Rapport de Stage :
Code de calibration, ou "post-calibration" du DRS NeoNarval

Lucas Herbert

6 août 2018

1 Introduction

Le but de ce rapport est d'expliciter les codes contenus dans le dossier calibration_methods. La réduction des données de NeoNarval passe par une étape appelé calibration. Lors de la réduction, on veut convertir le signal contenu dans un écran CCD en deux dimensions en un spectre exploitable pour des applications scientifiques, dans notre cas la détection d'exoplanètes par la méthode des vitesses radiales. Durant la réduction, on va déjà extraire l'information du spectre ordre par ordre (voir le fonctionnement de Narval sur la spécification technique du site associé). Cette information consistera en un spectre gradué seulement par des indices : une liste d'intensités représentant le spectre mais sans échelle de longueurs d'ondes associée. Ceci n'étant pas exploitable tel quel, il faut associer à ces intensité leurs longueurs d'ondes précises pour pouvoir comparer le spectre obtenu à une référence et savoir quelles raies d'absorption ou pics d'émission ont bougé et comment. Graduer le spectre en longueurs d'ondes est la mission de ce code de calibration.

Cependant, on ne part pas de zéro car le code de réduction permet déjà, en utilisant les lois de l'optique, de trouver grossièrement quelles longueurs d'ondes correspondent à telle ou telle partie du spectre réduit. Puis dans ce code, une inter-corrélation permet ensuite d'affiner cette première calibration en regardant localement de combien d'Angstroms le spectre déjà pré-calibré est décalé par rapport à une référence. Après ces deux étapes, la calibration déjà implémentée sur le DRS NeoNarval (voire DOC d'Arthur Fézard) est déjà extrêmement précise. Cependant, lors de la validation des spectres réduits par ce code, on a pu se rendre compte de quelques décalages et d'erreurs locales lors de la comparaison avec l'atlas du Thorium-Argon.

L'idée a alors été de perfectionner la calibration en utilisant les résultats des codes de validation. Pour cela, nous pouvons utiliser les listes de pics matchés après l'exécution du code matching.py. Si on sait en effet pour un certain nombre de pics par ordre quelles devraient être leurs longueurs d'ondes réelles (car ce sont celles de leurs pics matchés dans l'atlas), on peut interpoler entre ces différents pics (dés lors utilisés comme référence de

calibration) pour trouver, le long de l'ordre, une loi de conversion entre indice et longueur d'onde. On entrera ensuite la liste des indices de [0] (ou [N fois la longueur d'un ordre] qui est constante) à [longueur de l'ordre -1] (ou [N+1 fois la longueur d'un ordre - 1]) pour déterminer grâce à cette interpolation la liste associée de longueurs d'onde, nous donnant une calibration encore plus poussée que celle déjà utilisée jusque là. Voilà ce que doivent implémenter les codes qui seront présentés dans ce rapport.

2 Description des différentes parties du code

Il existe trois modules Python différents dans ce code de "recalibration" : interpolated_conversion, itered_calibration, et chelli_shift (auxquels s'ajoutent des versions "2D" que nous n'expliciterons probablement pas dans ce document (elles sont commentées dans le code) car elles n'ont pas abouti de manière satisfaisante). La philosophie de ces codes est la suivante : on converti les indices le long d'un ordre en longueurs d'onde en partant d'une première conversion (déjà contenue dans th_calibered.fits lors de sa réduction). Le matching nous donne des points fixes entre lesquels créer un polynôme d'interpolation qui sera notre nouvelle loi de conversion. Cela permet de recalibrer plus précisément, puis on va itérer le process en refaisant un matching sur la nouvelle interpolation. Cela nous donnera de nouveaux points fixes plus proches de la réalité (enfin supposés plus proches de la réalité en faisant toujours à notre référence qu'est l'atlas th_ar). De ce nouveau matching on pourra refaire une interpolation, puis rematcher, etc. L'itération prend fin lorsque l'erreur moyenne de calibration (shift entre les pics matchés côté drs et côté atlas) aura suffisament convergé. On arrive alors à la calibration la plus précise que l'on peut obtenir sans refaire une calibration primitive (sans refaire appel aux lois de l'optique, à une cross-correlation et à l'algorithme de chelli pour calibrer à partir de rien). Notez que tout le détail des entrées/sorties de chaque algorithme est donné en commentaire du code.

