

ADC $\Delta\Sigma$ et Antenne Active

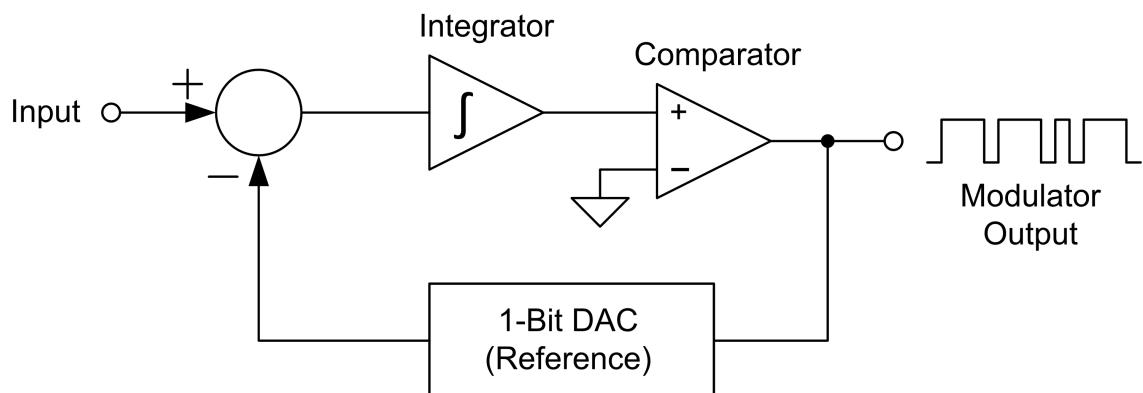


Table des matières

1	Introduction	3
2	Le fonctionnement de base de l'ADC $\Delta\Sigma$	3
2.1	Modulateur $\Delta\Sigma$	3
2.2	Filtrage numérique et décimation	5
2.3	Avantages et inconvénients des ADC $\Delta\Sigma$	6
2.3.1	Avantages	6
2.3.2	Inconvénients	7
3	Partie pratique : Modélisation de l'ADC	7
3.1	Modélisation python	7
3.1.1	Premier test avec un sinus en entrée	7
3.1.2	Test de l'ADC avec une entrée QPSK	8
3.1.3	SNR en fonction de la porteuse	10
3.2	Modélisation GNU radio	13
3.2.1	Premiers pas : la fonction amplifiers	13
3.2.2	Bloc python pour l'ADC en entrée sinusoïdale	14
3.2.3	Modélisation pour une QPSK	16
3.2.4	Bloc python personnalisé	16
4	Antenne active	17
4.1	Fonctionnement d'une antenne active	17
4.2	Modélisation d'une antenne active	19
4.3	Steering en temps / Steering en phase	20
4.4	Le phénomène de beamsquint	20
4.5	Simulation beamsquint	21
5	Conclusion	29
6	Sources	29
7	Annexe : Explication de la méthode M2M4 pour le calcul du SNR	30

1 Introduction

Depuis des années, les techniques analogiques ont le plus été utilisés dans le domaine du traitement du signal. Cependant, les techniques numériques commencent progressivement à s'y imposer, notamment dans le domaine des télécommunications spatiales. Cela implique l'utilisation de convertisseurs analogiques-numériques et de convertisseurs numériques-analogiques de plus en plus puissants. Une illustration parfaite de cette transition est la conception des convertisseurs analogique-numérique delta-sigma. Les ADC $\Delta\Sigma$ se distinguent par leur capacité à convertir des signaux analogiques sur une vaste gamme de fréquences, allant du continu jusqu'à plusieurs dizaine de gigahertz. Cette largeur de bande permet d'envisager la numérisation directe de la bande Ka Rx (30 GHz) sans utiliser de convertisseur de fréquences. De nouvelles architectures d'antennes active en bande Ka entièrement numériques apparaissent aujourd'hui. Le but de ce stage est d'étudier le fonctionnement de ces ADCs, leur intégration dans une antenne active de type DRA (Direct Radiating Array) et d'analyser les performances obtenues avec de la formation de faisceau numérique basées soit sur de la phase soit du délai.

2 Le fonctionnement de base de l'ADC $\Delta\Sigma$

Comme tout convertisseur, l'ADC $\Delta\Sigma$ transforme un signal analogique en un signal numérique. Voici un schéma basique :

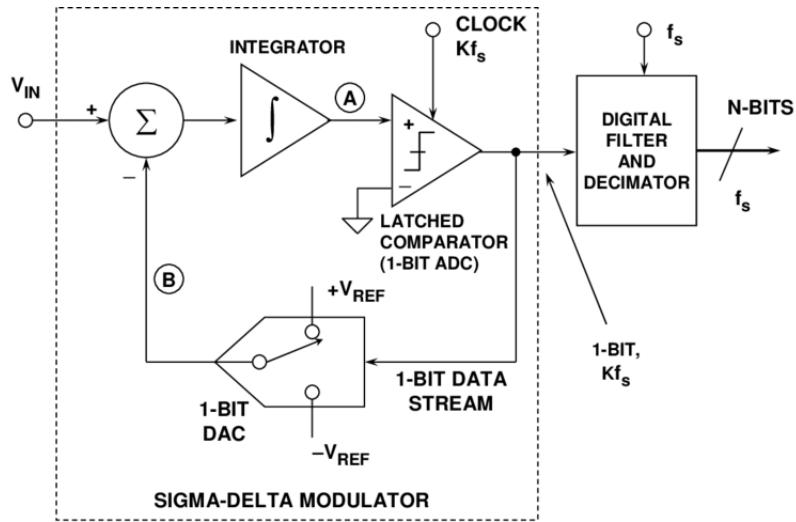


FIGURE 1 – Schéma basique d'un ADC $\Delta\Sigma$ d'ordre 1

L'entrée V_{in} correspond au signal analogique d'entrée, tandis que la sortie numérique N-BITS se trouve à la fin de la chaîne. Ce convertisseur est divisé en deux parties distinctes : le modulateur Delta-Sigma et le filtrage numérique/décimation. Expliquons le rôle de chacune des deux parties séparément.

