27.01.2023

Amélioration d'une bibliothèque Python pour un code correcteur d'erreur

Compte rendu de projet

Réalisé par

- BEKRI Meryem
- BENAMAR Mohamed
- KANDEL Grégoire
- NOËL Yoran

Encadré par

- LEROUX Camille
- TAJAN Romain

Table des matières

I. Présentation du projet

A. Environnement de travail

Ce projet a pour but d'améliorer un système de communication et le rendre plus efficace et plus robuste. Pour ceci on choisit d'utiliser deux bibliothèques **Aff3ct** et **Pyaf** .

1. Présentation des bibliothèques Aff3ct et pyaf:

La bibliothèque **Aff3ct** est une bibliothèque opensource pour les communications numériques qui permet la correction d'erreurs directes (FEC: Forward Error Correct Correction). Elle est décrite en C++ et supporte les codes turbo , polaire et ldpc. Elle comprend plusieurs blocs et classes afin d'exécuter une chaîne de communication numérique. Les blocs sont optimisés de telle façon à ce que l'exécution prenne un minimum de temps sur un processeur monocœur. Les blocs sont sous forme de modules et de tasks qui communiquent via des sockets (de type input , output ou input/output).

La bibliothèque **py_aff3ct** permet de passer par Python3 afin d'utiliser Aff3ct pour simuler et tester des systèmes de codage et de modulation pour des communications parfaitement adaptatives sur des canaux de transmission imparfaits.

2. Schéma initial du projet

Le code source initial fourni par nos encadrants peut être schématisé sous cette forme:

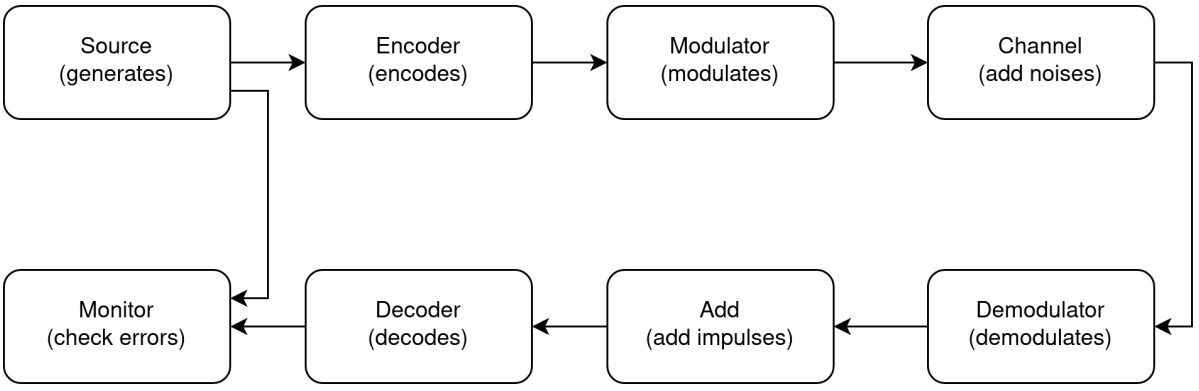

Figure 1 - Chaîne de communication numérique du programme

Dans un premier temps, l'information à transmettre par le canal émetteur. Elle est générée à l'aide du bloc **Source.** Le bloc **Encoder** transforme le message afin de le rendre moins susceptible aux erreurs et le bloc **modulateur** le démodule. Finalement, le module **Channel** altère l'information en y introduisant du bruit.

Dans le canal récepteur, les blocs **Démodulator** et **Decoder** permettent d'effectuer les opérations inverses en décodant et démodulant le signal afin de récupérer l'information produite par la source. Le bloc **Add** permet d'altérer le vecteur d'information , il sera détaillé dans la partie qui suit. Finalement, le bloc **Monitor** permet d'évaluer la chaîne de communication numérique en comparant l'information initiale et l'information finale obtenue par décodage.

3. Résultat initial du code source

En préambule, le code source fourni permet de générer une image de taille **160 pixels** x **160 pixels** représentée ci-dessous:

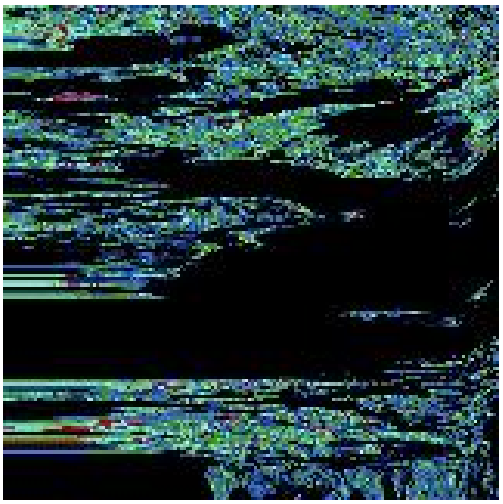

Figure 2 - Image générée par le code source fourni

La couleur de chaque pixel est le résultat du décodage d'un vecteur de taille N auquel on vient changer la valeur de deux éléments d'indices **ix_x** et **ix_y (cf. Figure 3 ci-dessous).**

