En bref

Infos en vrac (à réarranger)

Site Web JUICE officiel

https://www.cosmos.esa.int/web/juice

JUICE Livelink

Les présentations des différents SWT sont disponibles sur le livelink de JUCE :
https://dms.cosmos.esa.int/cs/cs?func=ll&objId=3169397&objAction=browse&viewType=1 (demander les identifiants à Laurent si besoin)

JUICE/RPWI Ground segment piipeline

Le traitement des données SCM (spectres et formes d'ondes) sera intégré au pipeline du consortium RPWI (lead. Uppsala).

Voir le plan de développement ici : https://hephaistos.lpp.polytechnique.fr/redmine/documents/182 (à mettre à jour -> demander à D. Andrews)
Gitlab: https://spis.irfu.se/rpwi/rpwi_pipeline/ (compte personnel avec adresse e-mail LPP)
Documentation: https://www.space.irfu.se/juice/rpwi_pipeline/index.html (demander les identifiants à Laurent si besoin)

Lien pour tutoriel pratique "Python Documentation Using Sphinx Autosummary" :
https://medium.datadriveninvestor.com/python-documentation-using-sphinx-f6dc87e1286b

Notes sur l'avancée du code

- actuellement, les tests de comparaison sont effectués pour un fs = 31.995 comme idl le fait de base, on obtient des résultats où la phase du kernel résultant vaut +/- 40, et on obtient pareil (à epsilon prêt) en python.
Si on force le code à utiliser fs = 32 à la place, les résultats changent, la phase dans le kernel IDL vaut +/- 1e-11 , et 0 en python (donc très proche)

-Note d'un problème en cours (qui n'est pas trop problématique mais ennuyeux quand même) : Problème d'arrondi qui provque différences entre IDL et python :

les résultats des tests de deconvo vec varient en fonction de fs, df et surtout la manière différente dont idl et python arrondissent df et f_i lors de leur creéation/manipulation Dépendant de comment on déclare df et f (f_i) en idl et de comment on le write dans le log, les résultat du test python en utilisant comme f_i celui du log vont être positifs par epsilon 1e-4 ou non.
Et je n'arrive juste pas à faire en sorte que prendre f_i de generate_freq_array en python donne des résultats positif, parceque les valeurs sont toujours différentes du f_i extrait du log.

Examples de situation :
fe = 31.9995, df = fe/float(nk), frq = findgen * df, write %23.16e (setting classique), python utilise ref_table -> fonctionne
fe = 32 -> fonctionne (logique, vu que pas besoin d'arrondi)
fe = 31.9995, df = fe/float(nk), frq = dindgen * df, write %23.16e , python utilise ref_table -> echec
(setting classique) + python utilise generate_freq_array (que ce soit round on pas) -> echec
fe = 31.9995, df = fe/float(nk) puis round, frq = findgen * df, write %23.16e (setting classique), python utilise generate_freq_array (round au même niveau) -> echec

Documentation code

Le readme.md contenu dans le code uploadé sur le github possède une explication complète de la structure du fichier IDL deconvo_vec qui contient tout ce que l'on traduit en python actuellement, et la structure du fichier python deconvo_vec équivalent et de toutes les fonctions qui en découle (en cours de construction).
Je copie une version ici (visuellement plus agréable dans github:

Documentation FR de l'avancé du portage IDL -> Python.

Actuellement on se concentre sur le portage de la fonction *mms_scm_deconvo_vec*

Cette fonction prend en entrée une waveform et des metadatas.
L'object est d'effectuer la calibration continue de la waveform en convolvant
le signal par un kernel que l'on construit au préalable.

Les différentes étapes de *mms_scm_deconvo_vec* sont :

1. (on précentre la waveform d'entrée)
2. On souhaite créer un kernel de taille n_k, on commence donc par former un "complex spectrum" **s** de base
(un array de complexes 1+0j, de taille n_k)
3. On applique à **s** la fonction *mms_scm_corgain* (même fichier)
1. On crée un array de fréquence **f** linéairement croissant de pas df = f_e / n_k (f_e = fréquence d'échantillonnage), de longueur n_k
auquel on soustrait f_e la seconde moitié, on a donc un array de fréquence allant de 0 -> fe/2 puis -fe/2 -> 0
(le format nécessaire pour la fft)
2. On calcule la réponse **c** renvoyée par *mms_corgain*
1. La fonction récupère les données dans le fichier de référence des antennes (dont on a donné le path),
qui contient, pour chaque antenne, un array de fréquence et pour chaque fréquence la réponse complexe référence
correspondante.
2. Si **f** contient des fréquences en dehors du range du fichier de référence,
on considère que la réponse de l'antenne va être calculée pour la fréquence référence la plus proche
3. On obtient la réponse de l'antenne, calculée pour chaque valeur de **f** par interpolation
des données de référence.
4. On multiplie cette réponse par la valeur absolue de la réponse du filtre dfb,
implémenté dans *mms_scm_dfb_dig_filter_resp* (qui renvoie une réponse pour chaque fréquence de **f**)
5. On multiplie cette réponse par la réponse du filtre bessel, implémenté dans *bessel_filter_resp*
(qui renvoie une réponse pour chaque fréquence de **f**)
6. On renvoie cette réponse
3. Maintenant que l'on a **c**, on divise **s** par **c** (on applique la correction au spectre d'entrée en somme)
4. La phase du terme de fréquence 0 de la réponse doit valoir 0 pour pouvoir appliquer la FFT,
on set ce terme à la valeur absolue de sa valeur complexe
4. On a maintenant un **s** qui a subit la correction de l'antenne + dfb + bessel
5. On applique un filtre passe bande par *mms_scm_filtspe* entre f_min et f_max donnés en entrée
6. On applique une transformée de fourier inverse, non encore normalisée, à **s**,
on obtient alors ce que le code appelle le kernel (qui n'est pas encore le kernel final)
7. Maintenant on prend seulement la partie réelle (sachant que l'imaginaire doit être négligeable si les calculs sont corrects)
8. On shift le kernel
9. On lui applique la fenetre de convolution (Hanning, coscub, trapezoid, etc)
10. On normalise le kernel
11. On peut maintenant effectuer la convolution (*mms_scm_fastconvol*) pour la waveform d'entrée et le kernel calculé