2.1 Modulateur $\Delta\Sigma$

Le modulateur $\Delta\Sigma$ est le cœur de l'ADC $\Delta\Sigma$. Il numérise le signal d'entrée analogique et réduit le bruit aux basses fréquences. Le taux d'échantillonnage du modulateur est des centaines de fois plus rapide que les résultats numériques aux ports de sortie, échantillonnant le signal d'entrée à un taux très élevé dans un flux de 1 bit. Ce suréchantillonnage ne vise pas à améliorer la granularité du signal, mais plutôt à disperser le bruit de quantification sur un spectre de fréquences plus large.

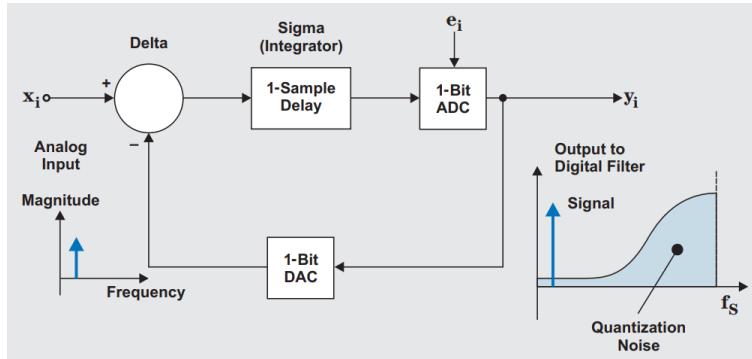


FIGURE 2 – ADC $\Delta\Sigma$ du premier ordre dans le domaine fréquentiel

Le modulateur convertit le signal d’entrée analogique en une onde d’impulsion modulée à haute vitesse et à un seul bit. Cette modulation pousse le bruit de basse fréquence vers des fréquences plus élevées, en dehors de la bande d’intérêt, grâce à une fonction appelée noise shaping.

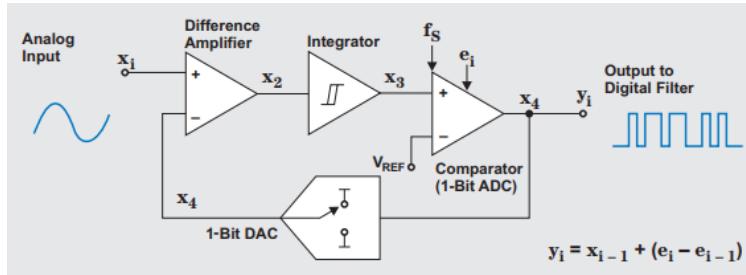


FIGURE 3 – ADC $\Delta\Sigma$ du premier ordre dans le domaine temporel

La quantification du modulateur sigma-delta est effectuée à un taux d’échantillonnage élevé correspondant à l’horloge du système, produisant un flux de valeurs numériques à 1 bit où le rapport entre les 1 et les 0 représente la tension analogique d’entrée. Ce signal de sortie est une représentation par impulsions du signal d’entrée à la fréquence d’échantillonnage (f_S). Si le train d’impulsions de sortie est moyenné, il est égal à la valeur du signal d’entrée.

L’utilisation d’un circuit avec deux intégrateurs au lieu d’un permet de réduire le bruit de quantification dans la bande du modulateur. La figure suivante montre un modulateur de second ordre à 1 bit qui comporte deux intégrateurs, où le terme de bruit dépend des deux erreurs précédentes :

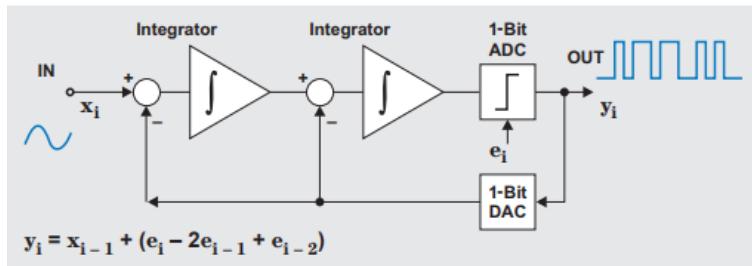


FIGURE 4 – ADC $\Delta\Sigma$ du second ordre

Les modulateurs de deuxième ordre ou d’ordre multiple sont plus complexes à concevoir en raison des boucles multiples, mais ils façonnent le bruit de quantification à des fréquences encore

plus élevées que les modulateurs d'ordre inférieur :

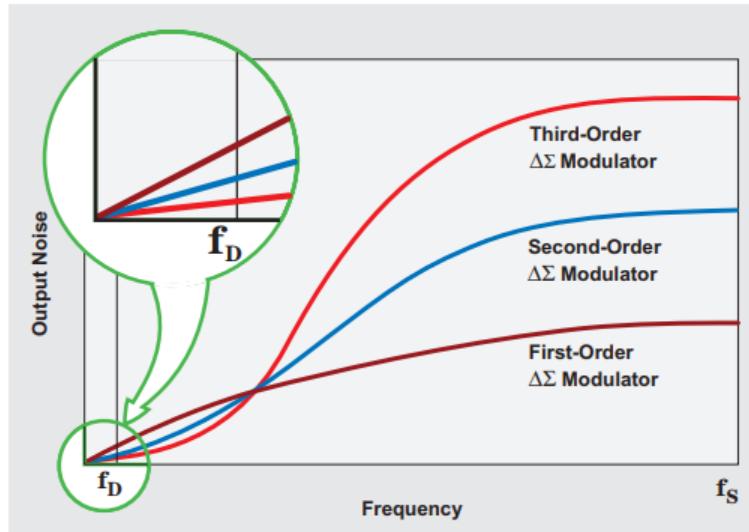


FIGURE 5 – Noise shaping en fonction de l'ordre du modulateur

Finalement, le modulateur de l'ADC $\Delta\Sigma$ réduit efficacement le bruit à basse fréquence pendant le processus de conversion, mais le bruit à haute fréquence reste un problème indésirable dans la sortie finale du convertisseur. La section suivante expliquera comment éliminer ce bruit à l'aide d'un filtre numérique/décimation passe-bas, qui correspond à la seconde partie de l'ADC $\Delta\Sigma$.