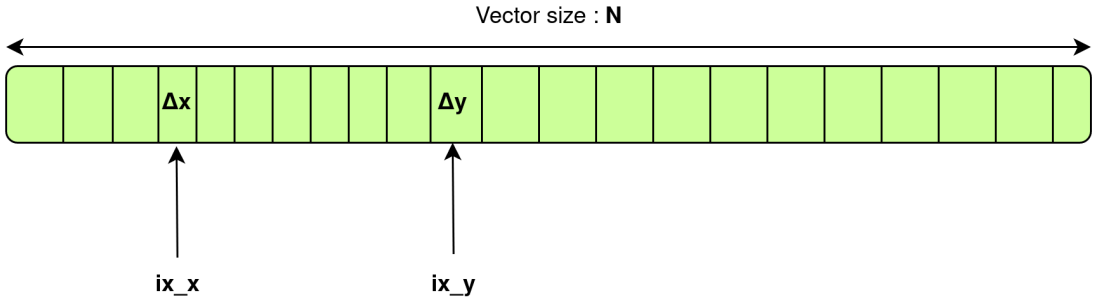

Figure 3 - vecteur "noisy vector"

Suite au décodage, si le vecteur a bien été décodé, le pixels correspondant au coordonnées (delta(x),delta(y)) prend une couleur noire. Dans le cas contraire, la couleur du pixel dépendra de la différence entre le vecteur initial **(Noisy Vector)** et le vecteur décodé. Il est à noter que la valeur de delta(x) et delta(y) varie entre -4 et 4.

Ainsi l'image est généré pixel par pixel en modifiant le vecteur **noisy vector 160** x **160** fois.

B. Objectif : code correcteur à améliorer

Le programme fourni est donc fonctionnel. Il permet de générer une image *pseudo-aléatoire* en fonction des caractéristiques d'un vecteur de données et du choix des éléments de la chaîne de communication numérique (encodeur et décodeur). Cependant, le temps d'exécution de ce programme reste long, de l'ordre de plusieurs secondes sur un ordinateur portable personnel. Ainsi, l'objectif est d'améliorer ce code correcteur afin de le rendre plus performant en termes de temps d'exécution.

Pour cela, nous utilisons la bibliothèque *pyaf* qui permet de lier du code C++ au code Python écrit avec la bibliothèque *py_aff3ct*. L'utilisation du langage C++ possède l'avantage d'être un langage compilé, et donc bien plus rapide que Python qui est un langage interprété. Dans un second temps, l'utilisation du C++ avec *pyaf* permettra de mettre en œuvre la technique du *pipelining*, ce qui pourra grandement améliorer les performances du programme.

Par ailleurs, pour rendre l'utilisation du programme plus intuitive, un objectif annexe est de mettre en place une interface graphique.

II. Réorganisation du code en deux séquences

On choisit de diviser la première séquence en deux sous séquences pour une bonne organisation du travail. Ça nous a permis de mieux comprendre le code existant, comme on a pu tester chaque module a part.

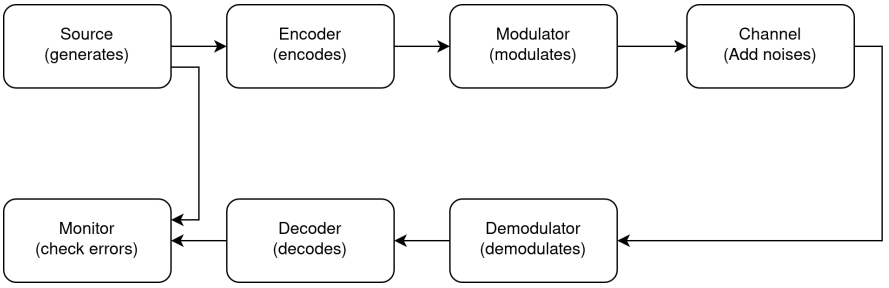

Figure 4 : Première séquence

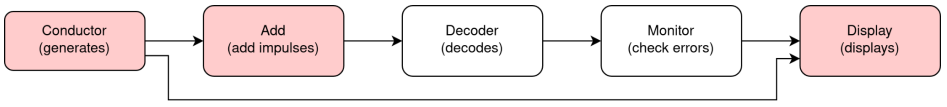

Figure 5 : Deuxième séquence

La première séquence génère un vecteur bruité que la deuxième récupère et modifie pour générer l'image. Sur la deuxième ***Figure 5***, trois nouveaux modules ont été ajoutés, ce

qui définit le travail que le groupe a été amené à faire. Certains blocs ont été transcodés du Python en C++ pour des raisons de performances.

III. Présentation des modules

A. Le module Conductor

1. Explication du module

Le module *Conductor* est le premier module de la deuxième séquence. La traduction française de ce module est littéralement “chef d’orchestre”; et c’est exactement le rôle de ce module dans la séquence. En effet, les modules *Add* et *Display* sont dirigés par le module *Conductor*. Le schéma comportant les sockets sortantes du *Conductor* est présenté ci-dessous (***Figure 6***).

