liaison logique consistait à ce que la classe du moteur renvoie un booléen à la classe du spectre une fois un pas effectué, pour que la classe du spectre lance une acquisition. Mais cela ne suffisait pas. Pour avoir une interface digne de ce nom, il fallait que l'acquisition soit dynamique. En laboratoire, on ne lance pas une acquisition immédiatement ; on se place autour de la zone de fonctionnement et on s'ajuste en regardant en temps réel ce que montre le spectrogramme.

J'ai donc créé une fonction qui met à jour l'acquisition du Mozza toutes les 3 ms, afin de donner l'impression d'une acquisition continue. Dans cette fonction, une boucle `if` reliée au booléen renvoyé par le moteur permet l'enregistrement d'un spectre.

Une fois cette tâche accomplie, j'ai créé un nouveau script nommé `MainWindow`, toujours avec `tkinter`, initialisé de la manière suivante :

```
>>
import tkinter as tk
```

```
from PIL import Image, ImageTk
from MotorCodes.Gui.interface_gui import MotorControllerGUI
from SpectroCodes.Gui_Periodic_plot import SpectroGUI
```

Voici le résultat de l'interface après un peu de mise en forme et l'insertion de logos et d'icônes (libres de droits, bien sûr) :

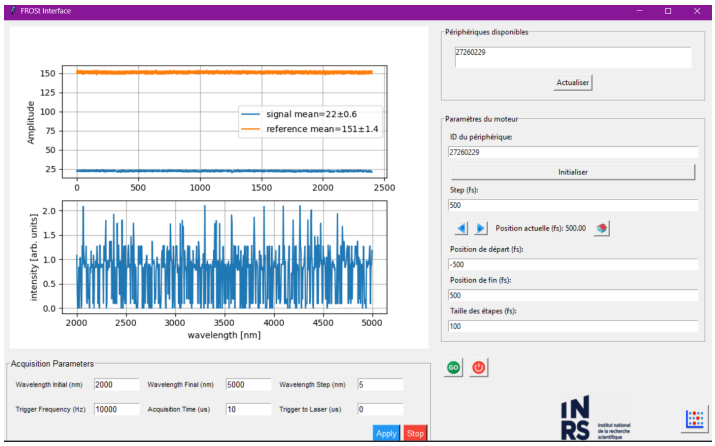


FIGURE 6 – Interface finale appelant dans une MainWindow la classe du spectromètre (sur la gauche) et la classe du moteur (sur la droite

À ce stade, le résultat visuel était satisfaisant, mais ce n'était pas encore complètement fonctionnel. Un problème physique que je n'avais pas initialement identifié se posait : le moteur se déplace en millimètres, mais nous travaillons en femtosecondes. Or, dans le contexte des lasers, qui se déplacent à la vitesse de la lumière, il y avait un piège : 1 mm de déplacement du moteur fait parcourir le double à la lumière. Il était donc nécessaire de modifier la classe du moteur comme suit :

```
>> def update_position_labels(self):
    current_fs = self.current_position / (C_MM_PER_FS / 2)
```

avec `current_position` représentant la position actuelle du moteur et `C_MM_PER_FS` la vitesse de la lumière en mm/fs (3e-4).

Un second problème concernait l'affichage des résultats. Pour y remédier, j'ai ajouté une fonction dans la boucle principale d'acquisition du Mozza qui, à chaque acquisition de spectre, crée un dossier `acquisition` (si ce n'est pas déjà fait) et, à la première itération, crée un sous-dossier avec la date et l'heure de l'acquisition, puis enregistre les spectres un à un dans ce répertoire en fonction des ordres du moteur. Finalement, un dernier script affiche successivement ces acquisitions et se déclenche lorsque l'utilisateur clique sur le bouton en bas à droite.

À ce stade, l'interface fonctionnait, mais la mise à jour du spectre ralentissait tellement les autres commandes saisies par l'utilisateur qu'il était nécessaire de mettre en pause l'acquisition pour manipuler le moteur. Pour pallier cette difficulté, j'ai mis en place des threads. Un *thread* est une unité d'exécution distincte au sein d'un programme, permettant de réaliser plusieurs tâches simultanément (en parallèle). Chaque thread fonctionne indépendamment, mais partage la même mémoire et les mêmes ressources que les autres threads du programme. En pratique, j'ai d'abord essayé de créer un thread pour l'acquisition de spectres, mais j'ai finalement opté pour un thread pour le déclenchement du mouvement du moteur, car la mise à jour dynamique était l'information la plus importante :

```
>> def start_movement(self):
    threading.Thread(target=self._start_movement).start()
```

Malheureusement, je n'ai pu effectuer des acquisitions qu'à mon bureau, sur du bruit, mais le résultat semble tout de même satisfaisant.

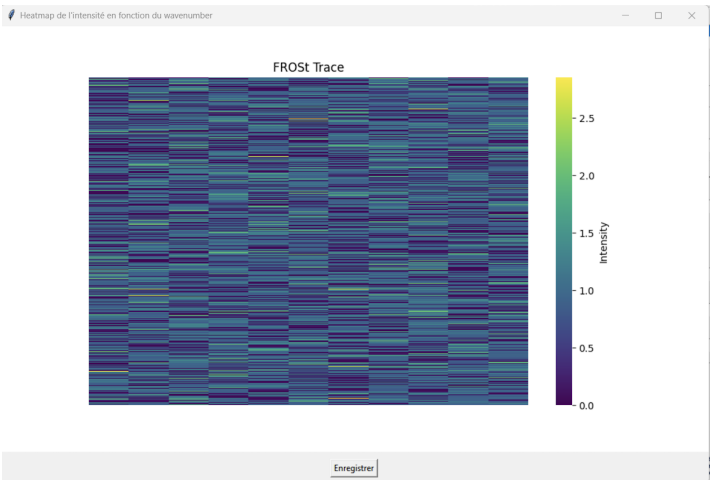


FIGURE 7 – Affichage de la trace acquis via l'interface

On peut alors se demander à juste titre comment se servir de ces acquisitions ?

Github : https://github.com/Enzo77360/FROSt_Interface

3 Conception et Mise en Œuvre d'une Interface GUI pour l'Analyse des Données FROSt

Après la réalisation du premier prototype de l'interface, une collaboration a été entamée avec Sydney Lepard pour mieux comprendre le fonctionnement de l'analyse des données FROSt.

En discutant avec Mayank, il est apparu que bien que les codes de prétraitement et de récupération des données FROSt soient bien établis, ils étaient en MATLAB. Or, MATLAB n'est pas forcément convivial pour des personnes n'ayant pas une formation en informatique et, de plus, il n'est pas une licence gratuite.

3.1 Algorithme de Récupération et Prétraitement des Traces FROSt

La première étape a consisté à récupérer les codes MATLAB du prétraitement, puis à les adapter pour Python. Cette tâche, bien que relativement standard, a présenté quelques difficultés, notamment en raison des différences dans le comportement des transformations de Fourier entre MATLAB et Python.

Concernant le cœur du code MATLAB pour la récupération des traces après prétraitement, les calculs reposent sur un algorithme de ptycographie. Étant donné leur complexité, ces calculs étaient effectués en MATLAB en appelant des fonctions écrites en C. L'adaptation de ces fonctions a été réalisée par Sydney. Mon rôle s'est limité à discuter et suivre ses progrès, ce qui m'a néanmoins permis de comprendre les bases de Cython, permettant de faire tourner des scripts C dans Python (avec le format `.pyx`).

Les différentes étapes du programme peuvent être résumées comme suit :

Étape 1 : Prétraitement Initial Cette première étape se concentre sur la préparation des données brutes. Elle commence par la *découpe des axes temporels*, où les limites de découpe (`t_crop`) sont définies pour extraire la portion pertinente de la trace (`trace1`). Ensuite, les *valeurs anormales* sont

éliminées en remplaçant les valeurs aberrantes par la moyenne des valeurs voisines afin de minimiser les erreurs dans les étapes ultérieures. Enfin, un *prétraitement de la trace* est réalisé, comprenant la normalisation des données, la suppression du fond par une opération d'ouverture en niveaux de gris, et l'élimination des valeurs négatives.

Étape 2 : Filtrage et Transformation La phase de filtrage et transformation est essentielle pour la conversion des données en un format analysable. Cette étape débute avec la *découpe temporelle*, où les tranches temporelles pertinentes (`t_slice`) sont extraites pour analyser des segments spécifiques de la trace. La *transformée de Fourier* est ensuite appliquée à chaque ligne du spectrogramme pour passer du domaine temporel au domaine fréquentiel. Un *filtre hyper-gaussien* est également utilisé pour éliminer les bruits de haute fréquence, suivi d'une inversion en temps de la trace filtrée.

Étape 3 : Correction et Ajustement Cette étape comprend la *calculation des profils non perturbés et perturbés*, où les profils `S_NP` et `S_P` sont calculés et comparés pour estimer les effets perturbateurs. Un *modèle de fond* est estimé et soustrait pour améliorer la précision des mesures.