2.1 Interpolated_conversion.py

Le but de cette routine est de traduire, le long d'un ordre donné, les indices (nommés dans les codes du DRS des "arturos") en longueurs d'onde, donc en Angstroms en pratique. Le DRS donne en effet un spectre en arturos, c'est une unité provisoire qui permet de d'indexer les longueurs d'onde le long d'un ordre, de 0 à 4611 si on ne prend qu'un arturo par pixel dans la réduction. Disons ici qu'on va de [0] à [order_len] (ou de [N fois la longeur d'un ordre] à [N+1 fois la longeur d'un ordre - 1]). Le code de matching nous retourne une liste de données utiles relatives à chaque pic détecté lors de la validation : pour chaque pic il renvoie sa longueur d'onde dejà calibrée et son indice (arturo). Il faut noter que si le matching a été suffisament précis (voir code matching), on aura remplacé la longueur d'onde déjà calibrée par celle de l'atlas pour corriger localement la calibration, et si ce n'était pas le cas mais que les deux longueurs d'onde étaient déjà suffisament proches pour être matchées, on aura juste retourné celle déjà calibrée du DRS. On part donc avec N points fixes disant que $indice\_k$ va avec λ_k, il

suffit donc d'interpoler entre ces valeurs de λ_k avec un polynôme d'ordre fixé (choisi empiriquement, dans notre cas ce sera entre 2 minimum et 10 maximum) et on obtient alors une loi qui traduit les indices (arturos) en Angstroms.

2.1.1 read_fits

Cette fonction sert seulement à lire un fichier fits où la première extension contient un spectre et à en renvoyer les longueurs d'onde et les intensités sous forme de listes. Les entrées et sorties sont décrites dans le code en commentaire.

2.1.2 matching_reader

Cette fonction récupère les résultats de matching.py et les renvoie sous un format pratique pour la suite dans notre contexte d'interpolation. Elle prend en entrée le chemin vers le fichier enregistré par les fonctions de matching.py et retourne les données contenus dans ce fichier. Les données sont contenues dans un dictionnaire data dont les données sont repérées par les attributs 'Spikes_wavelengths', 'Spikes_indices' et 'Matching_gaps_between_drs_and_atlas', leurs noms étant explicites sur les données : indices des pics matchés, longueurs d'onde des pics matchés et erreurs associées.

2.1.3 Select_order

Cette fonction utilise la longueur d'un ordre et les données des indices et longueurs d'onde des pics récoltées par les codes précédents pour retourner la liste des indices et longueurs d'onde des pics de l'ordre considéré. On parcourt ainsi les indices des pics retenus et s'ils sont entre [N*order_len] et [(N+1)*order_len - 1] (où N est le numéro de l'ordre et order_len la longueur d'un ordre), on les ajoute ainsi que leurs longueurs d'onde à l'output de cette fonction. Les entrées et sorties sont décrits dans le code.

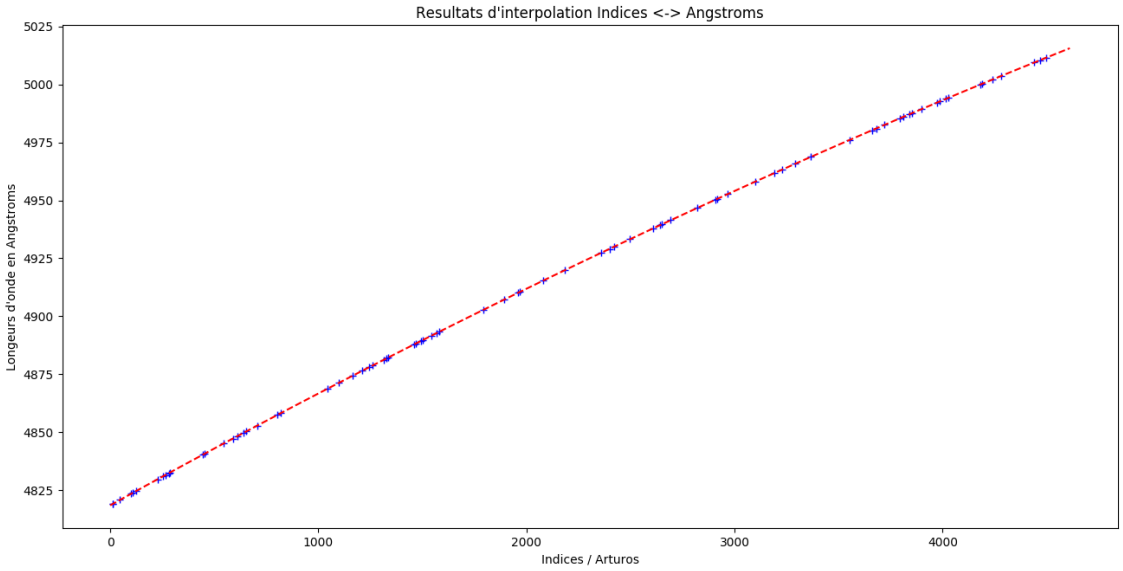
2.1.4 Polynom

Cette petite fonction retourne les valeurs prises au points donnés dans la liste d'entrée en calculant le polynome définit par la liste des coefficients donnée en entrée. Elle peut être remplacée par la fonction built-in de Python numpy.poly1d.