2.2 Filtrage numérique et décimation

Chaque échantillon individuel est accumulé dans le temps et "moyenné" avec les autres échantillons du signal d'entrée à travers le filtre numérique/décimation. Ce filtre prend les données échantillonnées et les convertit en un code numérique haute résolution, plus lent.

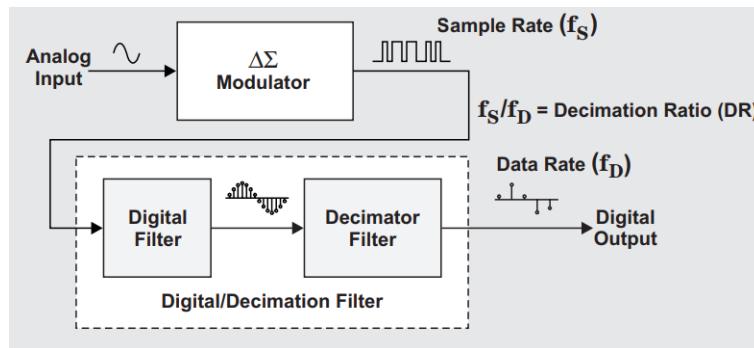


FIGURE 6 – Diagramme bloc complet

La fonction de filtre numérique implémente un filtre passe-bas en échantillonnant d'abord le flux du modulateur de code 1 bit. Un filtre de moyennage est la technique de filtrage la plus couramment utilisée dans les convertisseurs $\Delta\Sigma$. La vitesse de sortie d'un filtre numérique est la même que la fréquence d'échantillonnage.

Dans le domaine temporel, le filtre numérique est responsable de la haute résolution du convertisseur $\Delta\Sigma$. Dans le domaine fréquentiel, le filtre numérique applique uniquement un filtre passe-bas au signal, atténuant ainsi le bruit de quantification du modulateur aux fréquences plus élevées.

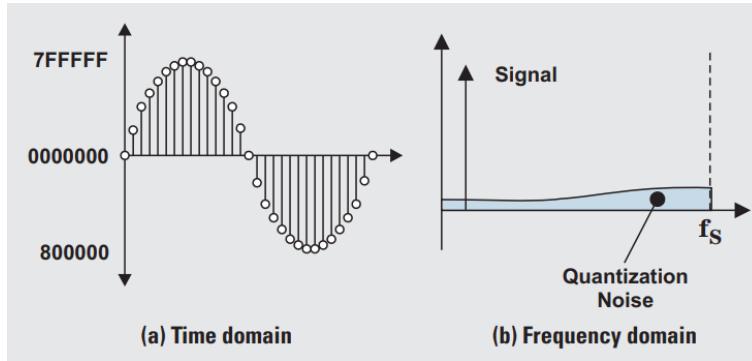


FIGURE 7 – Signal en sortie du filtre (Entrée sinusoïdale dans le modulateur)

Le signal est désormais une version numérique haute résolution du signal d'entrée. Cependant, fournir chaque échantillon du convertisseur serait inutile car cela nécessiterait un contrôleur ou processeur très rapide. De plus, bien qu'il semble y avoir une abondance d'échantillons de haute qualité à la fréquence d'échantillonnage élevée du modulateur, la plupart ne fournissent pas d'informations utiles. Il est donc nécessaire de réduire le nombre d'échantillons, et c'est exactement le rôle du décimateur. La décimation de certains échantillons n'entraîne aucune perte d'informations.

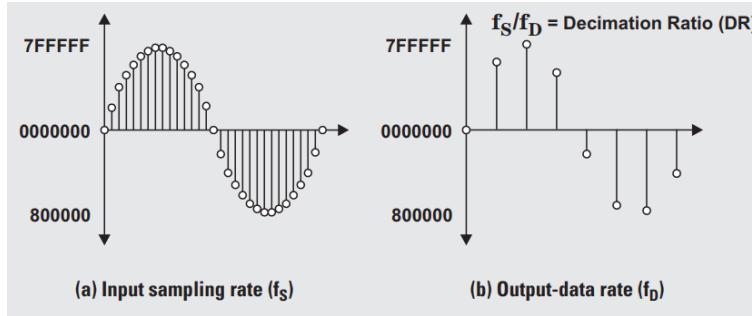


FIGURE 8 – Principe de décimation

2.3 Avantages et inconvénients des ADC $\Delta\Sigma$

Les convertisseurs analogique-numérique à modulation Delta-Sigma présentent plusieurs avantages et inconvénients qui les rendent adaptés à certaines applications, ou pas du tout pour d'autres.

2.3.1 Avantages

- **Haute résolution :** Les ADC Delta-Sigma sont capables d'atteindre des résolutions élevées, typiquement 16 bits et plus, ce qui les rend appropriés pour des applications nécessitant une précision fine dans la conversion analogique-numérique.
- **Réduction du bruit dans la bande de base :** Grâce au noise shaping, le bruit de quantification est déplacé vers des fréquences élevées, loin de la bande de fréquence d'intérêt. Cela permet d'obtenir des signaux numériques de haute qualité sans compromis sur la précision.
- **Flexibilité dans la conception :** Les ADC Delta-Sigma peuvent être adaptés à différentes applications en ajustant des paramètres tels que l'ordre du modulateur et la fréquence d'échantillonnage, offrant ainsi une flexibilité significative dans la conception des systèmes.

- **Faible sensibilité au bruit analogique :** En raison de leur architecture, les ADC Delta-Sigma sont moins sensibles au bruit analogique comparé à d'autres types d'ADC, ce qui est un avantage critique dans des environnements où le bruit électromagnétique est un problème.