Étape 4 : Lissage et Interpolation Pour améliorer la qualité des résultats, un *lissage* de la trace est effectué à l'aide d'un filtre gaussien afin de réduire le bruit. La trace est ensuite *interpolée* pour obtenir des vecteurs de fréquence et de délai avec un pas constant.

Étape 5 : Découpe et Filtrage Final Cette étape inclut la *découpe des fréquences*, où les plages de fréquences sont ajustées pour se concentrer sur les régions d'intérêt. Un *filtrage supplémentaire* est appliqué et les données sont ajustées pour compenser les effets de fréquence.

Étape 6 : Récupération et Post-traitement Enfin, la *récupération et le post-traitement* des données sont effectués. Cela inclut l'*interpolation des données* sur un nouveau vecteur de délai pour ajuster la taille temporelle et obtenir une résolution uniforme. Les nouveaux *paramètres temporels* sont calculés en fonction des interpolations pour créer un vecteur de temps ajusté. La trace est *étendue et normalisée* pour améliorer la lisibilité et la précision, avant le *calcul et affichage des résultats*, qui comprend les traces récupérées et les paramètres ajustés pour une analyse approfondie.

3.2 Interface Graphique

La principale tâche de mon travail a été de mettre en forme cet algorithme sous la forme d'une interface utilisateur.

Cependant, comme les codes initiaux en MATLAB et en C ont été rédigés manuellement par un ancien doctorant avant d'être convertis en Python, il était impossible de structurer correctement le code sous forme de classes pour obtenir une fenêtre principale appelant les éléments étape par étape. En raison des dépendances entre les étapes, il a fallu opter pour une solution alternative : enchaîner de nouvelles fenêtres à chaque étape, structurées par exemple comme suit pour l'étape 2 :

```
## STEP2
def init_step2(self): ...
    self.step2()

def step2(self): ...
    self.plot_step2()

def plot_step2(self): ...
    self.canvas.draw()
```

Ainsi, la fin de `plot_step2` appelle `init_step3`, et ainsi de suite.

Après mise en forme et ajout de logos et d'icônes, l'interface utilisateur prend la forme suivante :

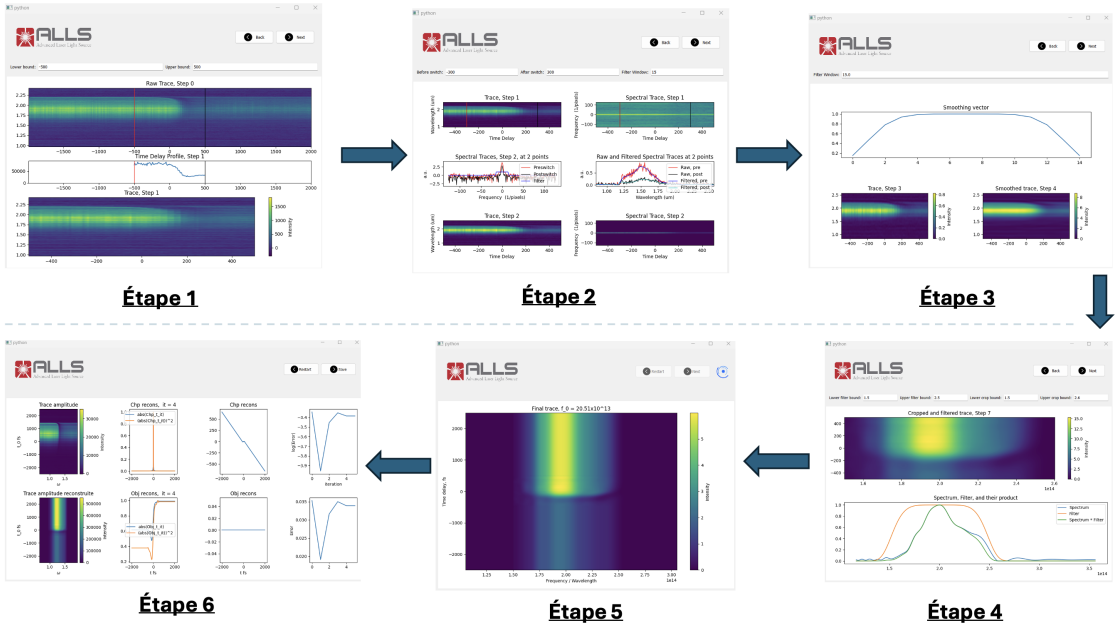


FIGURE 8 – Schéma globale représetnant l'ensemble des fenêtres des étapes après création d'un Gui sous python

Bilan L'objectif initial était de créer une application sous forme de fichier exécutable Python ou via Docker. Cependant, la structure actuelle n'est pas encore suffisamment propre pour ce type de déploiement ; par exemple, l'appel des fonctions C via Cython pose encore des problèmes, rendant la solution peu modulable.

Néanmoins, grâce à un raccourci, tout utilisateur pourra facilement effectuer un prétraitement et récupérer des traces FROSt via une interface intuitive et claire, sans avoir à consulter le moindre code source.

Ainsi, le travail se poursuit et l'intérêt se tourne désormais vers les recherches de Sydney Lepard, qui pourraient potentiellement simplifier considérablement le processus de récupération des traces.

Github : https://github.com/Enzo77360/Ptychography

4 Introduction aux Architectures de Réseaux Neuronaux

En explorant les travaux de Sydney Lepard, j'ai découvert pour la première fois le domaine de l'intelligence artificielle (IA) à travers les réseaux de neurones. Un réseau de neurones est un modèle informatique inspiré du cerveau humain, constitué de couches de neurones artificiels interconnectés. Chaque neurone reçoit des entrées, les transforme à l'aide d'une fonction d'activation, puis transmet une sortie aux neurones suivants. Les réseaux de neurones sont utilisés pour reconnaître des motifs, faire des prédictions ou classifier des données en apprenant à partir d'exemples, grâce à l'ajustement des poids des connexions entre les neurones.

Dans les premiers jours, j'ai rédigé un compte rendu sur le fonctionnement de base des réseaux de neurones, que je vous invite à consulter sur mon github

Mais qu'en est-il du projet de Sydney ? Lors de mon intégration au groupe, celui-ci consistait à récupérer les impulsions initiales à partir de la trace FROSt [3]. Pour ce faire, il est nécessaire d'effectuer un entraînement supervisé : nous générons des impulsions à l'aide d'un code, que nous transformons

ensuite en trace FROSt. Notre réseau de neurones récupère une impulsion que nous comparons avec la valeur vraie, d'où le terme "supervisé".

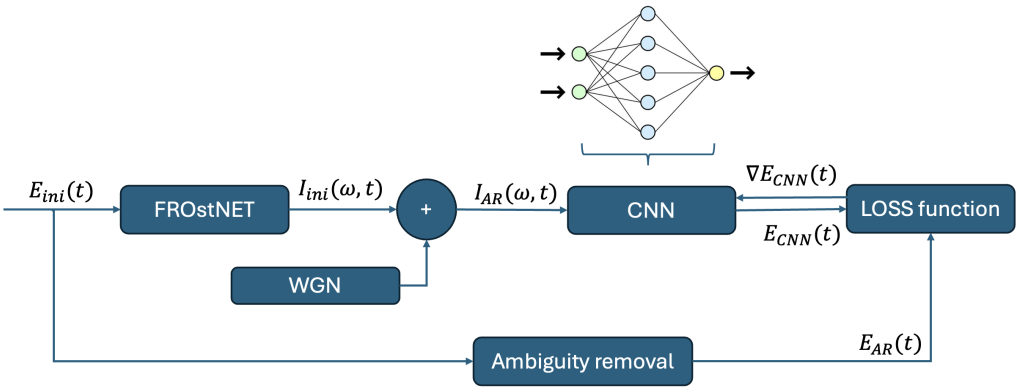


FIGURE 9 – Schéma de l'entraînement supervisé du modèle CNN

Je ne vais pas m'attarder sur les aspects techniques détaillés, car même si j'ai pu y participer, ceux-ci seront mieux résumés et scientifiquement plus rigoureux dans la thèse de maîtrise de Sydney.

En revanche, je vais résumer brièvement les étapes de son algorithme et mon apport à chaque étape. Après avoir discuté avec Sydney Lepard, mon rôle était d'apporter des idées nouvelles ainsi que de créer un dépôt Git propre répertoriant les codes du prétraitement et des modèles (https://github.com/Enzo77360/DeepFROSt).

4.1 Génération des données

J'ai d'abord eu accès aux codes de Gaëtan Jargot permettant la génération aléatoire des impulsions d'entrée.