2.1.5 Polynomial_interpolation

Cette fonction fonctionne en utilisant comme entrées les valeurs fixes des pics : indices et longueurs d'onde et le numéro de l'ordre à interpoler. Elle fit le meilleur polynôme d'ordre 5 (paramètre modifiable dans le code sans aucune difficulté) possible en utilisant les moindres carrés (module polyfit de Python) et renvoie les coefficients du polynôme obtenu. Elle peut aussi tracer les résultats du fit pour confirmer sa validité.

FIGURE 1 – Exemple de résultats d'interpolation pour l'ordre 10 : en bleu les points fixes (qui sont des couples (indice , longeur d'onde)) et en rouge le fit polynomial calculé.



2.1.6 Arturo_convert_angstroms_order

Comme son nom l'indique, cette fonction va convertir les indices (arturos) d'un ordre en Angstroms. Pour cela, elle va évidemment utiliser la fonction polynomial_interpolation décrite juste au dessus. Elle prend en entrée les données relatives aux pics (indices et longeurs d'onde, voire détails en commentaire du code) et le numéro de l'ordre. Elle va utiliser le polynôme d'interpolation entre ces points fixes sorti par polynomial_interpolation pour convertir les indices complets de l'ordre considéré (de [N*order_len] à [(N+1)*order_len - 1]) en longeurs d'onde graduée en Angstroms. Cette fonction retourne donc ces longueurs d'onde en Angstroms ainsi que les coefficients du polynôme d'interpolation utilisé car ils pourront être utiles plus tard (cf interpolation 2D). Cette fonction est illustrée avec la figure 1.

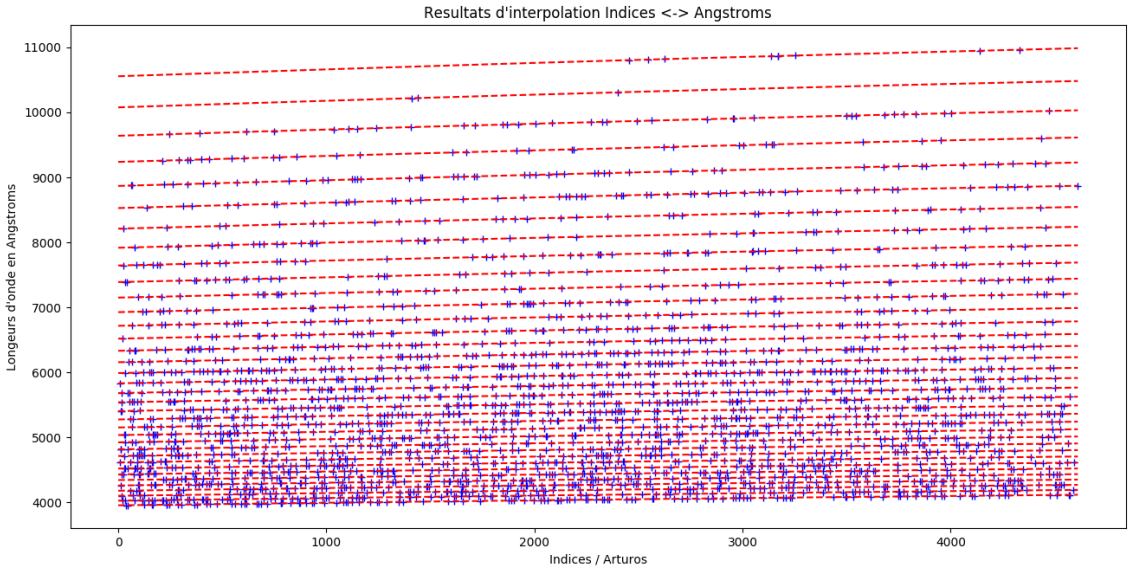
2.1.7 Arturo_convert_angstroms

Même fonction mais globale, appliquant arturo_convert_angstroms_order à tous les ordres. Cette fonction est illustrée avec la figure 2.

2.1.8 Convert_to_pickle

Cette fonction enregistre les résultats des fonctions au dessus dans un fichier au format pickle (.pkl).