2.3.2 Inconvénients

- **Complexité :** Les modulateurs Delta-Sigma d'ordre supérieur, nécessaires pour des performances optimales, peuvent être complexes à concevoir et à mettre en œuvre.
- **Latence élevée :** En raison du processus de filtrage numérique et de décimation, les ADC Delta-Sigma ont une latence inhérente qui peut être incompatible avec certaines applications nécessitant une réponse rapide en temps réel.
- **Cout :** La mise en œuvre de convertisseurs Delta-Sigma avec des performances élevées peut entraîner des coûts plus élevés en raison de la complexité du design et des exigences en matière de composants de haute précision.

3 Partie pratique : Modélisation de l'ADC

3.1 Modélisation python

3.1.1 Premier test avec un sinus en entrée

Le script python correspondant à cette section est "Premiere_partie_sinus.py".

Dans cette section, nous allons simplement essayer de vérifier le bon comportement de notre adc avec un signal en entrée très simple : une sinusoïde.

Notre fonction prend en entrer un signal analogique. Nous commençons par le suréchantillonner, puis ensuite nous rentrons dans la boucle. Dans cette boucle nous appliquons le même traitement pour tous les échantillons de données : nous différencions avec l'ancienne sortie, nous intégrons, nous quantifions et nous comparons avant de le rebalancer dans un DAC pour qu'il retourne dans la boucle. Voici notre fonction ADC Delta Sigma en python :

```
@jit(nopython=True)
def ADC_Delta_Sigma(signal_in, Vpp, enob, upsampling):
    signal_in_up = ZOH(np.real(signal_in),upsampling)

    signal_feedback = np.zeros(len(signal_in_up))
    signal_integrator = np.zeros(len(signal_in_up))
    integrator_values = np.zeros(len(signal_in_up))
    signal_out = np.zeros(len(signal_in_up))
    bits_out = np.zeros(len(signal_in_up))
    quant_noise = np.zeros(len(signal_in_up))

    # Loop
    for i in np.arange(0,len(signal_in_up)):

        # Delta
        if i == 0:
            signal_feedback[i] = signal_in_up[i]
        if i > 0:
            signal_feedback[i] = signal_in_up[i] - signal_out[i-1]

        # Intégrateur
        if i == 0:
            signal_integrator[i] = signal_feedback[i]
        if i > 0:
            signal_integrator[i] = signal_feedback[i] + signal_integrator[i-1]

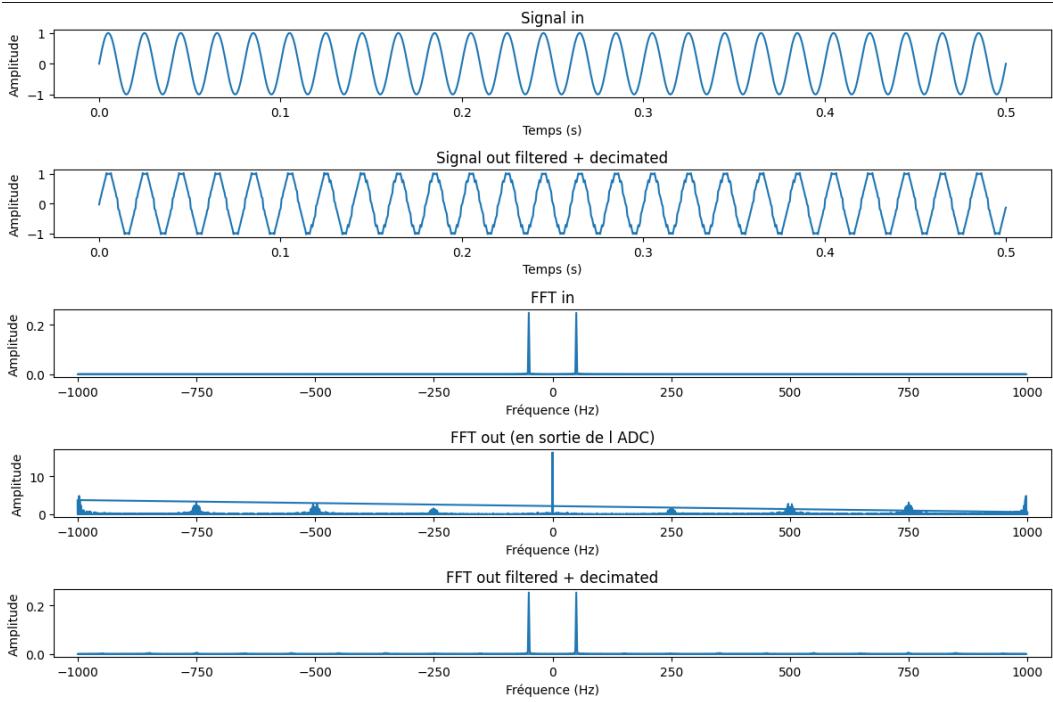
        signal_integrator[i] = Vpp/(2**enob)* np.round(signal_integrator[i]/(Vpp/(2**enob)))

        # Comparateur
        if signal_integrator[i] >= 0:
            signal_out[i] = Vpp/2
            bits_out[i] = 1
        if signal_integrator[i] < 0:
            signal_out[i] = -Vpp/2
            bits_out[i] = 0

    return signal_out, bits_out
```

FIGURE 9 – Fonction ADC Delta Sigma

Voici les résultats que nous obtenons pour une entrée sinusoïdale :