4.1.1 Architecture

Les données utilisées pour l'entraînement du réseau de neurones sont générées par une classe Python qui modélise la création d'impulsions lumineuses et la construction de leurs spectrogrammes. Le processus commence par la création d'une impulsion dans le domaine temporel, définie par des paramètres tels que la durée de l'impulsion, la longueur d'onde centrale et l'énergie. Cette impulsion peut ensuite être transformée en une impulsion chirpée, où un coefficient de phase est appliqué pour simuler une dispersion :

Listing 1 – Création d'une impulsion chirpée

```
self.phase_coeff = np.polyval(self.phase_coeff,
                              range(-self.N_points // 2, self.N_points // 2))
self.complex_Ew_field *= np.exp(1j * self.phase_w)
```

Les données d'entraînement, ou labels, sont obtenues en calculant le champ électrique associé et en générant un spectrogramme via la transformée de Fourier. Les spectrogrammes sont ensuite normalisés et organisés pour être utilisés comme entrées dans le réseau de neurones :

Listing 2 – Génération du spectrogramme

```
self.spectrogram_complex[:, i] = np.fft.fftshift(np.fft.fft(
    np.fft.ifftshift(switch_shift * self.complex_Et_field)))
```

Enfin, des fonctions spécifiques sont incluses pour organiser, sauvegarder et charger ces données, garantissant ainsi un flux de travail fluide pour l'entraînement du modèle.

4.1.2 FROStNET

Le modèle `FROStNET`, implémenté en TensorFlow, se compose de deux parties principales. La première est la génération de spectrogrammes, où la transformation de Fourier (FFT) est appliquée sur le signal électrique dans le domaine temporel. Les composantes réelles et imaginaires du signal sont d'abord séparées, puis traitées individuellement par une couche personnalisée `FFT()` avant d'être concaténées pour former un spectrogramme complet.

```
# Application de la FFT et concat nation des resultats
real_fft , imag_fft = self.fft([real_multiply , imag_multiply])
real_spec = self.concat_layer(real_slices)
```

La deuxième partie concerne le post-traitement du spectrogramme généré. Cela inclut des étapes de normalisation, de recadrage et d'ajout de bruit gaussien. Ces opérations permettent d'affiner le spectrogramme pour le rendre compatible avec les exigences de l'entraînement du réseau de neurones.

```
# Ajout de bruit gaussien pour augmenter les donnees
spec_noise = self.noise_layer(spec_crop)
```

J'ai ainsi pu générer un dataset d'environ 2^{10} impulsions en format `npy`, utilisable pour entraîner le modèle.

4.2 Les différents modèles

Je me suis alors renseigné sur les modèles développés par Sydney Lepard.

4.2.1 CNN1 Single/Dual output

Ce modèle, assez simple au premier abord, permet une bonne compréhension du problème, notamment de la différence entre les sorties simples et doubles.

Le modèle `CNN1_SO` est un réseau de neurones convolutionnel simple avec une seule sortie. Il se compose de trois couches convolutives, suivies d'une couche de `Dropout` pour la régularisation, d'une couche de `Flatten` pour aplatir les caractéristiques extraites, et de deux couches denses (`Dense`) pour produire la sortie finale.

Listing 3 – Exemple de couches convolutives dans CNN1_SO

```
self.conv1 = tf.keras.layers.Conv2D(32, kernel_size=(4,4),
             strides=(2,2), padding='same', activation='relu')
self.fc2 = tf.keras.layers.Dense(128, activation='linear')
```

En revanche, le modèle `CNN1_DO` étend cette architecture en utilisant deux branches de couches denses à la fin, ce qui permet de générer deux sorties indépendantes à partir des mêmes couches convolutionnelles de base. Ce modèle est particulièrement utile lorsqu'il est nécessaire de prédire simultanément deux ensembles de données liés mais distincts.

Listing 4 – Exemple de branches pour sorties doubles dans CNN1_DO

```
branch1_output = self.output1(self.branch1(x))
branch2_output = self.output2(self.branch2(x))
```

La principale différence entre `CNN1_SO` et `CNN1_DO` réside dans le fait que `CNN1_SO` produit une seule sortie, tandis que `CNN1_DO` produit deux sorties, chacune à partir de branches séparées mais partant d'une base commune.

Ces sorties correspondent à la partie réelle et imaginaire de l'impulsion, permettant de récupérer l'intégralité des propriétés de cette impulsion. À ce stade, certaines questions se sont posées : pourquoi ne pas plutôt utiliser l'amplitude et la phase ? Ou pourquoi ne pas envoyer une matrice de deux lignes pour la partie réelle et la partie imaginaire respectivement ?

Bien que ces idées semblent intéressantes, l'expérience a montré qu'elles n'étaient pas plus performantes et, dans la plupart des cas, étaient moins efficaces.

4.2.2 CNN2 Single/Dual output

En fait, l'architecture du modèle a joué un rôle clé, et c'est pourquoi le modèle `CNN2` a été développé. Je ne répéterai pas la différence entre les sorties simples et doubles, car il s'agit de la même chose, mais je vais détailler cette nouvelle architecture.

Le modèle `CNN2_DO` est une version avancée du modèle à sorties doubles. Il introduit une couche de convolution multirésolutionnelle personnalisée appelée `Multires`, capable de capturer des caractéristiques à différentes échelles de résolution. Cette couche est composée de plusieurs filtres de convolution de tailles variées, ce qui permet d'extraire des informations pertinentes à différentes échelles spatiales. Après la concaténation des résultats, une couche de convolution supplémentaire est appliquée pour combiner ces caractéristiques multiéchelles.

Listing 5 – Exemple de la création de couches multirésolutionnelles dans CNN2_DO

```
self.filters_layers = [tf.keras.layers.Conv2D(filters=self.num_filters,
    kernel_size=(size, size), strides=(stride, stride), padding='valid',
    activation='relu') for size in self.kernel_sizes]
outputs = self.concat(self.x)
```

Comme pour `CNN1_DO`, `CNN2_DO` se termine par deux branches distinctes de couches entièrement connectées (`Dense`) pour produire deux sorties séparées. Cependant, grâce à sa capacité à traiter des résolutions multiples, `CNN2_DO` est mieux équipé pour capturer des caractéristiques complexes dans les données d'entrée, ce qui améliore la précision des prédictions pour des tâches nécessitant une compréhension plus fine des détails à différentes échelles.

Listing 6 – Exemple des branches de sorties dans CNN2_DO

```
branch1_output = self.output1(self.branch1(output))
branch2_output = self.output2(self.branch2(output))
```

Le modèle `CNN2_DO` surpasse `CNN1_DO` grâce à sa capacité à extraire des caractéristiques plus riches et complexes à travers ses filtres multirésolutionnels. En incorporant des convolutions avec des tailles de noyaux variées, `CNN2_DO` permet une meilleure modélisation des données à différentes échelles, ce qui se traduit par une performance améliorée, particulièrement pour des tâches où la résolution des détails locaux est critique.

4.3 Entraînement du modèle et modélisation

Une fois que le dataset et le modèle sont créés, il est essentiel de procéder à l'entraînement du modèle. Cette étape consiste à optimiser les poids du modèle afin qu'il puisse faire des prédictions précises sur de nouvelles données. La fonction `train_model` [4] présentée ci-dessous illustre le processus d'entraînement, en tenant compte des différents types de sortie du modèle.

Listing 7 – Entraînement du modèle

```
def train_model(self, model, output_type="single"):
    reduce_lr = ReduceLROnPlateau(monitor='val_loss',
                factor=0.2, patience=5, min_lr=0.0001)
    self.model = model  # Set the model
```

Pour commencer, un rappel des paramètres d'entraînement est effectué via la création d'un callback `ReduceLROnPlateau`, qui ajuste dynamiquement le taux d'apprentissage si la perte de validation ne s'améliore pas pendant un certain nombre d'époques.

Listing 8 – Entraînement du modèle avec sortie simple

```
if output_type == "single":
    self.model.compile(optimizer=tf.keras.optimizers.Adam(),
                       loss=tf.keras.losses.MeanAbsoluteError(),
                       metrics=['mae'])
    self.history = self.model.fit(
        self.data_train,  # Training data
        self.label_train,  # Training labels
        validation_split=0.1,  # Use 10% of the training data for validation
        batch_size=32,  # Batch size
        epochs=11,  # Number of epochs
        verbose=1  # Verbosity level
    )
```

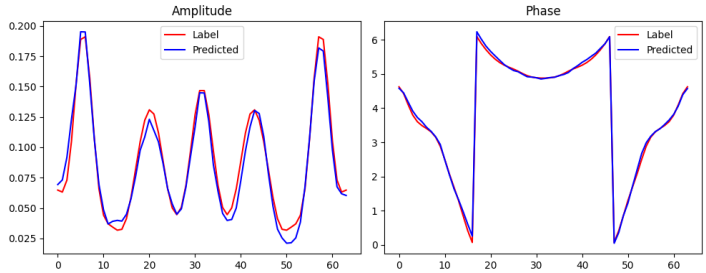
Pour un modèle avec une seule sortie, le modèle est compilé avec l'optimiseur `Adam` et une fonction de perte d'erreur absolue moyenne (`MeanAbsoluteError`). L'entraînement est réalisé avec une séparation de 10% des données pour la validation, un batch de taille 32, et sur 11 époques. L'historique de l'entraînement est sauvegardé pour évaluer la performance du modèle.