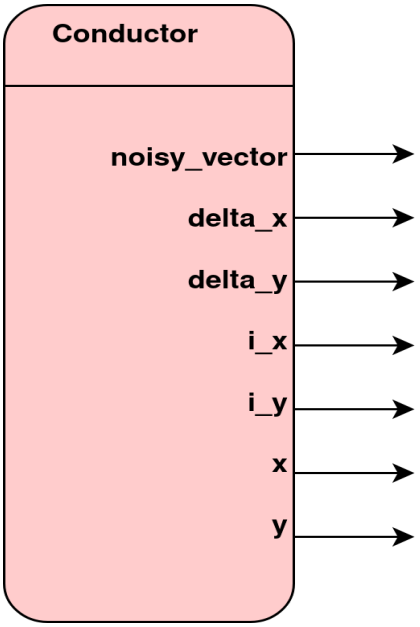

Figure 6 - Module Conductor

Sur la ***Figure 6,*** ce module possède sept sockets sortantes, cinq étant dirigées vers le module *Add* et deux vers le module *Display*.

Un schéma plus illustré est présenté ci-dessous (***Figure 7***).

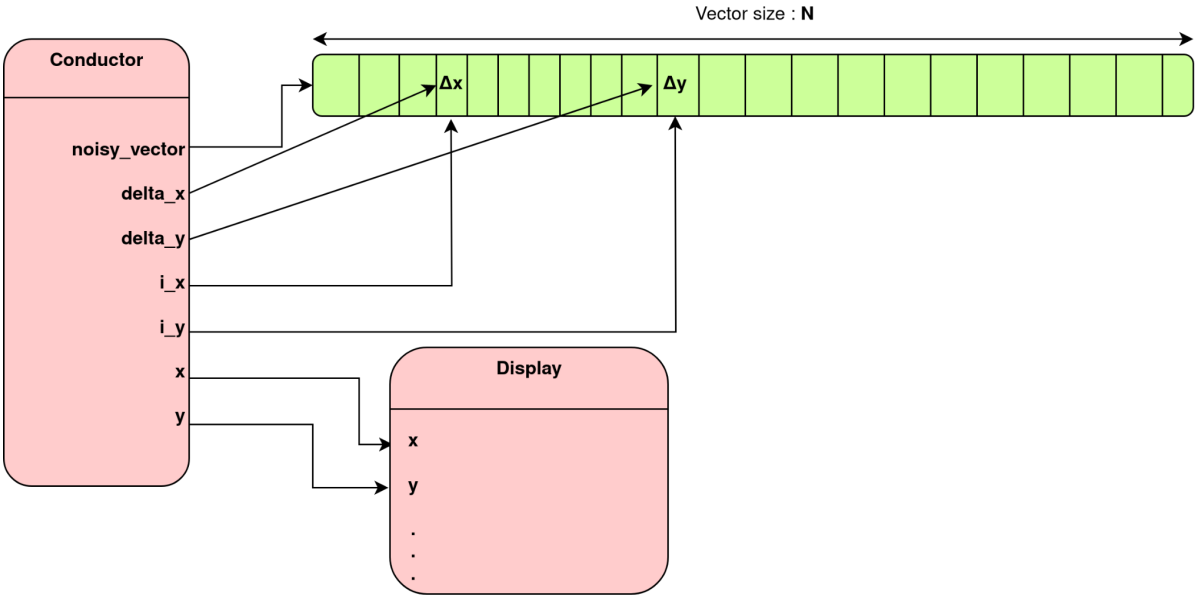

Figure 7 - Module Conductor et signification des sockets sortantes

Le *noisy_vector* ayant été produit à l'issue de la première séquence, le *Conductor* le garde en mémoire pour l'envoyer au module *Add*. De plus, les modifications du vecteur ont lieu au niveau du *i_x* et du *i_y*. Ainsi, à l'instar de *noisy_vector*, *i_x* et *i_y* sont des constantes qui sont envoyées à chaque "tour de boucle" de la séquence 2.

Par ailleurs, le *Conductor* possède aussi des sockets ayant des valeurs variables. C'est le cas de Δx et de Δy. À chaque variation de Δx ou de Δy est associée un pixel dans l'image.

À titre d'exemple, Δx et Δy peuvent varier entre -4 et 4 par pas de 0.05. Cela donnera une image de 160 par 160 pixels. Diminuer le pas augmente la résolution de l'image, alors que modifier la plage de variation modifie l'image (zoom si la plage de valeurs diminue et dézoom si cette plage augmente). Nous pouvons noter que le *Conductor* permet de définir un certain nombre de paramètres relatifs à l'image produite :

- Borne inférieure de la plage de variation,
- Borne supérieure de la plage de variation,
- Pas de déplacement (ou résolution).

Les paramètres Δx et Δy générés comme expliqué ci-dessus s'adressent au module *Add* car c'est lui qui *ajoute* les modifications au *noisy_vector*.