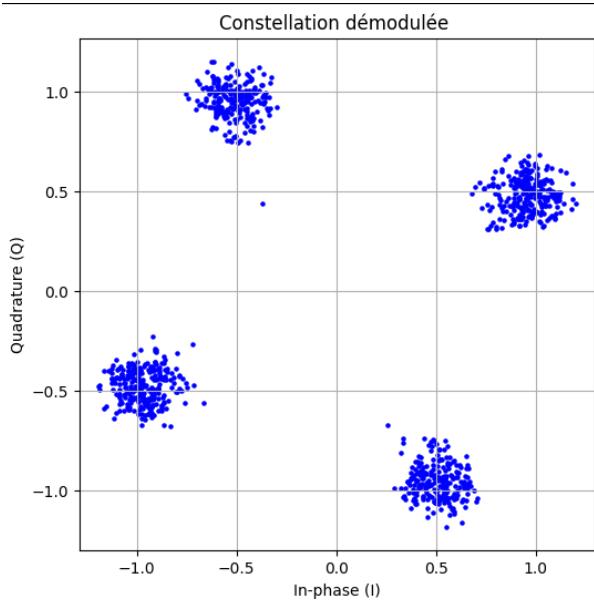
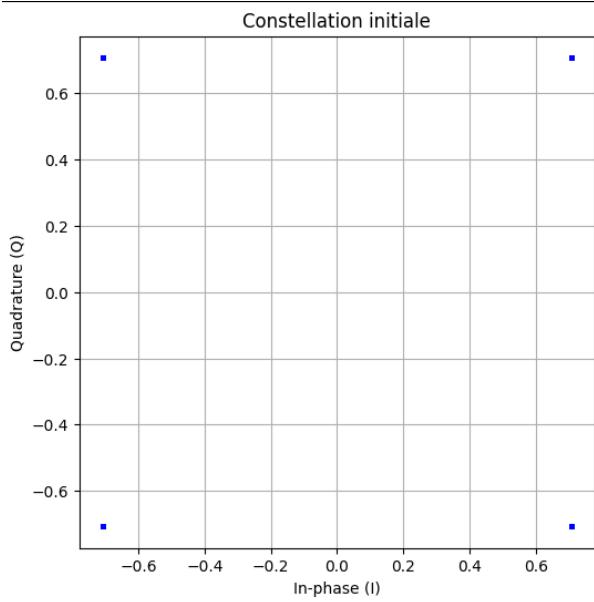
3.1.2 Test de l'ADC avec une entrée QPSK

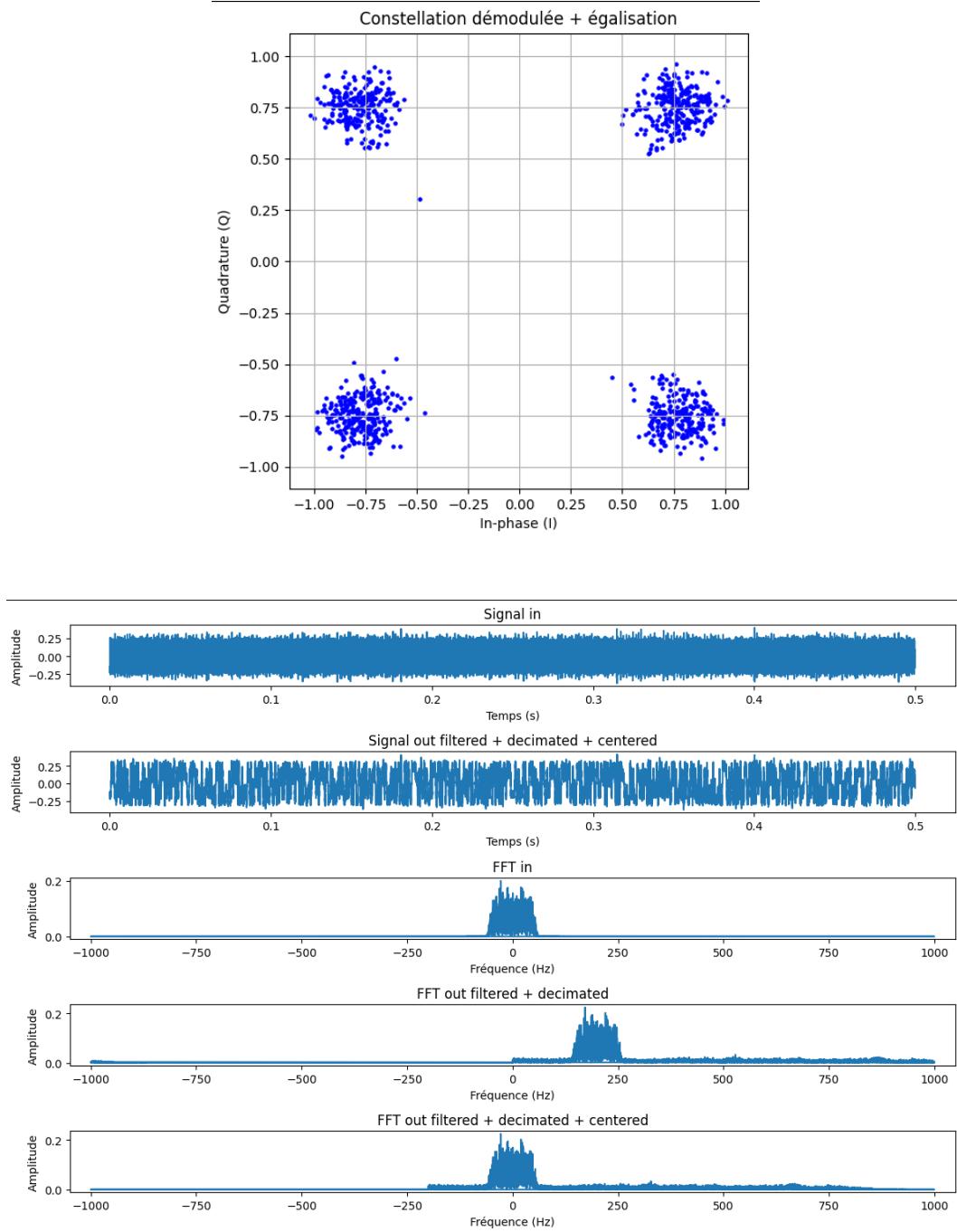
Le script python correspondant à cette section est "Premiere_partie_QPSK.py".

Dans cette section nous allons modéliser simplement le passage d'un signal QPSK dans un adc delta sigma et comparer les différentes constellations afin de vérifier la bonne modélisation mise en place.