Listing 9 – Entraînement du modèle avec sorties multiples

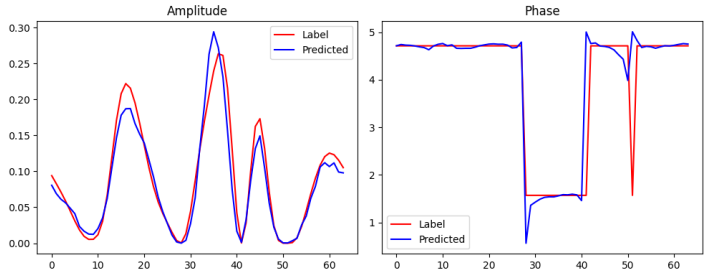
```
else:
    self.model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
        loss=[tf.keras.losses.MeanSquaredError(),
        tf.keras.losses.MeanAbsoluteError()],
        metrics=(['mae', 'mae']))  # Compile the model for multiple outputs

    self.history = self.model.fit(
        self.data_train, [self.real_train, self.imag_train],
        validation_split=0.1,  # Use 10% of the training data for validation
        batch_size=32,  # Batch size
        epochs=11,  # Number of epochs
        callbacks=[reduce_lr], # en suspens pour le moment
        verbose=1  # Verbosity level
    )
```

Pour les modèles avec plusieurs sorties, tels que les modèles de sortie double, le modèle est compilé avec une fonction de perte composée de l'erreur quadratique moyenne (`MeanSquaredError`) et de l'erreur absolue moyenne (`MeanAbsoluteError`). Le modèle est également configuré pour utiliser le callback `ReduceLROnPlateau` pour ajuster le taux d'apprentissage pendant l'entraînement. Les données d'entraînement et les étiquettes correspondantes pour les sorties réelles et imaginaires sont fournies, avec une séparation de 10% pour la validation.

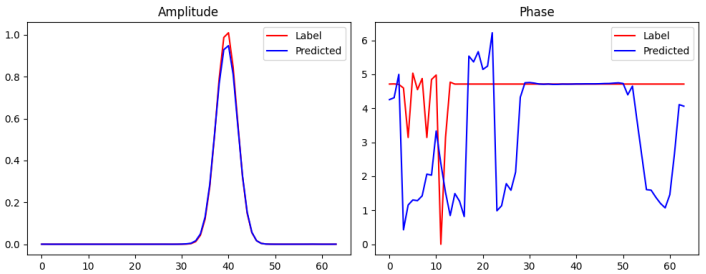
C'est avec ces différents programmes que nous avons obtenu des résultats très encourageants :



(a)



(b)



(c)

FIGURE 10 – (a), (b), (c) ; Affichage graphique des résultats pour le modèle CNN2 pour 3 impulsions d'entrée aléatoire présentant une erreur mean absolute error (mae) de l'ordre de 10%

J'ai ainsi participé activement au projet avec Sydney et approfondi ma compréhension des réseaux de neurones. Cependant, ce type de projet informatique ne nécessite pas toujours l'apport de deux personnes. C'est pourquoi, après avoir discuté avec Gaëtan, j'ai décidé de rester dans la salle de Sydney pour continuer à participer à son projet en discutant, mais aussi de me lancer dans une toute nouvelle tâche, utilisant un nouveau langage informatique : Julia.

5 Utilisation de Luna pour la Modélisation et la Conception d'une IA

L'objectif principal de cette partie était de tester les connaissances acquises sur la modélisation de phénomènes physiques à l'aide de l'IA. Je me suis donc intéressé à une bibliothèque couramment utilisée dans les laboratoires pour simuler des propagations : la bibliothèque Luna.

5.1 Présentation de Luna

Luna (https://github.com/LupoLab/Luna.jl) est une plateforme flexible dédiée à la simulation des dynamiques optiques non linéaires, que ce soit dans des guides d'ondes, tels que les fibres optiques, ou dans des géométries en espace libre. La simulation repose principalement sur l'équation de propagation des impulsions unidirectionnelle (UPPE) et ses formes approchées, telles que l'équation de Schrödinger non linéaire généralisée (GNLSE). Point intéressant pour ma compréhension, cette plateforme est codée en Julia, un langage de programmation émergent offrant une meilleure puissance de calcul que Python, tout en étant plus accessible que le C++.

Luna apparaît donc aujourd'hui comme un outil puissant permettant des simulations avancées des dynamiques optiques non linéaires, ce qui explique pourquoi cette plateforme est utilisée à l'INRS.

En suivant les instructions du README de ce projet, nous comprenons que notre intérêt se portera sur la fonction `prop_capillary`, qui permet d'effectuer une simple simulation de la propagation d'une impulsion ultrarapide dans une fibre capillaire creuse remplie de gaz.

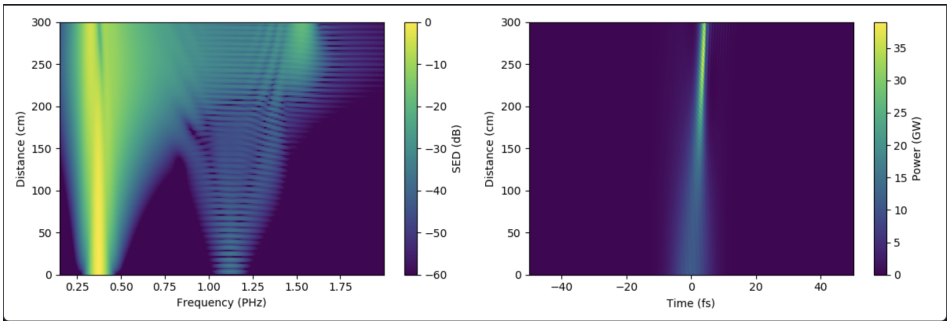


FIGURE 11 – Simulation sous Julia de la fonction prop-capillary pour une fibre capillaire creuse de 3 mètres de long avec un rayon de cœur de 125 μm, remplie de 1 bar de gaz hélium, et des impulsions centrées à 800 nm de longueur d'onde, avec 120 μJ d'énergie et 10 fs de durée, et en considérant une grille de fréquences allant de 120 nm à 4 μm avec une fenêtre temporelle de 1 ps

5.1.1 Situation Problème

La génération de ce genre de propagation nécessite un temps de calcul considérable. En effet, ce projet a été codé en Julia plutôt qu'en Python, et dès que les valeurs d'entrée commencent à varier, bien que le code fonctionne toujours, les temps de calcul augmentent significativement parfois allant jusqu'à plusieurs minutes pour une seule propagation. Une des solutions possibles est donc l'implémentation d'une IA capable d'anticiper presque instantanément la forme de l'impulsion dans une fibre à chaque étape. Plus précisément, l'idée est de créer un modèle de réseau de neurones qui éviterait à la machine de devoir résoudre mathématiquement l'UPPE à chaque étape.

Dans un premier temps, la question s'est posée de tenter de créer ce réseau de neurones directement en Julia pour rester dans le même langage. Cependant, après réflexion, il est apparu que Python est bien plus développé dans ce domaine. Il semble donc plus judicieux de générer les données en Julia, mais de les exploiter en Python.

5.2 Génération du Dataset

Comme pour tout modèle d'IA, des données sont nécessaires pour l'entraîner. C'est sûrement la composante la plus importante, car plus le jeu de données est important et de qualité, moins l'architecture du modèle a besoin d'être travaillée.

Comme expliqué précédemment, la génération du dataset sera réalisée en utilisant le langage Julia et la bibliothèque Luna pour simuler la propagation des impulsions laser à travers une fibre creuse remplie de gaz, en variant plusieurs paramètres physiques.

Une fois cette idée en tête, il ne faut pas perdre de vue l'objectif : quelles sont les informations dont nous avons besoin ? Logiquement, il nous faut nos entrées : il faut stocker les différentes informations que nous voudrons donner à notre IA entraînée dans le futur. Il faut également les sorties de la fonction `prop_capillary`, qui vont logiquement nous donner le champ électrique E et donc l'intensité à chaque pas z le long de la fibre. Ces informations seront celles sur lesquelles notre IA sera entraînée et ce qu'elle devra prédire à l'avenir.

Concrètement, les paramètres ajustés et leurs valeurs spécifiques sont les suivants :
— **Rayon du cœur de la fibre** : 250 μm.
— **Type de gaz** : Azote (N_2).
— **Pression du gaz** : de 1 à 4 bars, par incréments de 1 bar.
— **Fenêtre temporelle de l'impulsion** : de 300 à 750 fs, par incréments de 50 fs.
— **Longueur de la fibre** : 1 et 2 mètres.
— **Énergie de l'impulsion** : de 50 à 200 μJ, par incréments de 25 μJ.
— **Longueur d'onde centrale** : 1030 nm.
— **Limites de la grille de fréquences** : de 150 nm à 4 μm.

Pour chaque combinaison de ces paramètres, une simulation a été effectuée à l'aide de la fonction `prop_capillary` de la bibliothèque Luna. Les résultats de chaque simulation incluent les spectres complexes $E(\omega)$ du champ électrique, qui ont ensuite été séparés en leurs parties réelles et imaginaires.