Les paramètres qui sont reliés au module *Add* ont été présentés, il s'agit de présenter ceux qui sont reliés au module *Display*. Les deux sockets de sorties concernées sont *x* et *y*. Elles définissent quel pixel (*x*,*y*) de l'image doit être traité par le module *Display* à chaque étape de la séquence 2. La ***Figure 8*** ci-dessous présente la manière dont sont produits les coordonnées *x* et *y* de chaque pixel.

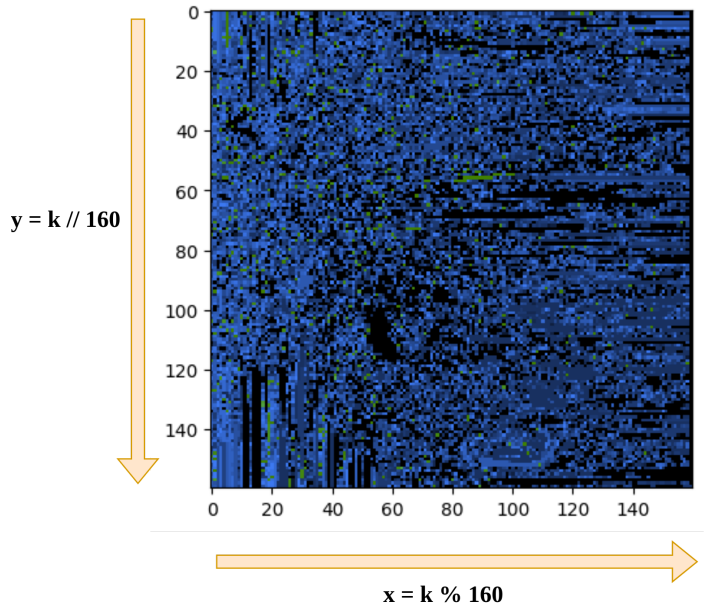

Figure 8 - Lien entre la génération des pixels et l'image produite

Sur la ***Figure 8*** ci-dessus, nous pouvons noter que **k** est un compteur qui est incrémenté à chaque fois que la séquence 2 s'exécute. Ainsi, les *x* sont générés via un *modulo* et les *y* sont générés via une *division entière*. Les pixels sont donc générés ligne par ligne de gauche à droite, puis de haut en bas, comme le sens d'écriture occidental.

2. Conception du module en Python

Après avoir cerné les différentes contraintes liées au module, la conception en Python a été réalisée à l'aide de la bibliothèque Py_aff3ct. Un module est une classe Python qui possède les caractéristiques suivantes :

- Un constructeur qui définit :
  - des attributs,
  - les sockets entrantes et/ou sortantes,
  - le lien entre les *task* et les *sockets*.
- Une ou plusieurs méthodes appelées *task* qui permettent de réaliser une action.

Dans le constructor sont déclarés l'ensemble des attributs. De plus, les attributs *Noisy_vector*, *i_x et i_y*, c'est-à-dire tous les attributs constants, sont initialisés via les paramètres passés au constructeur du module.

Par ailleurs, le module *Conductor* réalise une *task*, c'est-à-dire une action (ou fonction), qui est effectuée à chaque appel du module. Une *task* est définie sous la forme d'une fonction. Cette *task* s'appelle *generate* car elle génère toutes les données nécessaires aux autres modules.

En paramètre de cette fonction qui est la *task* sont passés des arguments que l'on appelle *sockets*. Ce qui nous amène à la troisième étape de création d'un module : la définition des *sockets*. Ici, toutes les *sockets* sont sortantes, il s'agit donc de donner un nom, une dimension et un type à chacune de ces *sockets* de sortie.

Finalement, la dernière étape de création du *Conductor* est d'établir le lien entre les *sockets* de sortie et la *task* précédemment définie.

Une fois toutes ces étapes réalisées, il s'agit de valider le fonctionnement du module. Pour cela, deux techniques sont appliquées :

- Réalisation de tests unitaires pour les 160 par 160 pixels,
- Comparaison des images générées avec et sans le module *Conductor* (ces images doivent être identiques).

La méthode utilisant les tests unitaires a pour avantage d'être exhaustive si l'on applique ces tests sur uniquement 160 par 160 itérations. Cela ne fait en effet que 25600 itération de boucles. Les tests sont présentés ci-dessous sur la ***Figure 9***.

```
1. for x in range(len(delta_x_range)):
2.     for y in range(len(delta_y_range)):
3.         delta_x[:] = delta_x_range[x]
4.         delta_y[:] = delta_y_range[y]
5.         my_src['generate'].exec()
6.
7.        assert((my_src["generate :: r_in" ][0,:]   == r_in[0,:]).all())
8.        assert( my_src["generate :: delta_x"][:]   == delta_x[:]      )
9.        assert( my_src["generate :: delta_y"][:]   == delta_y[:]      )
10.       assert( my_src["generate :: ix_x"   ][:]   == ix_x[:]         )
11.       assert( my_src["generate :: ix_y"   ][:]   == ix_y[:]         )
12.       assert( my_src["generate :: x"      ][:]   == x               )
13.       assert( my_src["generate :: y"      ][:]   == y               )
14.print("Module de source validé !\n")
```

Figure 9 - Tests unitaires pour le module Conductor

Une fois les sept assertions de ce programme de tests validées, le module *Conductor* a été inséré dans la séquence 2 afin d'être officiellement validé.