On commence par créer un signal QPSK disponible dans usual.py, que l'on bruite avant de le mettre sur une porteuse. Ensuite nous faisons comme dans la partie précédente, nous faisons rentrer le signal dans l'adc, et nous récupérons la sortie dans la variable `signal_out`. nous appliquons par la suite tout le traitement nécessaire à la démodulation du signal afin de retrouver la constellation initiale. La fonction `demodulator_non_data_aided` ne fonctionnant pas sur mon code, il a fallu démoduler à la main en trouvant le premier échantillon qui allait maximiser le SNR. Il a aussi fallu rajouter un peu d'égalisation.

Voici les résultats :

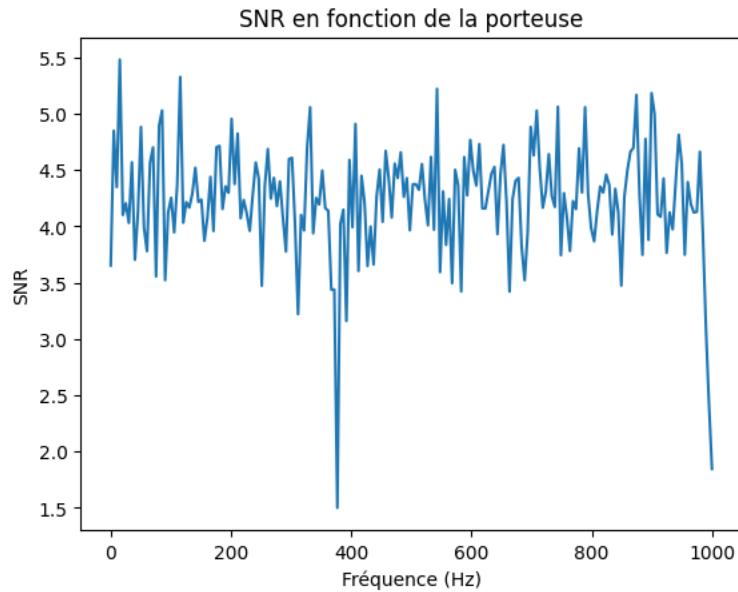




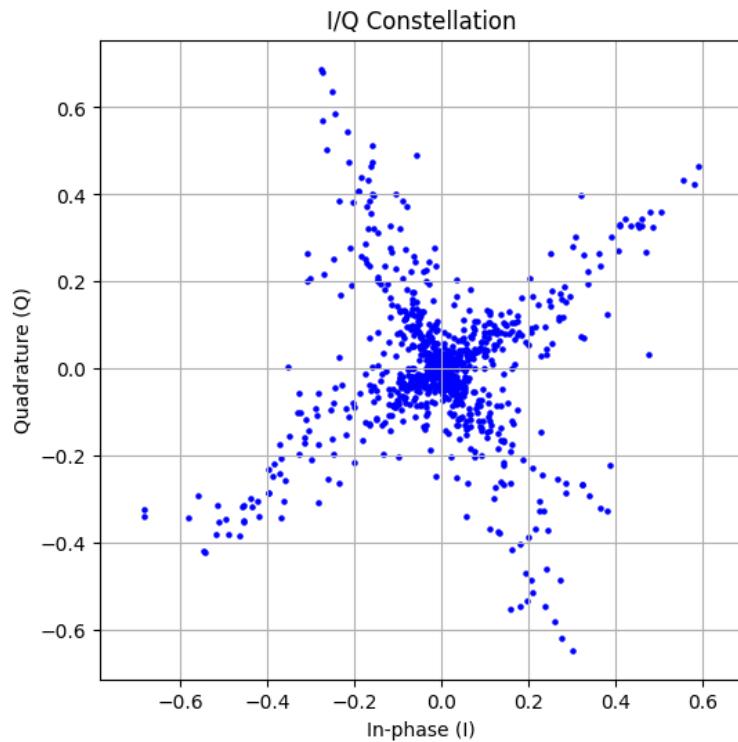
3.1.3 SNR en fonction de la porteuse

Le script python correspondant à cette section est "Premiere_partie_SNR.py" (Compilation assez longue $\simeq 5\text{min}$).