Dans le code python, nous avons une fonction ***deconvo_vec*** correspondant à *mms_scm_deconvo_vec* :

1. (on précentre la waveform d'entrée)
2. On crée l'array de fréquence **f** en premier (étape 3.i)
2. On crée le kernel non normalisé (-> étapes 2 à 6 sauf 3.i), en utilisant la fonction ***kernel_creation***
1. On crée le complex spectrum de base **spectrum** équivalent à **s** (étape 2)
2. On applique la fonction ***corr_gain_ant*** à **spectrum** avec **f** comme argument (étape 3.ii:iv)
1. On obtient **gain_array** (équivalent de **c**), initialement la réponse de l'antenne calculée par la fonction ***ant_resp*** (3.ii.a:c) :
1. La fonction récupère les données du fichier de référence (3.ii.a)
2. Elle règle le problème de valeur hors range (3.ii.b)
3. On effectue l'interpolation pour obtenir la réponse de l'antenne (3.ii.c)
2. On multiplie **gain_array** par la valeur absolue de la réponse du filter dfb, implémenté dans ***dfb_filter*** (3.ii.d)
3. On multiplie **gain_array** par la réponse du filter bessel, implémenté dans ***bessel_filter*** (3.ii.e)
4. On divise **spectrum** par **gain_array** (3.iii)
5. On modifie le terme de fréquence 0 (3.iv)
3. On applique une filtre passe bande à **spectrum** (***bandpass_filter***) (5)
4. On applique la transformée de fourier inverse à **spectrum** (***fft***) et obtient **kernel** (6)
3. On vérifie que la partie réelle est bien négligeable et on prend uniquement la partie réelle. (7)
4. On shift le kernel (8))
5. On applique la fenêtre de convolution choisie à **kernel** en utilisant les fonction ***conv_windows*** (9)
6. On normalise le kernel (10)

Notice Installation

Voir le readme.md

En ce qui concerne le pycharm, ce qui suit est toujours valide :

1. Installer Pycharm (J'utilise la version professional 2022.3.2 mais ça devrait fonctionner sans problème quelque soit la version https://www.jetbrains.com/fr-fr/pycharm/download/)
2. Faire un git clone du code dans le répertoire de son choix (il faut donc avoir git installé) : git clone https://stassen@hephaistos.lpp.polytechnique.fr/rhodecode/GIT_REPOSITORIES/LPP/DATA-PROCESSING/SCM-Waveforms-Calibration
3. Ouvrir Pycharm, Aller dans File -> Open -> Chercher le répertoire où se trouve le code -> Ouvrir

Installation Pylint (Pour PEP8) :

1. Pour installer le module pylint de pycharm : File -> Settings -> Plugins : chercher pylint dans la liste, et l'installer.
2. Pylint apparaît en bas du menu de gauche dans File -> Settings (si non, relancer pycharm)
3. Dans le terminal de pycharm, en étant bien avec le venv activé, taper 'pip install pylint'
4. Dans File -> Settings -> Pylint, pour le "Path to Pylint executable", normalement cela affiche Auto-detected : path/pylint -> Appuyer sur test, si test validé -> Appuyer sur Ok. Si le path n'est pas trouvé directement il faut trouver le path du pylint dans le venv du projet, et le copier là.
5. Une fois que c'est fait, en bas à côté des icônes terminal etc se trouve Pylint, le module qui permet d'analyser le PEP8 de ce que l'on souhaite.
6. Pycharm affichera également les PEP8 warnings dans le code directement.

Modifications à faire dans le code IDL (corrections d'erreurs) pour que les résultats soient les mêmes que ceux obtenus par python

1. Lors de la lecture du fichier textuel de calibration de l'antenne, le résultat est placé dans un complex array simple > il ne lit pas correctement tout les digits du fichier -> mettre dcomplexarr à la place
2. Dans mms_scm_calibration, quand on appelle mms_scm_deconvo_vec pour chaque antenne, le code passe 1 comme numéro d'antenne pour le deconvo vec de x, 3 pour Y et 2 pour Z, ce qui est une erreur puisque quand le code appelle plus loin le fichier de calibration, on a bien x=1, y=2, z=3, ce qui provoque une erreur non négligeable, les kernel de y et z sont tout simplement interverti (le seul endroit où le kernel dépend de l'antenne c'est la réponse de l'antenne, on on interverti celle-ci). > donc très important, remplacer 3 par 2 et 2 par 3 lors des appels "mms_scm_deconvo_vec, yfo, " et "mms_scm_deconvo_vec, zfo, "