3. Transcodage du module en C++

La définition du module *Conductor* en Python permet d'avoir une conception fonctionnelle à défaut d'être performante. L'amélioration des performances peut être obtenue par le transcodage en C++ de ce module à l'aide de la bibliothèque Pyaf.

Le transcodage ne pose qu'un problème syntaxique car les problèmes liés à la conception ont déjà été résolus. Cependant, toutes les fonctions Python ne trouvent pas leur équivalent en C++. C'est le cas de la fonction *Numpy.arange* nécessaire pour générer les Δx et Δy. Ainsi, une fonction équivalente à du être codée. Les données produites par cette fonction ne sont pas strictement les mêmes que celles produites par la fonction *Numpy.arange*. Ainsi, les assertions ont dû être ajustées lors des tests unitaires pour le test du module en C++. En effet, les conditions d'égalité entre deux flottants sont rarement remplies.

Le module écrit en C++, une fois lié au code Python à l'aide d'un Wrapper, a pu être validé par les tests unitaires. En effet, l'erreur maximale obtenue pour les Δx et Δy est de l'ordre de 1e-14, ce qui est négligeable.

Finalement, le module *Conductor* en C++ a pu être validé en l'insérant dans la séquence 2.

B. Le module Add

1. Explication du module

Comme présenté en haut, le module **conductor** contrôle les autres blocs, l'un d'eux est le bloc **Add** qui lui génère l'image en modifiant le *noisy_vector*.

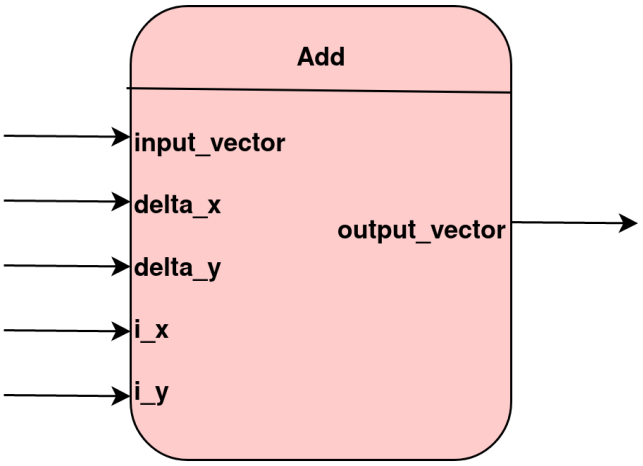

Figure 10 - Présentation du module Add

Le chef d'orchestre fournit au module *add* 5 sockets qui sont; le vecteur d'entrée qui est le *noisy_vector,* les attributs *i_x* et *i_y* qui déterminent à quel niveau du vecteur les modifications auront lieu, et ainsi les valeurs de modification *delta_x* et *delta_y*. Après modification, le vecteur est récupéré en sortie.

Si le décodage est bien fait, le pixel correspondant est noir, sinon il est de couleur differente .

2. Explication du code python

```
class add_impulses(Py_Module):

    def add(self, ix_x, ix_y, delta_x, delta_y, enable, r_in, r_out):
        r_out[0,:] = r_in[:]
        if(enable == 1):
            r_out[0, ix_x] = delta_x
            r_out[0, ix_y] = delta_y
        return 0

    def __init__(self, N):

        Py_Module.__init__(self) # Call the aff3ct Py_Module __init__
        self.name = "add_impulses"   # Set your module's name

        t_add = self.create_task("add") # create a task for your module

        s_ix_x    = self.create_socket_in  (t_add, "ix_x"    , 1, np.int32   ) # create an input socket for the task t_add
        s_ix_y    = self.create_socket_in  (t_add, "ix_y"    , 1, np.int32   ) # create an input socket for the task t_add
        s_delta_x = self.create_socket_in  (t_add, "delta_x", 1, np.float32 ) # create an input socket for the task t_add
        s_delta_y = self.create_socket_in  (t_add, "delta_y", 1, np.float32 ) # create an input socket for the task t_add
        s_enable  = self.create_socket_in  (t_add, "enable" , 1, np.int32   ) # create an input socket for the task t_add
        s_r_in    = self.create_socket_in  (t_add, "r_in"   , N, np.float32 ) # create an input socket for the task t_add
        s_r_out   = self.create_socket_out (t_add, "r_out"  , N, np.float32 ) # create an output socket for the task t_add

        self.create_codelet(t_add, lambda slf, lsk, fid: slf.add(lsk[s_ix_x], lsk[s_ix_y],
        lsk[s_delta_x], lsk[s_delta_y], lsk[s_enable], lsk[s_r_in], lsk[s_r_out])) # create codelet
```

Fig 11 : version python du module *Add*

La fonction "add" modifie le contenu de "r_out" en fonction de la valeur de "enable". Si "enable" vaut 1, les valeurs de "r_out" à l'index "ix_x" et "ix_y" sont remplacées par "delta_x" et "delta_y" respectivement, ce qui ajoute des impulsions aux données d'entrée. Si "enable" est égal à 0, les données d'entrée sont simplement copiées dans "r_out" sans modification.