Ces données spectrales, accompagnées des valeurs des paramètres utilisés, ont été enregistrées dans des fichiers CSV. Chaque fichier contient à la fois les métadonnées associées à la simulation (telles que le rayon du cœur, le type de gaz, etc.) et les données spectrales correspondantes.

Nous obtenons ainsi un dataset de plusieurs centaines de propagations, balayant les paramètres que l'on peut facilement rencontrer dans les laboratoires de l'ALLS.

5.3 Modélisation avec les LSTM

Une fois le dataset généré au format CSV avec toutes les informations nécessaires, nous nous retrouvons en Python avec un simple problème de création de modèle.

À première vue, la propagation dans une fibre avec un pas dz et l'observation du spectre à une distance z donnée rappelle l'idée de "prévision temporelle", couramment appliquée en météorologie. Par exemple, on observe la température des dix dernières heures pour anticiper la température de l'heure suivante.

Ce principe bien connu fait appel à un réseau de neurones récurrent de type LSTM (Long Short-Term Memory) [5]. Une couche LSTM est un type spécial de réseau de neurones récurrents (RNN) conçu pour mieux capturer et gérer les dépendances à long terme dans les séquences de données. Contrairement aux RNN classiques, les LSTM utilisent des structures internes appelées "cellules de mémoire" pour réguler le flux d'informations à travers trois portes : la porte d'entrée, la porte de sortie et la porte d'oubli. Cela permet aux LSTM de conserver des informations pertinentes sur de longues séquences tout en atténuant le problème du gradient évanescent, ce qui les rend particulièrement efficaces pour les tâches de traitement de séquences comme la modélisation du langage, la traduction automatique et la reconnaissance vocale.

Il semblait donc intéressant d'explorer une forme de "prévision spatiale" avec ces LSTM. Le modèle LSTM que nous avons conçu est structuré comme suit :

— **Longueur de la séquence** : 10 pas de temps (`l_seq`).
— **Nombre de caractéristiques dans la séquence** : 4097 (`n_features`).
— **Nombre de caractéristiques du paramètre conditionnel** : 5 (`cond_features`).
— **Nombre d'unités LSTM** : 128 (`lstm_units`).
— **Fonction d'activation de la couche Dense** : Sigmoïde (`a_func`).

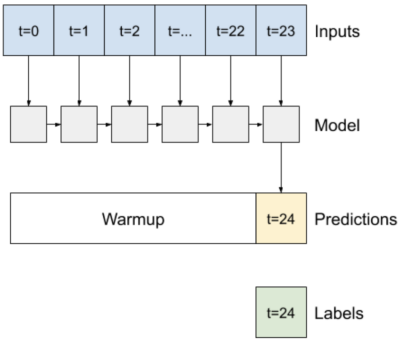


FIGURE 12 – Schéma de tensorflow representant le fonctionnement d'une couche LSTM retounant TRUE pour un input de 23 instants et une prédictions de l'instant 24

Architecture du Modèle Le modèle est construit en utilisant les bibliothèques `tensorflow` et `keras` en Python. La structure du modèle est la suivante :

1. **Entrée Séquentielle** : La séquence d'entrée est définie avec une forme de $[l_seq, n_features]$ (`i`).
2. **Paramètres Conditionnels** : Les paramètres conditionnels sont définis avec une forme de $[cond_features]$ (`c`). Ces paramètres sont étendus à la longueur de la séquence à l'aide de `RepeatVector`.
3. **Combinaison des Entrées** : Les données séquentielles et les paramètres conditionnels sont combinés en une seule entrée à l'aide de `Concatenate`.
4. **Couches LSTM** : Le modèle comprend deux couches LSTM :
   — La première couche LSTM retourne les séquences (`return_sequences=True`).
   — La deuxième couche LSTM retourne le dernier état caché (`return_sequences=False`).
5. **Couche Dense de Sortie** : La sortie de la couche LSTM est passée à une couche Dense avec une fonction d'activation sigmoïde pour générer les prédictions (`Dense(units=n_features, activation=a_func)`).

Compilation et Sauvegarde du Modèle Le modèle est compilé avec les paramètres par défaut. Après la compilation, le modèle est sauvegardé sous le nom `Single_Iw.h5`.

Résultats Concernant les résultats, j'ai réussi à faire tourner ce modèle et à simuler une propagation dans 2 mètres de fibre en utilisant les 10 centimètres précédents à chaque fois. Après optimisation des hyperparamètres avec Optuna, on observe les résultats suivants :

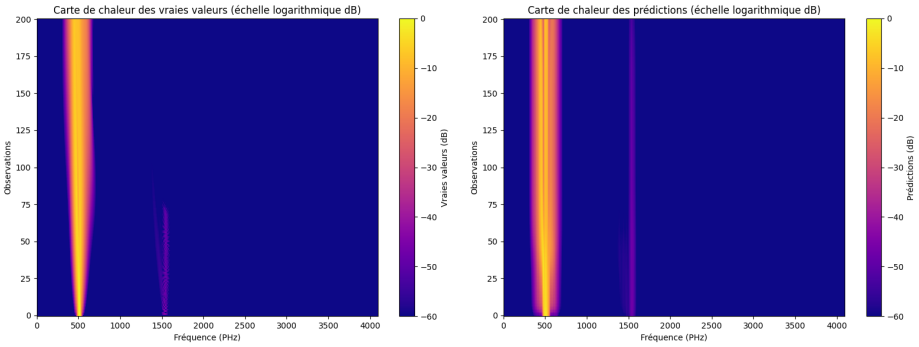


FIGURE 13 – Résultat du modèle LSTM avec le label et la prediction à gauche et à droite respectivement presentant une erreur Root Mean Squared Error (RMSE) de l'ordre de 15% sur chaque spectre

Bien que les résultats ne soient pas parfaitement alignés avec nos attentes, un problème majeur subsiste : pour anticiper le 11ème spectre, nous avons besoin des 10 spectres précédents, qui doivent être les valeurs réelles. Supposons que nous n'utilisions que les 10 premiers spectres pour calculer le 11ème, puis que nous utilisions les spectres de 1 à 11 (y compris le 11ème prédit) pour prédire le 12ème, etc. Même si l'erreur est infime sur une propagation de 2 mètres, elle devient rapidement non négligeable et le système a tendance à diverger.

Bien que l'idée soit intéressante, elle ne fonctionne pas dans ce contexte. Le point positif est que nous comprenons mieux le fonctionnement d'un LSTM et que ce modèle pourrait être utile dans un contexte spatial si suffisamment d'étapes précédentes sont disponibles et si l'on ne cherche à anticiper qu'une seule étape suivante.

Une solution consiste à se tourner vers un modèle plus classique.

5.4 Modélisation avec les Réseaux de Neurones Convolutifs (ConvNet)

Il a fallu changer notre approche : au lieu de prédire l'étape suivante, nous cherchons à reproduire les équations de Schrödinger. Pour cela, nous avons utilisé un réseau de neurones convolutif (ConvNet).

Contrairement au modèle LSTM, où la propagation était traitée étape par étape, ici nous donnons en entrée les mêmes informations que précédemment, mais cette fois avec le spectre d'entrée et le pas z concerné. Le modèle ConvNet que nous avons conçu est structuré comme suit :

— **Nombre de caractéristiques dans la séquence** : 4097 (`n_features`).
— **Nombre de caractéristiques du paramètre conditionnel** : 5 (`cond_features`).
— **Fonction d'activation des couches Dense** : Tangente hyperbolique (`a_func`).

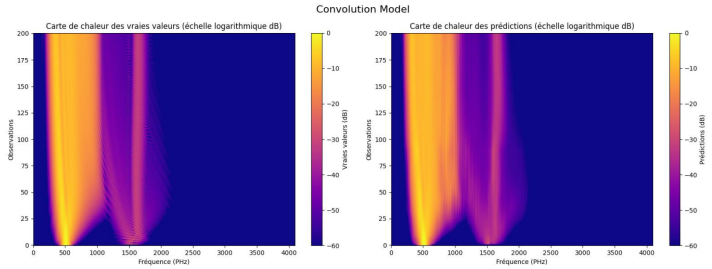
Architecture du Modèle Le modèle est construit en utilisant les bibliothèques `tensorflow` et `keras` en Python. La structure du modèle est la suivante :

1. **Définition des Entrées** :
   — **Entrée Séquentielle** : Définie avec une forme de $(n_features,)$ (`seq_input`).
   — **Paramètres Conditionnels** : Définis avec une forme de $(cond_features,)$ (`cond_input`).
2. **Concaténation des Entrées** : Les caractéristiques séquentielles sont combinées avec les paramètres conditionnels en une seule entrée à l'aide de `Concatenate` (`combined_input`).
3. **Création d'une Couche Personnalisée** : Une couche personnalisée `ExpandDimsLayer` est utilisée pour ajouter une dimension supplémentaire nécessaire pour les convolutions.
4. **Application des Convolutions** :
   — Trois couches `Conv1D` avec 128 filtres et un noyau de taille 1 sont appliquées, chacune utilisant la fonction d'activation ReLU.
   — La sortie est aplatie à l'aide de la couche `Flatten`.