FIGURE 2 – Exemple de résultats d'interpolation pour tous les ordres : en bleu les points fixes (qui sont des couples (indice , longeur d'onde)) et en rouge le fit polynomial calculé. On peut noter que cela ressemble au CCD d'un spectromètre à échelle, on voit que pour les ordres bleus, les gammes de longueurs d'ondes peuvent se chevaucher.



2.2 Itered_calibration.py

Une fois le module interpolated_conversion.py expliqué, nous pouvons entrer dans le vif du sujet de cette "post-calibration". Comme vous l'aurez déjà compris en lisant ce document, le DRS effectue déjà une calibration, mais l'idée est ici de l'améliorer en bénéficiant des résulats de la validation (expliquée dans le rapport Validation DRS). Comme expliqué précédemment, on part de la calibration du DRS, calculée grâce aux lois optiques et à une cross-correlation et affinnée par l'algrothme de Chelli. On utilise donc cette calibration pour faire notre premier matching, dont les résultats sont enregistrés dans des fichiers .pkl comme décrit précédemment. On utilise ensuite ces résultats pour interpoler et refaire une calibration grâce aux codes du module interpolated_conversion.py. Cette nouvelle calibration par interpolation polynomiale peut cependant être améliorée exactement avec la même méthode, en refaisant une validation et un matching utilisant les nouvelles longueurs d'ondes interpolées. L'intérêt est de procéder par itération en cherchant comme critère d'arrêt la minimisation de l'erreur moyenne de calibration ou l'écart-type entre valeurs matchées (DRS - atlas). C'est le but des codes de ce module, que nous allons à présent expliquer. Encore une fois, tout le détail des explications est en commentaire directement dans le code alors n'hésitez pas à aller y jeter un oeil.

On ajoute qu'en entrée de toutes les fonctions de ce module ou presque, on retrouvera les paramètres suivant : max_detection qui est le minimum de détection d'un pic d'émission du Th_Ar dans le code de validation (intervenant donc dans le matching) et precision qui est la distance minimale entre deux pics à matcher pour les considérer comme matchés. Pour comprendre ce que cela signifie, il faut se référer au rapport et aux codes de validation et de matching qui utilisent ces paramètres (tout y est expliqué).

2.2.1 Compute_order_first_conversion

La première fonction de ce module est celle qui va initier la méthode itérative pour un ordre donné en input. La première itération part de la calibration du DRS alors que les suivantes partent des calibrations successives par interpolation. Elles n'utilisent donc pas les mêmes fichiers en input. Cette fonction doit donc, en partant des résultats du matching sur la première calibration effectuée par le DRS, refaire une calibration interpolée et renvoyer les résultats (nouvelle liste de longueurs d'onde et statistiques du premier matching) de cette itération initiative. Le détail algorithmique étant décrit dans le code en commentaires, nous ne détaillons ici que la méthode sans entrer dans les précisions techniques. On commence donc par faire le matching de notre premier spectre réduit par le DRS et calibré par le DRS. On utilise cest résultats pour tracer et afficher les statistiques de ce matching : erreur de calibration moyenne, standard-deviation moyenne pour l'ordre, etc. Puis on va pouvoir, en utilisant la liste des pics matchés, refaire une conversion par interpolation (en utilisant cette fois le module interpolated_conversion.py). Les résultats de cette conversion sont alors enregistrés sous forme de pickle (fichier .pkl) pour le matching de l'itération suivante). On affiche ensuite des graphiques montrant la comparaison entre l'ancienne et la nouvelle calibration, mais cela est secondaire.

2.2.2 Compute_first_conversion

Cette fonction reprend le code de compute_order_first_coversion pour tous les ordres possibles (ceux où la validation et le matching ont donné des résultats satisfaisant donc dans notre numérotations tous sauf les deux derniers où me manque de pics matchés empêche une interpolation et une nouvelle calibration par manque de points fixes à interpoler...).

2.2.3 Compute_order_conversion

Cette fonction fait la même chose que compute_order_first_conversion mais pour n'importe quelle itération, donc ne prend pas exactement les mêmes inputs puisqu'on doit spécifier en entrée de quelle calibration (nouvelle liste de longueurs d'onde pour l'ordre considéré) on veut partir. Toutes les inputs et outputs sont clairement expliqués dans le commentaire du code donc nous les détaillons pas ici. En l'occurence si on entre la calibration originale du DRS, cette fonction revient à la même chose que compute_order_first_conversion, la rendant inutile (mais nous avons commencé par coder la version initiative donc on l'a gardé en mémoire, vous pouvez tout à fait vous en passer cela impliquant peu de modifications des codes si vous avez bien compris le fonctionnement des itérations que nous décrirons plus tard). C'est donc la fonction de référence pour une itération : elle part d'un matching, refait la conversion (et donc la calibration) en utilisant les pics matchés en points fixes entre lesquels interpoler, puis retourne les résultats (liste des nouvelles longueurs d'onde recalibrées). Remettre en input l'output de cette fonction revient littéralement à itérer, vous pouvez voir les commentaires du code pour comprendre quoi mettre en input, etc, tout y est décrit. Ce sera donc la fonction centrale de la fonction d'itération décrite plus loin dans ce rapport.