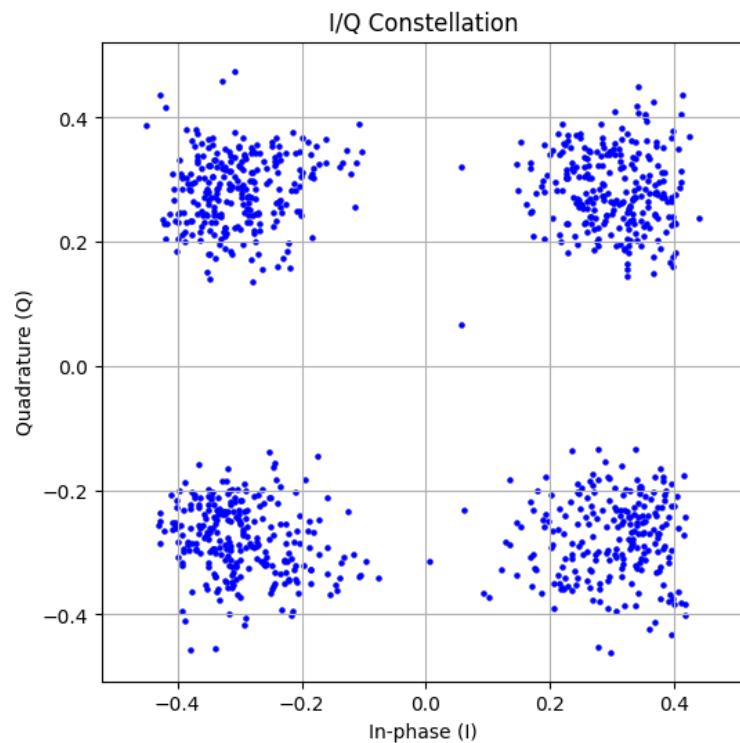
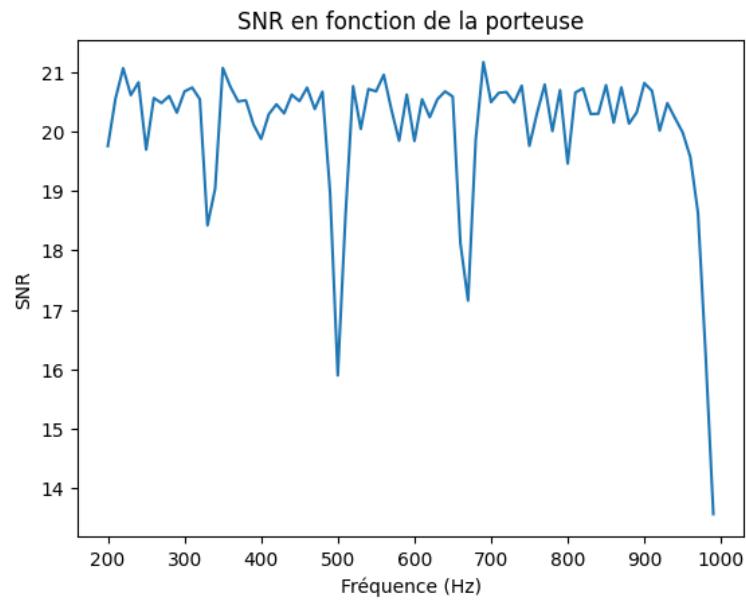
On essaye maintenant de tracer la courbe du SNR en fonction de la porteuse utilisée. Nous commençons par diminuer la taille de la bande passante du signal et pour ce faire, nous augmentons simplement le nombre d'échantillons par symbole. Le résultat est visible sur la courbe suivante :



On remarque que le SNR est particulièrement bas. Alors en traçant la constellation pour vérifier le bon fonctionnement de l'adc, nous obtenons ceci :



Ceci semble dû au fait que la quantification n'est pas assez précise. En passant de enob = 3 à enob = 4 nous obtenons de meilleures performances :



Finalement, le SNR reste assez constant pour n'importe quelle porteuse de fréquence $< fs/2$ (fs = fréquence d'échantillonnage = 2000 Hz ici), à part une légère perte aux alentours de 500Hz .

3.2 Modélisation GNU radio

3.2.1 Premiers pas : la fonction amplifiers

Le script grc correspondant à cette section est "Amplifier.grc".

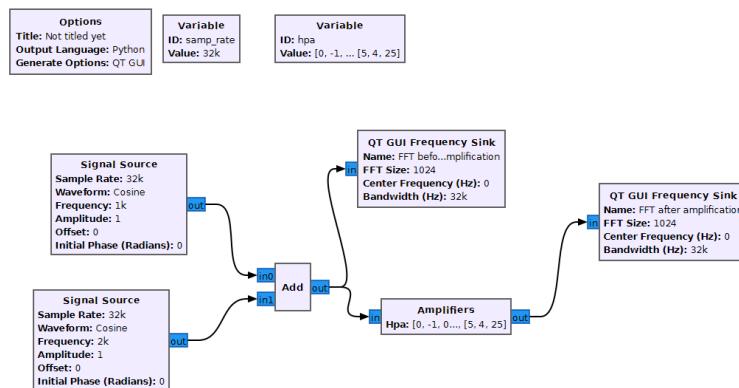
Nous allons maintenant essayer de modéliser tout ça dans GNU radio. Mais pour l'instant, il faut d'abord se familiariser avec l'outil. Il se compose de plusieurs blocs déjà présents dans le logiciel, mais il permet aussi de créer nos propres blocs. Pour ce faire, nous allons essayer d'implémenter une fonction toute simple : usual.amplifier. Cette fonction modélise l'amplification d'un signal d'entrée en utilisant un modèle de conversion AMAM (Amplitude-to-Amplitude) et AMPM (Amplitude-to-Phase). Si tout se passe correctement, nous serons en mesure de voir les non-linéarités du signal de sortie. Voici notre code python de notre bloc "Amplifiers" :

```

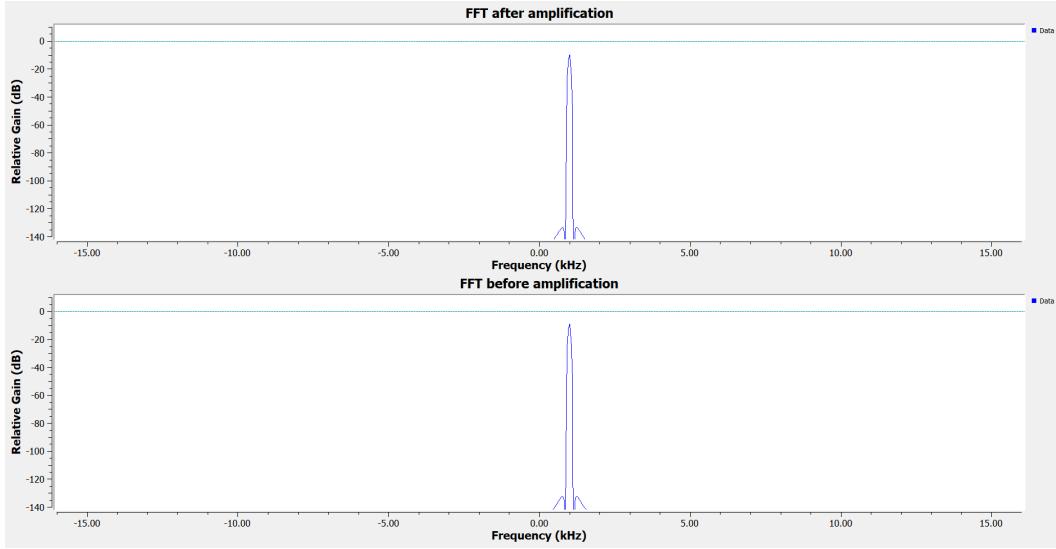
1 import numpy as np
2 from gnuradio import gr
3
4 class blk(gr.sync_block): # other base classes are basic_block, decim_block, interp_block
5     """Embedded Python Block example - a simple multiply const"""
6
7     def __init__(self, hpa=1):
8         """arguments to this function show up as parameters in GRC"""
9         gr.sync_block.__init__(
10             self,
11             name='Amplifiers', # will show up in GRC
12             in_sig=[np.complex64],
13             out_sig=[np.complex64]
14         )
15         self.hpa = np.array(hpa)
16
17     def amplifier(self, signal_in):
18         ibo_range = self.hpa[:, 0]
19         obo_range = self.hpa[:, 1]
20         phase_range = self.hpa[:, 2]
21         signal_in_am = 20 * np.log10(np.abs(signal_in))
22         signal_in_pm = np.angle(signal_in)
23         signal_out_am = np.interp(signal_in_am, ibo_range, obo_range)
24         signal_out_pm = np.interp(signal_in_am, ibo_range, phase_range) * np.pi / 180
25         signal_out = 10**signal_out_am * np.exp(1j * (signal_in_pm + signal_out_pm))
26
27         return signal_out
28
29     def work(self, input_items, output_items):
30         signal_in = input_items[0]
31         signal_out = self.amplifier(signal_in)
32
33         output_items[0][:] = signal_out
34
35         return len(output_items[0])