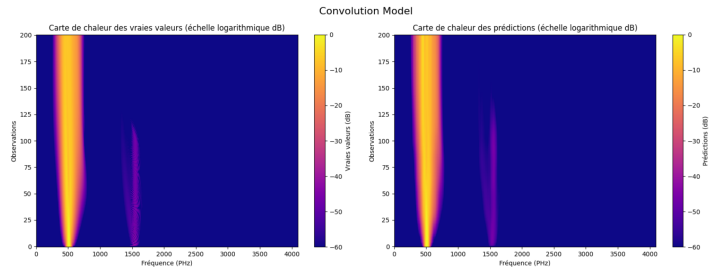
5. **Couche Dense** :
   — Deux couches Dense de 100 unités chacune utilisent la fonction d'activation tangente hyperbolique (`tanh`).
   — Une couche Dense finale génère la sortie avec une taille égale à `n_features`.

Compilation et Sauvegarde du Modèle Le modèle est compilé avec les paramètres par défaut. Après la compilation, le modèle est sauvegardé sous le nom `Conv_Model.h5`.

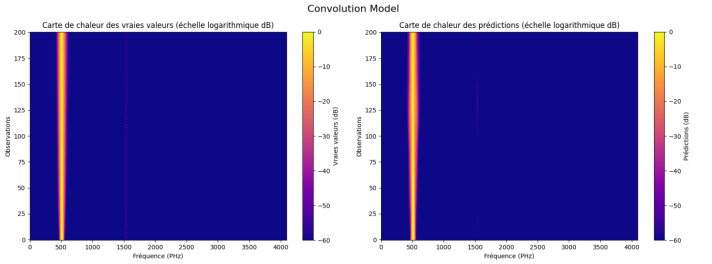
Résultats Les résultats obtenus avec ce modèle sont très satisfaisants. Je dois admettre que je suis surpris par la robustesse du modèle. L'optimisation des hyperparamètres avec Optuna semble avoir un impact principalement sur le temps de calcul, ce qui m'a poussé à orienter mon optimisation vers la vitesse d'exécution. Voici les résultats pour différentes configurations :



(a)



(b)



(c)

FIGURE 14 – (a),(b),(c), Résultat du modèle ConvNet avec le label et la prédiction à gauche et à droite respectivement présentant une erreur Root Mean Squared Error (RMSE) de l'ordre de 6% sur chaque spectre

Utilité de ce modèle Ce modèle pourrait être très utile en laboratoire pour obtenir un aperçu rapide de la propagation dans des conditions spécifiques, là où la résolution mathématique des équations à

chaque étape peut être fastidieuse. Au-delà de son utilité propre, ce travail apporte une meilleure compréhension de l'utilisation optimale des modèles LSTM et démontre la robustesse d'un modèle convolutif pour les problèmes de propagation.

Github : https://github.com/Enzo77360/Luna2.0_

6 Développement d'une Interface Polyvalente sous PyMoDAQ

Il est maintenant intéressant de prendre du recul sur le travail effectué : nous avons réussi à créer une interface et à explorer des aspects de machine learning. Cependant, il est crucial de se poser la question suivante : ce projet est-il vraiment fonctionnel ? En effet, il ne fonctionne qu'avec un spectromètre Mozza, et si des problèmes informatiques venaient à apparaître, je doute que mes codes soient suffisamment modulables et compréhensibles pour d'autres utilisateurs. C'est pour cette raison que je me suis penché cette fois qvec l'aide de Lucile Chanay sur l'utilisation de la plateforme PyMoDAQ.

6.1 Présentation de PyMoDAQ

PyMoDAQ est une plateforme de mesure et de contrôle expérimentale développée par le CNRS pour simplifier l'acquisition de données et l'automatisation des expériences. Sa conception répond à deux besoins principaux :

1. **Interface Complète pour Mesures Automatisées** : PyMoDAQ fournit une interface complète permettant d'effectuer des mesures automatisées ou d'enregistrer des données sans nécessiter le développement d'une interface utilisateur pour chaque nouvelle expérience. Cela réduit le temps et les efforts nécessaires pour configurer de nouvelles expériences, en offrant un environnement unifié et extensible pour le contrôle des instruments et la gestion des données.
2. **Outils pour la Création d'Applications Personnalisées** : En plus de ses fonctionnalités de base, PyMoDAQ offre divers outils pour faciliter la création d'applications personnalisées. Ces outils incluent des interfaces utilisateur préconstruites et des classes dédiées à des tâches spécifiques, permettant aux utilisateurs de développer rapidement des applications adaptées à leurs besoins particuliers. Cela inclut la possibilité de concevoir des interfaces graphiques personnalisées et d'ajouter des fonctionnalités spécifiques à des équipements expérimentaux.

De cette description, on peut conclure que PyMoDAQ pourrait constituer une base solide pour intégrer nos propres bibliothèques. En particulier, la classe `DAQ_Move` pour le contrôle de moteurs et `DAQ_Viewer` pour le contrôle de spectromètres pourraient être exploitées.

Objectifs L'objectif est de développer nos propres bibliothèques PyPI pour intégrer indépendamment nos moteurs et spectromètres, tout en conservant la compatibilité avec la base de code de PyMoDAQ, gérée par le CNRS. En procédant ainsi, nous pourrions créer une interface unique permettant d'ajouter facilement de nouveaux outils. En cas de problème, le débogage serait simplifié, car le code source resterait inchangé—il suffirait d'ajuster la bibliothèque spécifique, comme celle du KDC101, si nécessaire. Bien que ce projet puisse sembler ambitieux, une telle interface, si elle fonctionne, aurait un usage significatif en laboratoire.

6.2 Adaptation des Plugins pour le Moteur et les Spectromètres

6.2.1 Implémentation du Plugin pour le Moteur KDC101

Avant tout, nous avons essayé de faire fonctionner le moteur KDC101, dont nous maîtrisons maintenant correctement les DLLs sur PyMoDAQ. Le premier problème rencontré est que PyMoDAQ est conçu pour récupérer de manière dynamique le nom des plugins disponibles à partir de l'interpréteur

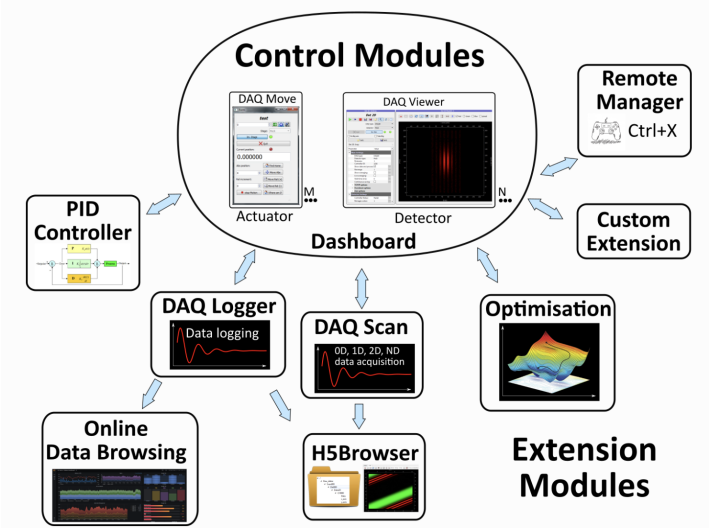


FIGURE 15 – Architecture de PyMoDAQ

Python pour une plus grande modularité. Nous ne pouvons donc pas simplement ajouter notre moteur au code ; il faut créer une bibliothèque à partir d'un des templates de PyMoDAQ pour que celui-ci reconnaisse le moteur.

L'implémentation repose sur plusieurs étapes clés ;

Le code commence par l'importation des bibliothèques nécessaires, y compris les modules spécifiques de Thorlabs pour le contrôle des moteurs et les utilitaires de PyMoDAQ. Ensuite, une classe `Kinesis` est définie pour encapsuler les opérations de base telles que la connexion, la déconnexion, le déplacement et l'arrêt du moteur. En d'autres termes, nous récupérons notre classe utilisée précédemment dans la partie 2.1. La seule différence est qu'ici nous ne faisons pas mention du nom du moteur, car chaque moteur de Thorlabs utilise les mêmes noms pour ses DLLs. De cette manière, tout autre instrument issu des codes Kinesis pourra être intégré sur cette base.

Ensuite, étant donné que nous nous intéressons au KDC101, la classe `KDC101` hérite de `Kinesis` et ajoute une vérification du numéro de série pour s'assurer que le périphérique est bien un moteur KDC101 avant de le connecter. Cette vérification n'était pas évidente précédemment, où nous appliquions nous-mêmes le numéro de série. Maintenant, le moteur est connecté de manière dynamique, ce qui est beaucoup plus modulable.

Point important : c'est uniquement cette portion de code qu'il faudrait modifier pour créer une bibliothèque pour un autre moteur.