2.2.4 All_orders_iterated_conversion

Cette fonction effectue exactement la même chose que celle décrite juste au dessus mais pour tous les ordres, ce qui implique l'enregistrement de nombreux .pkl (les matchings, les nouvelles calibrations, etc). Voir le code pour plus de détail tout est encore une fois clairement commenté.

2.2.5 Order_iterated_conversion

Cette fonction est l'implémentation de la calibration itérative donc on a tant parlé. Elle prend en entrée l'ordre à itérer, les paramètre max_detection et precision déjà évoqués, le chemin du fichier où enregistrer les nouvelles longueurs d'onde et un critère de convergence stop_value. Pour stopper notre itération , on comparera à chaque fois l'écart-type des erreurs de calibration à ce critère stop_value, et lorsque cet écart-type sera suffisament petit on pourra considérer que la convergence est allé suffisament loin et qu'on peut donc arrêter d'itérer car on n'améliore plus significativement la calibration. Voici ce que nous entendons par écart-type des erreurs de calibration :

$$\sqrt{\frac{1}{N}\sum(\epsilon - \mu_\epsilon)^2)}$$

Où N est le nombre de pics matchés, les ϵ sont les erreurs de calibrations pour chaque pic (on rapelle que l'erreur de calibration d'un pic est la différence entre la longueur d'onde du pic DRS et celle du pic Atlas) et μ_ϵ est la moyenne de ces erreurs.

En pratique, la fonction suit les étapes suivantes :

- on commence par définir le fichier d'enregistrement des résultats finaux (pour plus tard)

- on effectue compute_order_first_conversion et on retient ses résultats (calibration, moyenne des erreurs de calibration, écart-type sur les erreurs de calibration) dans des variables gardées en mémoire

- on initialise la boucle d'itération en utilisant compute_order_conversion avec en entrées de la fonction la calibration obtenue juste avant avec compute_order_first_conversion. On retient les résulats dans des variables gardées en mémoire (nouvelle calibration, nouvelle moyenne des erreurs de calibration, nouvel écart-type).

- on a maintenant une base pour itérer : deux calibrations (la première obtenue avec compute_order_first_conversion et la nouvelle obtenue avec compute_order_conversion), deux moyennes (l'ancienne et la nouvelle erreur moyenne de calibration) et deux écarts-types (l'ancien/nouvel écart-type) que l'on va pouvoir comparer. Si (while en pratique) l'écart entre ancien et nouvel écart-type est supérieur à stop_value, on réutilise compute_order_conversion, les anciens calibration/moyenne/écart-type sont supprimés, les ex-nouveaux deviennent les anciens et ceux sortis de compute_order_conversion deviennent nos nouveaux calibration/moyenne/écart-type, ainsi de suite jusqu'à convergence.

- durant toutes les itérations, on ajoute à deux listes les écarts-type et erreurs moyennes calculées, et les algorithmes utilisés dans cette fonction (matching, validation, etc) vont tracer sur un même graphique toutes les erreurs de calibration ce qui nous permettra d'en suivre l'évolution. Tout cela sera enregistré sous format pickle pour pouvoir retracer ces évolutions et en discuter pour changer les valeurs max_detection, precision et stop_value ce qui permettra d'améliorer la convergence et d'optimiser la nouvelle calibration.

- à la fin on retourne les listes des erreurs moyennes et des écarts-types (qu'on a enregistrées) pour pouvoir visualiser leurs évolutions (normalement ces paramètres doivent décroître avec la convergence, signe de l'amélioration de notre "post calibration").

Remarque : on précise qu'une variable d'arrêt des itération "d'urgence" a été prévue en cas de non convergence : dans la boucle while d'itération il y a une condition sur la variable (appelée stop dans le code) comptant le nombre d'itérations qui précise que celle-ci ne doit pas dépasser tant d'itérations (valeur à fixer dans le code).

Vous pouvez vous référer aux figures 3,4 et 5 pour visualiser les évolutions des erreurs de calibration au fur et à mesure que l'on itère le processus de "post calibration".