La fonction "__init__" permet d'initialiser l'objet de la classe. Elle crée la task *"t_add"*, qui est utilisée pour décrire l'opération d'ajout d'impulsions. Les deux méthodes *"create_socket_in"* et *"create_socket_out"* créent les sockets d'entrée et de sortie de la task . Elles permettent le transfert de données entre différentes parties du système.

Enfin, la méthode "*create_codelet*" est utilisée pour créer un objet "*codelet*" qui relie les entrées et les sorties de la tâche à la méthode "*add*". Un codelet est une fonction qui est exécutée dans un contexte parallèle pour améliorer les performances.
En utilisant les codelets, la librairie Aff3ct peut gérer de manière efficace les tâches parallèles et les flux de données entre les différents modules. Cela permet d'optimiser les performances des systèmes de communication.

3. Transcodage vers le C++

Le C++ est un langage de programmation de bas niveau qui permet de contrôler directement les ressources systèmes, comme la mémoire ou les entrées/sorties. Cela permet de réaliser des optimisations supplémentaires en utilisant les ressources système de manière plus efficace, ce qui peut réduire considérablement le temps d'exécution du programme .

Le choix du passage du python en C++ est expliqué par le fait qu'on désire avoir une meilleure performance, en diminuant le temps d'exécution du programme, pour générer l'image plus rapidement .

Une façon de le faire est d'utiliser la technique du multi-threading . En étant pas si simple a implémenter en python, le passage par le C++ permet d'appliquer facilement cette méthode en divisant les tâches de l'application en plusieurs threads qui peuvent être exécutés simultanément sur différents cœurs.

C. Le module Display

Le module display a pour but de stocker l'ensemble des informations permettant de générer l'image. Pour ce faire, il y a deux fonctions essentielles pour réaliser ce module. La fonction Display_impulses qui permet de créer le bloc pour toute l'image. Ajouté à ça, il y a la fonction display qui permet de calculer la valeur des pixels de l'image avec les résultats des erreurs du décodeur. Cette dernière dépendant de la fonction précédente lorsque l'on a créé le bloc display.

Création du module:**Display_impulses**(int **H**,int **W**,int **heat_map**[5][3],int **h_ix**);
Tout d'abord, dans notre fonction Display_impulses, on prend en entrée la taille de l'image, avec les entiers H et W (pour height and weight) qui sont respectivement la hauteur et la largeur de l'image créée. A ça, on y ajoute un heat_map, tableau de 5 par 3 qui contient en

paramètre 5 vecteurs de taille 3. Le vecteur étant utilisé à celui du numéro h_ix correspondant et celui-ci sert de code couleur pour l'image.
Également dans le module, un vecteur d'entiers Map de taille HxWx3 est créé et contient les valeurs de l'image.

Calcul des valeurs de l'image: **display**(int **x**,int **y**,int **BE**,int **enable**);
Cette fonction ci-dessus permet le calcul sur un pixel de l'image. La fonction prend en entrée les coordonnées x et y du point en question et la valeur retranscrite du compteur d'erreurs qui décompte le nombre de valeurs différentes entre l'entrée au niveau du générateur et la sortie du décodeur.
De plus, pour répondre au programme déjà présent, le tableau devait être rempli de la façon suivante:

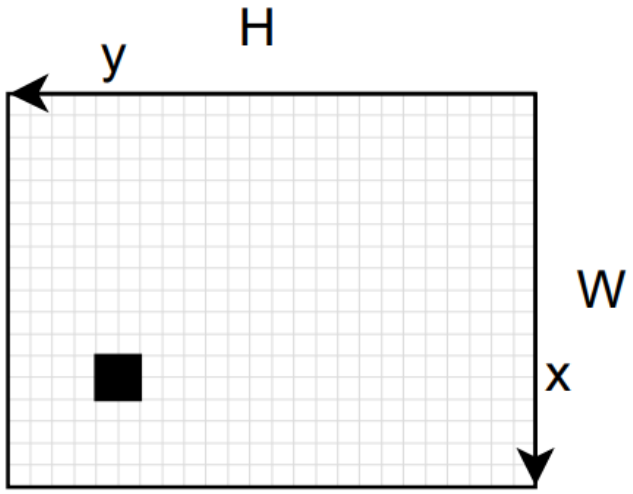

Figure 12 - Schéma représentatif pour la création de l'image

Tout d'abord, les colonnes devaient être inversées et représenter l'axe y. Les colonnes correspondent à la valeur de coordonnée x. Puis, on a également les 3 pixels à remplir, rouge vert et bleu donc la valeur dépend du nombre d'erreurs ainsi que la heatmap.
Ainsi, l'attribution de la valeur des pixels se fait par les affectations suivantes.

```
map[len(delta_y_range)-y-1,x,0] = BE*s_heat_map[s_h_ix,0];
map[len(delta_y_range)-y-1,x,1] = BE*s_heat_map[s_h_ix,1];
map[len(delta_y_range)-y-1,x,2] = BE*s_heat_map[s_h_ix,2];
```

Figure 13 - Code en python de l'attribution des données d'un pixel

A cela, on a également une variable **int** enable qui, quand elle est non nulle, indique que la valeur reçue est correcte et que les calculs pour le pixel en question à bien été terminé. Cela permet d'éviter d'enregistrer des valeurs plusieurs fois ou de le faire alors que le calcul n'a pas été effectué.