Une fois cette première partie correctement configurée, nous utilisons un template de PyMoDAQ pour lier notre moteur à la plateforme :

```
pymodaq_plugins_template > DAQ_move_template
```

La classe `DAQ_Move_KDC101` étend `DAQ_Move_base` pour intégrer le moteur dans l'interface PyMoDAQ. Elle gère les paramètres de configuration, l'initialisation du périphérique, ainsi que les commandes de mouvement, que ce soit pour des positions absolues, relatives, ou pour le retour à la position d'origine.

Pour la mise en ligne sur PyPI, le code du plugin est empaqueté en un module Python compatible avec PyPI. Cela implique la création d'un fichier `setup.py` pour décrire le package et ses dépendances. Une fois préparé, le package est publié sur PyPI en utilisant des outils comme `twine`, ce qui rend le plugin accessible à la communauté des utilisateurs de PyMoDAQ.

De cette manière, il suffit à n'importe quel utilisateur de créer un environnement virtuel avec PyMoDAQ et d'utiliser la commande suivante pour que le moteur apparaisse dans les dispositifs possibles de `DAQ_Move` :

```
>> pip install pymodaq-plugins-KDC101
```

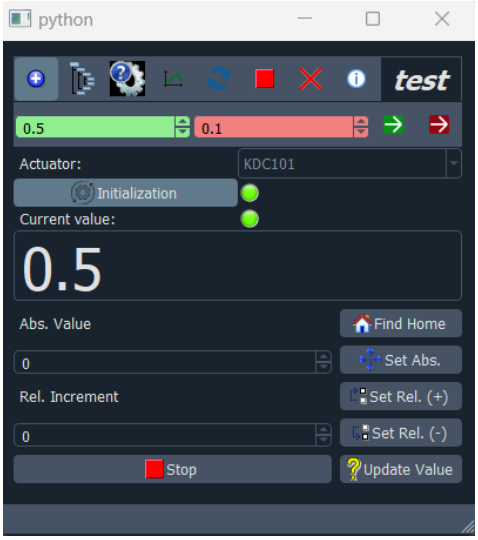


FIGURE 16 – Interface du moteur KDC101 sous PyMoDAQ

En cas de problème ou de changements quelconques, il suffit de modifier le code du moteur, d'envoyer une nouvelle version sur PyPI et de mettre à jour la bibliothèque dans l'environnement où elle est utilisée. Il est également intéressant de noter qu'il existe un `plugin_manager` dans PyMoDAQ, qui permet d'installer ces bibliothèques créées en plus du module de base. J'avais commencé par ajouter mes plugins dans ce gestionnaire, mais j'ai cessé de le faire pour éviter de le saturer avec des versions de plugins pas forcément utiles. C'est pourquoi il est important d'ajouter la bibliothèque depuis l'interpréteur et non pas depuis le `plugin_manager` au risque d'installer une ancienne version de la bibliothèque.

Github : https://github.com/Enzo77360/pymodaq_plugins_KDC101

6.2.2 Utilisation du Plugin pour les Spectromètres Ocean Optics

Cette section est importante car elle nous a permis d'explorer la possibilité de créer un plugin pour les spectromètres, ce qui n'était pas nécessairement évident en pratique. Cependant, peu de travail a été effectué puisque le plugin pour les spectromètres Ocean Optics était déjà développé et prêt à l'emploi. L'intégration de ce plugin dans PyMoDAQ a été simplifiée par l'utilisation du gestionnaire de plugins de PyMoDAQ et l'installation de la bibliothèque OmniDriver, qui fournit les DLL nécessaires pour la communication avec les spectromètres.

Pour que le plugin fonctionne correctement, les étapes suivantes ont été réalisées :

1. **Installation d'OmniDriver** : OmniDriver est le SDK fourni par Ocean Optics pour le contrôle de leurs spectromètres. Il inclut les DLL nécessaires pour l'interaction avec les dispositifs matériels. La première étape a été d'installer OmniDriver sur le système. Cette installation permet de s'assurer que les bibliothèques nécessaires sont disponibles pour la communication avec les spectromètres Ocean Optics.
2. **Configuration du Plugin dans PyMoDAQ** : Le plugin pour les spectromètres Ocean Optics, déjà développé, a été intégré dans l'environnement PyMoDAQ en utilisant le Plugin Manager.

Ce gestionnaire de plugins simplifie l'ajout et la gestion des plugins au sein de PyMoDAQ. L'utilisation du Plugin Manager a permis de charger le plugin Ocean Optics sans nécessiter de modifications supplémentaires du code.

À ce stade, nous avons atteint une étape assez gratifiante : nous pouvons faire fonctionner notre moteur et une large gamme de spectromètres sur PyMoDAQ. Cependant, nous ne comptons pas nous arrêter en si bon chemin.

Les codes étant open-source, ils ont pu servir de base pour la suite.

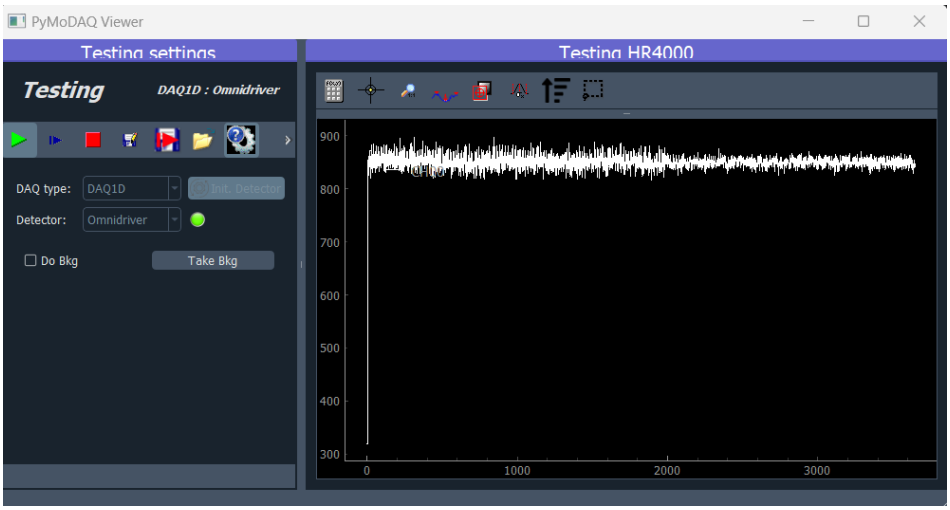


FIGURE 17 – Interface du Spectromètre OceanOptics sous PyMoDAQ

6.2.3 Implémentation du Plugin pour les Spectromètres Avantes

Passons maintenant aux types de spectromètres les plus communs en laboratoire : les spectromètres AvaSpec-2048L. Nous sommes maintenant habitués à la procédure. Nous avons installé le fichier .exe pour voir le fonctionnement de celui-ci : c'est bien plus intuitif car ces spectromètres, ayant une plage de fonctionnement dans le proche-visible, présentent quelques variations avec la lampe du plafonnier, ce qui change du simple bruit. Une fois que nous avons observé son fonctionnement, nous avons cherché à installer sa DLL, ce que nous avons réussi à faire sans trop de difficultés.

Un point intéressant est que, grâce à la popularité de ce modèle, j'ai découvert que des codes open-source permettant l'acquisition de spectre existent déjà. La bibliothèque MSL (Measurement Software Library) est déjà toute faite et permet de se connecter de manière dynamique au spectromètre et de récupérer les données d'acquisition.

Finalement, alors que nous avions besoin de créer une classe Kinesis pour le moteur, dans ce cas précis, seule l'implémentation de MSL-Equipment et la connaissance de la structure des données de sortie étaient nécessaires.

Dans les faits, l'acquisition générait des tableaux NumPy de la même manière qu'OmniDriver pour Ocean Optics, il nous suffisait donc de mimer le template d'OmniDriver dans un nouveau plugin dans la section :

```
pymodaq_plugins_template > DAQ_move_template
```

En détail, comme pour le cas du moteur, l'implémentation commence par l'importation des bibliothèques nécessaires, y compris MSL-Equipment et les modules de PyMoDAQ pour la gestion des

visualiseurs et les outils spécifiques pour la communication avec les spectromètres Avantes. La fonction `get_spectrometers_list` est utilisée pour détecter les spectromètres connectés via le SDK Avantes, en extrayant les numéros de série des périphériques disponibles.

La classe `DAQ_1DViewer_AvaSpec` est le cœur du plugin. Elle hérite de `DAQ_Viewer_base` et fournit les fonctionnalités nécessaires pour interagir avec les spectromètres Avantes. Les paramètres de configuration incluent le chemin vers la DLL Avantes, le numéro de série du spectromètre, et d'autres paramètres liés à l'acquisition des données.

Le plugin utilise la bibliothèque MSL (Measurement Software Library) pour se connecter au spectromètre et récupérer les données. La méthode `initialize_controller` configure la connexion avec le spectromètre en utilisant les informations fournies. Les méthodes `get_xaxis` et `grab_data` sont utilisées pour obtenir les longueurs d'onde et acquérir les données spectrales respectivement.