FIGURE 3 – Illustration de l'évolution de l'erreur de calibration au fur et à mesure des itérations. Chaque doublet rouge/jaune (rouge = DRS, jaune = Atlas) a pour coordonnées : (X = longueur d'onde du pic, Y = distance entre ces deux longueurs d'onde (erreur)). On voit que l'erreur diminue au fur et à mesure des itérations, ce qui le but recherché car cela signifie que les longueurs d'ondes des pics du DRS se rapprochent de celle de l'atlas (on se rapproche de la référence).

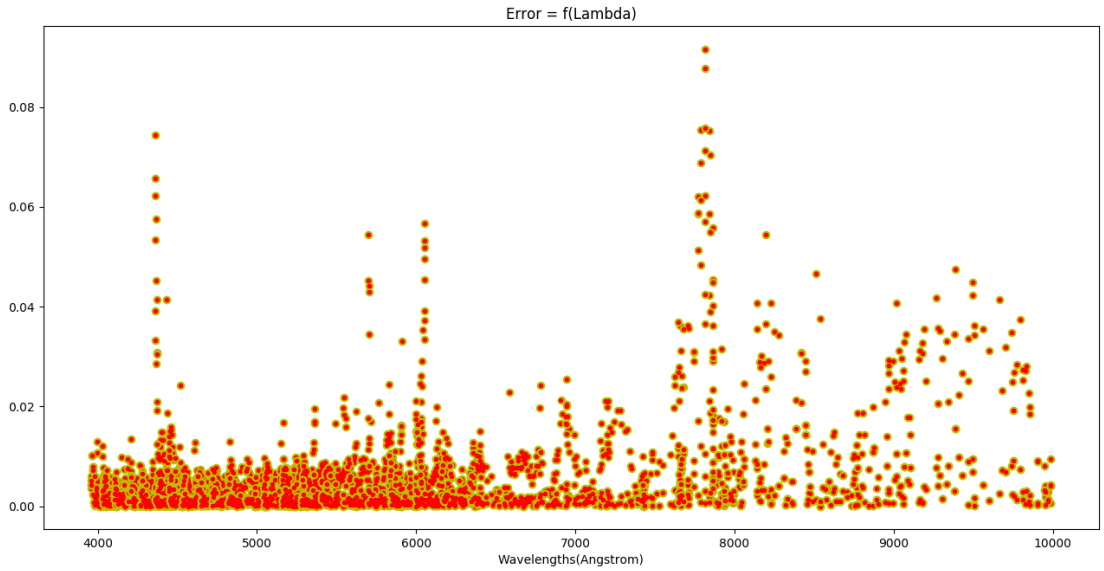


FIGURE 4 – Evolution de l'erreur de calibration pour un pic particulier (ordre 16, 20 itérations, à 5556.15 Angstroms).