De plus, pour connaître le nombre de pixel reçu, on a une variable **int** tempo = H*W; qui permet de décompter le nombre de pixels reçues. Cette variable est équivalente au compteur k du module conductor. Ainsi, une fois notre image complète, on renvoie le signal toggle_done() afin d'informer la fin de la génération du display.

Le vecteur map complet est donc récupéré et peut être enregistré pour l'afficher par la suite.

Pour augmenter les performances, il reste à finir le module en cpp et le rendre fonctionnel avec le reste du projet.

IV. Analyse des performances

A. Présentation des performances

L'amélioration des performances étant l'objectif du projet, commençons par présenter les performances du programme utilisant les modules *Conductor*, *Add* et *Display* écrits en Python.

Module	Temps d'exécution (en s)	Temps d'exécution (en %)
Conductor	1.41	15.92
Add	0.87	9.80
Decoder	4.53	51.14
Monitor	0.06	0.62
Display	2.00	22.52

Figure 14 - Performance du programme de base

Les performances présentées sur la ***Figure 14*** montrent que le goulet d'étranglement du programme est pour moitié le *decoder*, pour moitié les trois modules que nous avons à coder en C++. Dans l'ordre, il s'agit d'écrire tous les modules en C++ puis ensuite d'utiliser le pipelining afin d'accélérer le processus au niveau du *decoder*.

À l'heure de l'écriture de ce rapport, seuls les modules *Conductor* et *add* ont pu être testés en C++. Ainsi, nous présentons exclusivement les performances de ces modules dans ce rapport. Il est intéressant de noter que la simple utilisation du langage C++ permet d'augmenter drastiquement le débit en même temps de réduire la latence des modules. C'est d'ailleurs ce que montre la ***Figure 15*** ci-dessous, comparant les performances des modules codés en Python puis en C++.

Module	Throughput ratio (C++ / Python)	Latency ratio (Python / C++)
Add	199	196
Conductor	152	160

Figure 15 - Comparaison performances Python et C++

Notons que le module *Add* est environ 200 fois plus rapide que le même module Python. De même, le module *Conductor* est plus de 150 fois plus performant que le même module codé en Python.

Les améliorations réalisées permettent d'obtenir les performances globales présentées ***Figure 16***. Ces améliorations font relativiser les améliorations précédentes car le temps de génération d'une image reste relativement long.

Module	Temps d'exécution (en s)	Temps d'exécution (en %)
Conductor	0.01	0.14
Add	0.01	0.09
Decoder	4.23	69.11
Monitor	0.04	0.62
Display	1.83	29.98

Figure 16 - Performance du programme avec modules Pyaf

Ainsi, le temps global d'exécution est amélioré de 30 % environ. Cette amélioration, bien que significative, reste humble en vue de ce qu'il est possible d'obtenir avec certaines techniques. Ces techniques sont présentées au paragraphe suivant.

B. Améliorations possibles

Les performances sont encore loin d'être optimales. En effet, une fois que le module *Display* sera écrit en C++, les performances devraient s'améliorer d'environ 25 % (cf. ***Figure 16***). Ensuite, il sera possible d'utiliser une technique de *pipelining* basée sur le *multiprocessing*. Cette technique est implémentable avec la bibliothèque *py_aff3ct*. Elle permet de mettre en place une architecture pipeline en utilisant les différents cœurs du processeur. Ainsi, les performances peuvent être améliorées d'un facteur proportionnel au nombre de cœurs du processeur utilisés.

V. Présentation de l'interface graphique

A. Présentation de l'outil de développement:

Afin de faciliter l'utilisation de la libraire, il a été décidé d'intégrer le travail effectué dans une interface graphique. Pour ce faire, on a fait appel à l'outil **Pyqt**. Il s'agit d'un module libre permettant de lier le langage Python avec la bibliothèque Qt. Il permet ainsi de créer des interfaces graphiques en Python. Ce module contient plusieurs classes qui sont un bon outil pour l'utilisateur. Pamis les classes utilisés dans ce projet:

- La classe QWidget: Cette classe fournit la capacité de base d'affichage à l'écran et de gestion des événements. Tous les éléments graphiques que Qt fournit sont hérités de QWidget ou sont utilisés avec une classe fille de QWidget.
- La classe QtGui: Elle contient tous les éléments GUI(Graphical user Interface) fournis par Qt allant d'un simple label à la complexe vue graphique. Tous les éléments de cette classe sont appelés widgets. Voici quelques exemples:

- QLabel: un simple widget affichant une image ou un text
- QLineEdit: une zone d'entrée sur une ligne
- QPushButton: Bouton standard pouvant être pressé
- QCheckBox: une simple case à cocher

- La classe QtCore: Cette classe contient toutes les classes essentielles et multiplateformes qui forment le squelette d'une application Pyqt. Elles vont des chaînes de caractères à la gestion des processus , des entrées ainsi que les différentes structures de données. (Exemples : QString , QFile, QProcess, QDate)

B. Présentation du résultat:

Après développement, le résultat obtenu est représenté ci-dessous:

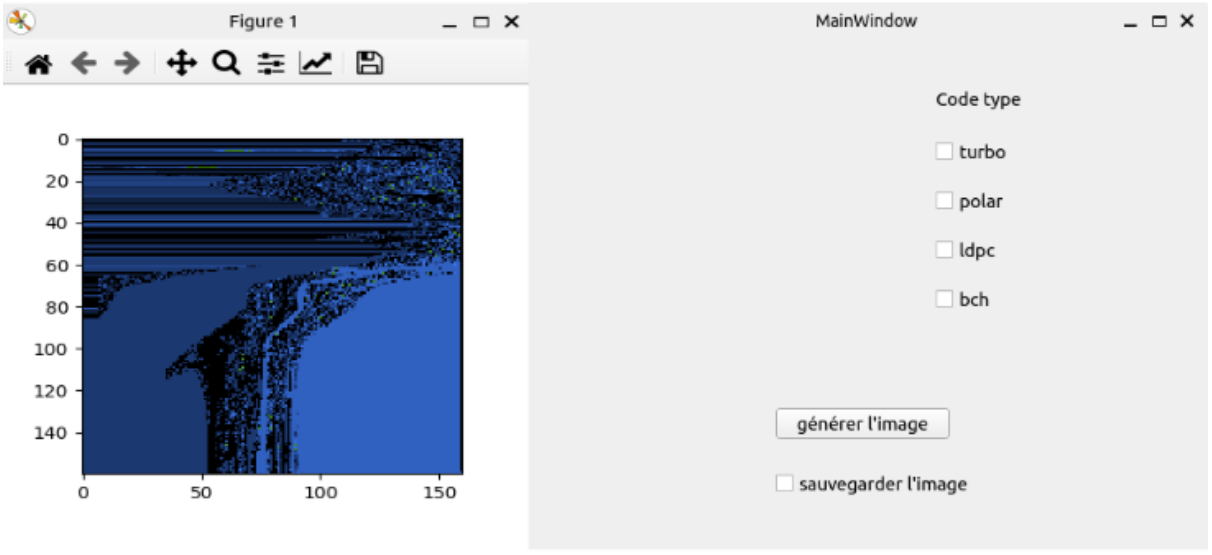

Figure 17: L'interface graphique

Dans un premier temps, l'utilisateur sélectionne le type du code correcteur , il a le choix entre:

- **Turbo code** : C'est une technique de correction FEC utilisée pour améliorer la fiabilité de la transmission de données sur des canaux de communication bruyants. Le principe de ce code est d'introduire une redondance dans le message pour le rendre moins sensible aux bruits et perturbations subies lors de la transmission. Les codes turbos peuvent atteindre des performances proches de la limite de Shannon et ont été largement utilisés dans les systèmes de communication sans fil tels que les réseaux cellulaires 3G,4G …
- **Code polaire:** C'est un type de codage utilisé dans les systèmes de communication numérique afin de transmettre les données binaires. Ce type de codage est caractérisé par deux états de polarité : **positif ou négatif .** Ce type de codage est souvent utilisé pour les systèmes de transmissions de haut débit.
- **code LDPC (Low Density Parity Check):** est un type de codage pour la correction des erreurs qui utilise une matrice de parité à faible densité pour

détecter et corriger les erreurs dans les données transmises. Ce type de code est utilisé dans les réseaux de télécommunications, les systèmes de stockage des données et les systèmes de transmission vidéo.

Après avoir sélectionné le type du code, l'utilisateur génère l'image en cliquant sur le bouton **"générer l'image"** . L'enregistrement de l'image générée est possible en cochant la case **"sauvegarder l'image" .** Cette dernière est enregistrée dans le répertoire de travail.

VI. Conclusion

Pour conclure, l'objectif du projet était d'améliorer une bibliothèque Python. Nous avons donc répondu aux attentes de ce projet, à savoir que nous avons amélioré les performances du programme en enrichissant la bibliothèque *pyaf* de nos trois modules : *Conductor*, *Add* et *Display*. Cependant, il est possible d'améliorer encore les performances du programme en utilisant une technique nommée "pipelining". Par ailleurs, nous avons mis au point une interface graphique qui permet une utilisation plus visuelle du programme.

De manière plus générale, ce projet nous a permis d'approfondir le langage Python ainsi que le langage C++ sur un projet ludique.