Le plugin est intégré dans DAQ-Viewer sans nécessiter de modifications itératives, en utilisant des fonctionnalités déjà disponibles pour l'affichage et la gestion des données. Ainsi, de la même manière que pour le moteur, nous obtenons une nouvelle bibliothèque utilisable avec la commande suivante :
`>> pip install pymodaq-plugins_AvaSpec`.

Github : https://github.com/Enzo77360/pymodaq_AvaSpec

6.2.4 Implémentation du Plugin pour les Spectromètres Mozza de Fastlite

Pour boucler ce projet, nous devons maintenant créer un plugin pour le spectromètre Mozza.

Le plus dur est déjà fait : nous savons utiliser correctement la bibliothèque `libmozza` de ce spectromètre pour faire l'acquisition, et nous savons, dans la mesure où les données acquises sont des tableaux NumPy, les intégrer dans le template de `DAQ_Viewer`. Cependant, cette condition a posé quelques problèmes. En effet, la fonction `get_spectrum` de Mozza ne retournait pas un tableau NumPy mais plutôt l'emplacement de la mémoire allouée au tableau de données. Il a donc fallu créer une fonction capable d'aller chercher cet emplacement et d'y récupérer nos données. La véritable difficulté était d'identifier ce problème ; une fois cette étape passée, il suffisait d'ajouter la fonction suivante dans notre template en utilisant la bibliothèque `numpy` et `ctypes` :

```
>> def convert_to_numpy_array(self, c_array, length):
    """Convert a C array to a NumPy array."""
    # Convert C array to a NumPy array
    return np.ctypeslib.as_array(c_array, shape=(length,))
```

Cette fonction est alors utilisée directement après l'appel d'acquisition de `libmozza` pour que les données puissent être traitées correctement par la suite.

À partir de là, nous obtenons le même résultat que pour le spectromètre Avantes et nous avons maintenant créé notre troisième bibliothèque utilisable avec la commande :
`>> pip install pymodaq-plugins_MozzaSpectro`.

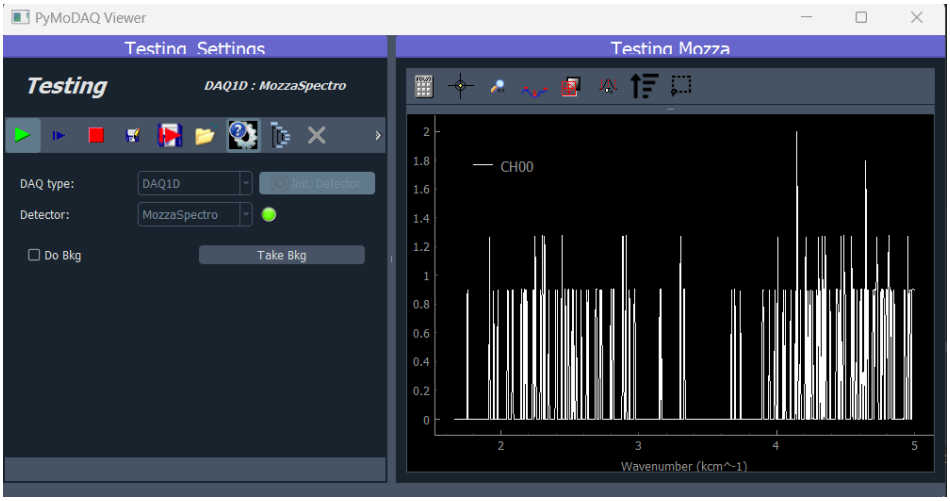


FIGURE 18 – Interface du MOZZA sous PyMoDAQ

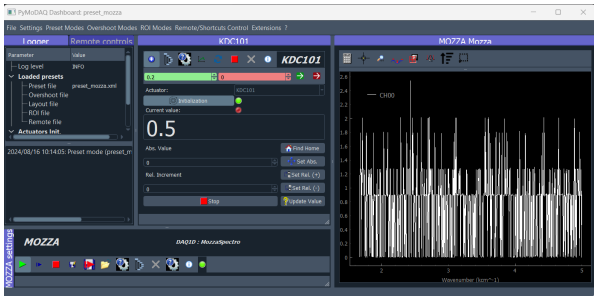
Github : https://github.com/Enzo77360/pymodaq_MozzaSpectro

6.3 Utilisation du Dashboard

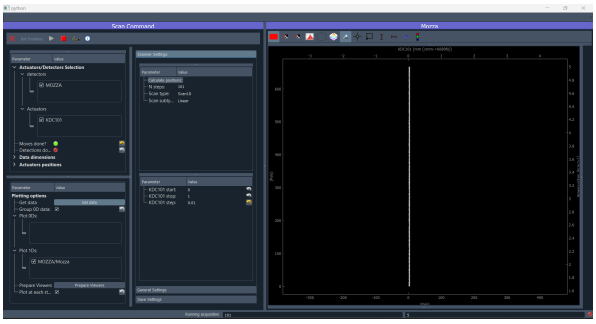
Maintenant tout est prêt ! Nous n'avons plus qu'à comprendre comment fonctionne PyMoDAQ en détail pour tirer le meilleur parti de nos bibliothèques.

Après quelques recherches, il est intéressant d'utiliser la fonction `Dashboard.py` : cette fonction nous permet de créer une interface appelant à la fois `DAQ_Move` et `DAQ_Viewer` pour le contrôle du moteur et la visualisation du spectromètre respectivement.

En résumé, nous avons créé un raccourci vers le dashboard après avoir installé PyMoDAQ et nos dépendances sur un des ordinateurs du laboratoire. Ce raccourci ouvre une interface dans laquelle nous pouvons initialiser notre moteur KDC101 et visualiser le spectromètre de notre choix. Ensuite, dans la section `Extension > do_scans`, nous pouvons programmer l'acquisition d'une trace FROSt ou Frog en sélectionnant la borne min et max du moteur ainsi que le pas, et voir la trace se construire peu à peu. Celle-ci s'enregistre automatiquement sous un format H5 dans un fichier DATA existant ou créé à la première itération. L'objectif est alors atteint, et à l'aide d'un programme très court, nous parvenons à envoyer les traces obtenues dans notre programme d'analyse de données FROSt, comme présenté dans 3.



(a)



(b)

(c)

FIGURE 19 – (a) Interface de dashboard pour l'utilisation simultané du KDC101 et du MOZZA, (b) Interface de l'option DO-SCAN permettant l'acquisition de trace FROSt/FROG, (c) Trace FROG acquise en laboratoire implémenté dans le Gui de post-traitement

7 Conclusion

7.1 Bilan

On peut finalement prendre du recul sur le travail effectué et dire que les missions de ce stage sont plutôt accomplies. Grâce à ce travail sur PyMoDAQ, nous nous assurons que notre interface pourra être utilisée et modifiée à l'avenir très facilement. Notre algorithme d'analyse de données pour le FROSt fonctionne correctement et est facile d'utilisation. Enfin, à notre échelle, nous avons pu participer au projet visant à développer une IA performante pour le traitement des données.

Il est intéressant de souligner ce que ce stage m'a apporté sur le plan personnel. N'ayant pas forcément eu l'occasion auparavant, ce stage a été ma première expérience de travail. En repensant à la personne que j'étais il y a quatre mois, les changements sont assez drastiques. Au-delà d'avoir appris à être curieux, à prendre de l'autonomie ou encore à gérer la frustration, j'ai développé des compétences techniques dans le domaine exact dans lequel j'ai envie de progresser et d'exercer mon métier plus tard, à savoir la modélisation informatique et le machine learning au sein d'un groupe de recherche. Tout cela en étant bien entouré et en venant avec plaisir chaque matin à l'INRS. Je peux donc dire que ce stage est une réussite, tant sur le plan professionnel que personnel.

7.2 Remerciements

Gaetan Jargot, François Légaré, Sydney Lepard, Lucile Chanay, Marie Ouillée

Références

[1] Elissa Haddad. Complete characterization of a yb-based opa at a high repetition rate. *Optics Express*, 31 :4534–4545, 2023.

[2] Adrien LeBlanc. Phase-matching-free pulse retrieval using frog. *Journal of Optical Physics*, 27 :1021–1028, 2019.

[3] Tom Zahavy, Alex Dikopoltsev, Daniel Moss, Gil Ilan Haham, Oren Cohen, Shie Mannor, and Mordechai Segev. Deep learning reconstruction of ultrashort pulses. *Optics Express*, 30 :1234–1245, 2022.

[4] Diederik P. Kingma and Jimmy Ba. Adam : A method for stochastic optimization. *arXiv preprint arXiv :1412.6980*, 2014. Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[5] Simone Lauria and Mohammed F. Saleh. Conditional recurrent neural networks for broad applications in nonlinear optics. *Optics Express*, 30 :567–589, 2022. SUPA, Institute of Photonics and Quantum Sciences, Heriot-Watt University, Edinburgh, UK.