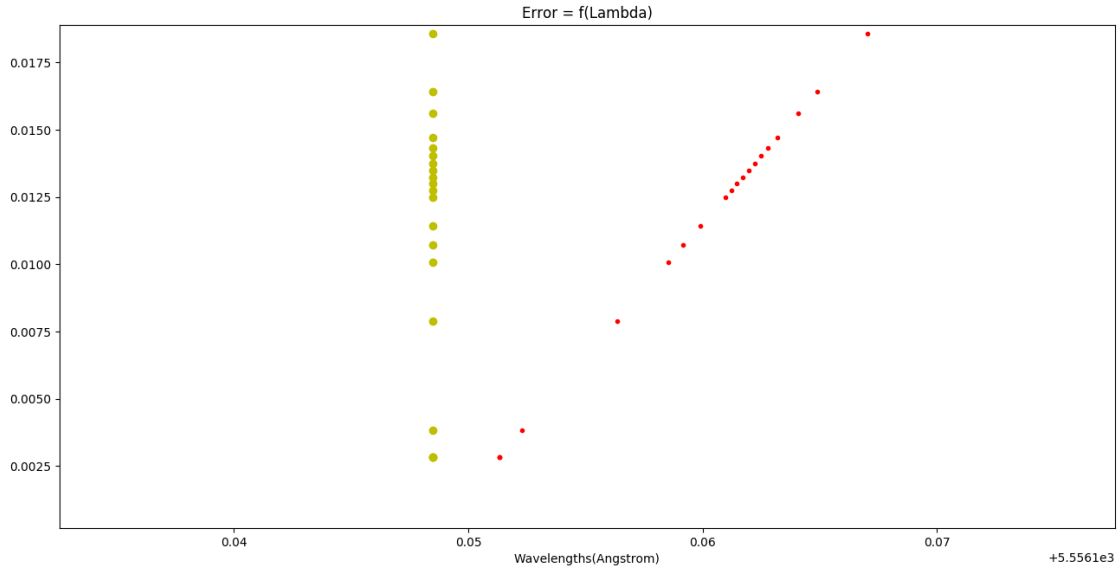
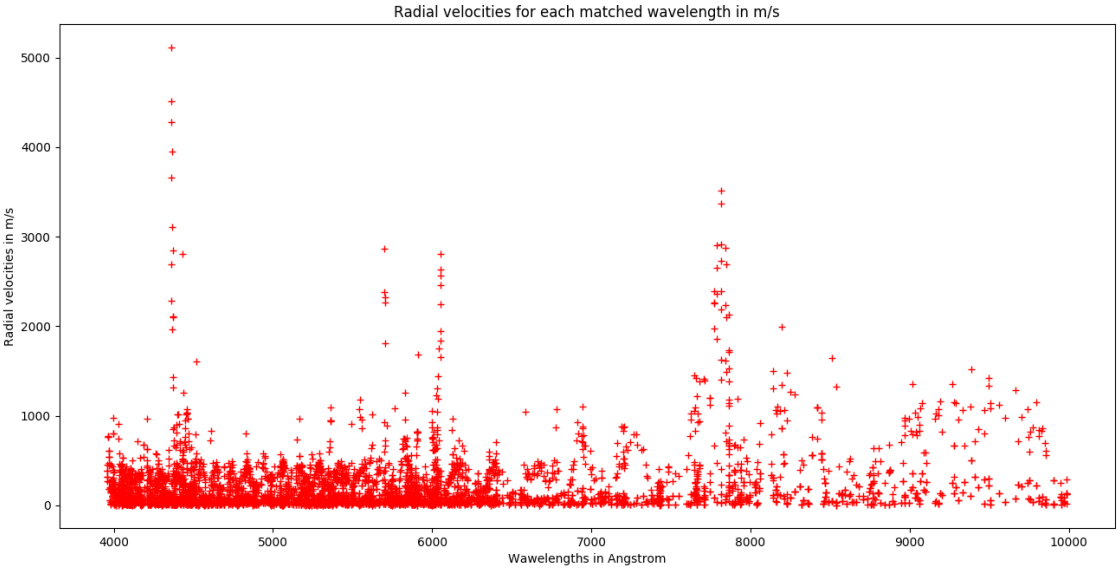


FIGURE 5 – Evolution de l'erreur de calibration mais écrite en m/s.



2.2.6 Thar_pickler

Cette fonction assez particulière mérite quelques explications supplémentaires. Elle est très simple mais les raisons de sa création le sont moins. Lors des différents matchings et calibrations successives faits avec les itérations, on a remarqué que certains pics détectés et matchés étaient visiblement mauvais : l'erreur ne diminuait pas ou restait assez grosse par rapport à l'erreur moyenne de calibration, alors nous sommes allé voir directement à la longueur d'onde en question sur le spectre ce qu'il en était. Ces pics correspondent souvent à des longueurs d'ondes "blindées", c'est à dire un endroit du CCD ou le surplus de luminosité (de photons) a entrainé une saturation locale des pixels, comme si le pixel initialement trop éclairé avait "bavé" sur les pixels autour de lui. Cela entraine sur le spectre un énorme pic très large sur lequel les pics qui se détachent sont imprécis. Ces zones sont à éviter dans la validation et le matching car on risque de faire des erreurs : mieux vaut ne pas en tenir compte. Alors comme il nous était impossible de supprimer les blindes et très difficile d'imaginer un algorithme pour les détecter et les enlever de nos listes de pics automatiquement nous avons décider de simplement retirer dans l'atlas les pics qui matchaient avec des pics blindés : nous avons créé une nouvelle référence de comparaison où on a enlevé les pics posant problème. C'est très insatisfaisant physiquement parlant mais c'était informatiquement la chose la plus simple à faire pour éviter que les blindes ne faussent nos statistiques de validation et compromettent notre nouvelle calibration.

C'est ainsi que fut codée cette fonction dont le rôle est de prendre en entrée la longueur d'onde à supprimer de l'atlas et de réenregistrer une version de l'atlas sans cette longueur d'onde que nous aurons jugée problématique. Le format d'enregistrement sera encore une fois pickle (.pkl) et le nouveau fichier de référence permettra d'ignorer les blindes. On précise qu'on n'enlève de la reférence que les pics (atlas) dont le pic matché (DRS) pose un problème visible sur le spectre (si on remarque une grande erreur de calibration entre atlas et DRS mais que le pic du DRS semble normal on ne touche à rien évidemment, car ce serait tricher pour améliorer les statistiques de validation !).

2.2.7 Autres fonctions

Il n'y a pas à proprement parler d'autres fonctions mais le reste du code (qui a été mis en format commentaire) sert à tracer de nombreuses figures, nous le laissons en l'état car cela ne vaut pas le coup d'écrire une fonction et il peut facilement être réécrit.

2.3 Chelli_shift.py

Ce module a pour but de détecter et quantifier un décalage entre deux spectres donnés en entrée. On suppose en effet que ces deux spectres viennent de la même source donc sont relativement similaires. De nombreux facteurs physiques influencent en effet l'instrument ce qui provoque un décalage progressif des longueurs d'ondes sur le ccd, ce décalage doit être régulièrement mesuré pour "recalibrer" l'appareil et être sûr que le décalage mesuré

au final est un décalage "vrai", ce qui permettra de faire des vitesses radiales comme prévu.

2.3.1 Shifter

La fonction nommée shifter utilise la transformée de Fourier et la transformée inverse pour décaler un spectre du shift donné en entrée. On notera qu'ici le shift est en indices car on manipule des listes.

2.3.2 Chelli_shift

C'est l'algorithme codé par Arthur Fézard déjà décrit dans son rapport. Il calcule le shift entre deux spectres en utilisant l'algorithme de Chelli. Cependant il ne fonctionne que sur des shifts inférieurs à 1 indice en valeur absolue d'où l'utilité de fonctions calculant le shift à l'indice près, elles sont décrites plus loin. Les entrées et sorties sont décrites dans le code.

2.3.3 Chelli_shift_v2

Comme l'algorithme de Chelli semble mieux fonctionner sur un spectre petit (une petite liste), on découpe le spectre mis en entrée pour calculer le shift avec l'algorithme de Chelli sur chaque partie du spectre. On calcule ensuite la moyenne de ces shifts, que l'on retourne : cela nous donne une estimation du shift moyen sur le spectre.

2.3.4 Find_big_shift

Comme son nom l'indique, cette fonction permet de calculer un shift grossier entre deux spectres donnés en entrée. Elle servira donc d'amorce à chelli_shift. Elle utilise pour cela les transformées de Fourier et leurs inverses, prenant le premier spectre donné en entrée et le décalant en utilisant shifter pour le comparer au deuxième spectre donné en entrée. Elle minimise un critère de moindres carrés entre le spectre décalé et le deuxième pour trouver le shift correspondant le mieux, et retourne ce shift. La précision du shift à retourner est réglable en modifiant la boucle for (nombre d'itérations). Le détail est commenté dans le code.

2.3.5 Find_shift

Cette fonction utilise find_big_shift et chelli_shift pour calculer le shift précis entre deux spectres. Elle commence par calculer le shift grossier avec find_big_shift puis utilise chelli_shift pour affiner. Puis elle reboucle chelli_shift sur lui même en l'initiant avec le shift qu'il a trouvé à l'itération précédante pour maximiser la précision. Le shift étant donné en indices, il est ensuite converti en longueurs d'onde en utilisant une calibration (cet alogrithme est pour le moment destiné à des spectres réduits par Narval, de type étoile ou Thorium-Argon, donc les longueurs d'onde ont déjà été calibrées et dont on veut affiner cette calibration).

2.3.6 Correlator

Une deuxième manière de trouver un shift grossier à un indice de décalage près est d'utiliser l'auto-corrélation entre deux spectres. Celle-ci étant implémentée en Python, on utilise directement numpy.correlate pour trouver le décalage entre les deux spectres donnés en entrée.

2.3.7 Find_shift2

C'est exactement la même fonction que find_shift mais au lieu d'utiliser find_big_shift pour initialiser la recherche avec chelli_shift, on utilise correlator. Tout le reste est décrit dans le code.

3 Utilisation pratique des codes

Comme pour les codes de validation, il faut lancer les fonctions en étant dans le bon répertoire. Pour cela, ouvrez Pyzo (ou lancez les fonctions) en étant dans le dossier Codes_Lucas_Herbert, où se situent normalement le dossier calibration_methods et le dossier validation_methods. Ensuite dans les dossier Validation_files et Calibration_files, libre à vous de placer des spectres à valider et recalibrer (en donnant une première calibration, c'est à dire une liste de longueurs d'onde de même longueur que le spectre à valider/recalibrer) et d'indiquer leurs chemins en entrée des différentes fonctions. Les fonctions d'itered_calibration referont d'elles-même le matching et le recalibration, sortant au fur et à mesure des données graphique sur la progression du matching, etc.