```

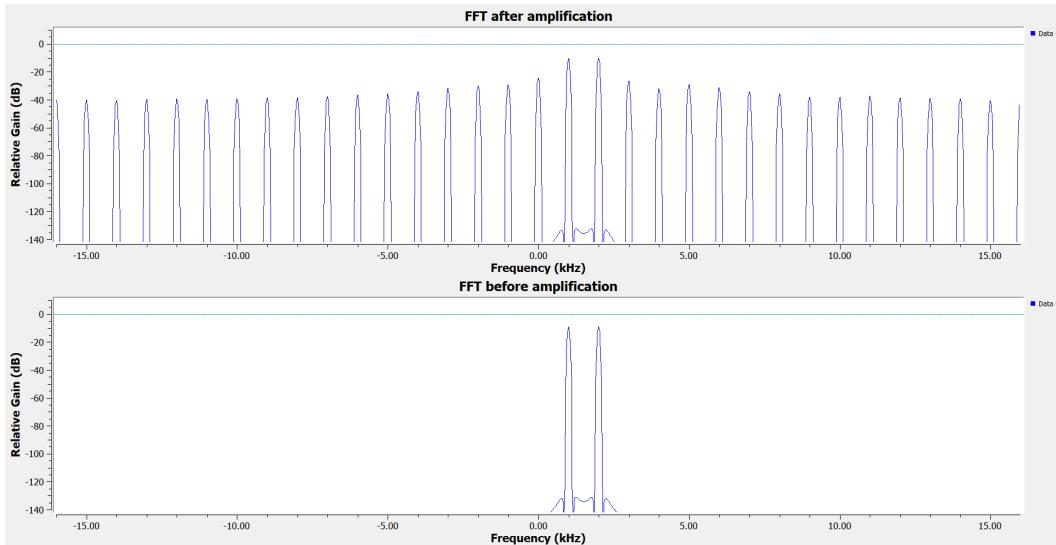
Et voici les blocs correspondants :



Nous avons deux sources à deux fréquences différentes. Si nous décidons de voir ce que donne la fft d'un seul signal avant et après être passé dans la fonction, voici ce que nous obtenons :



Et maintenant, voici les résultats que nous obtenons pour une somme de deux signaux :

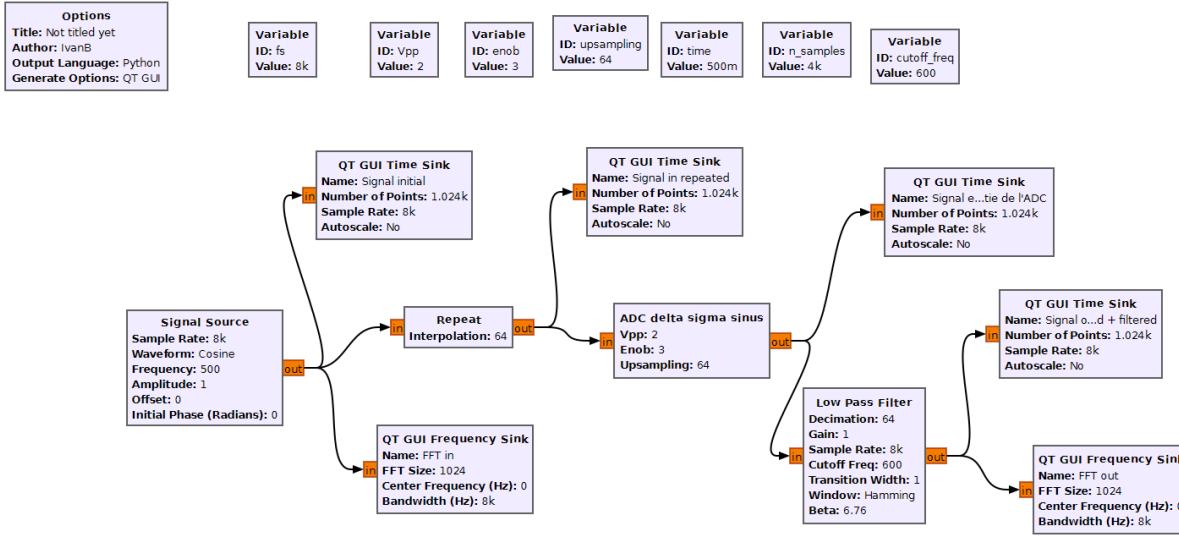


Notre bloc "Amplifiers" fonctionne donc correctement. Nous avons donc maintenant les bases de la construction d'un bloc GNU radio.

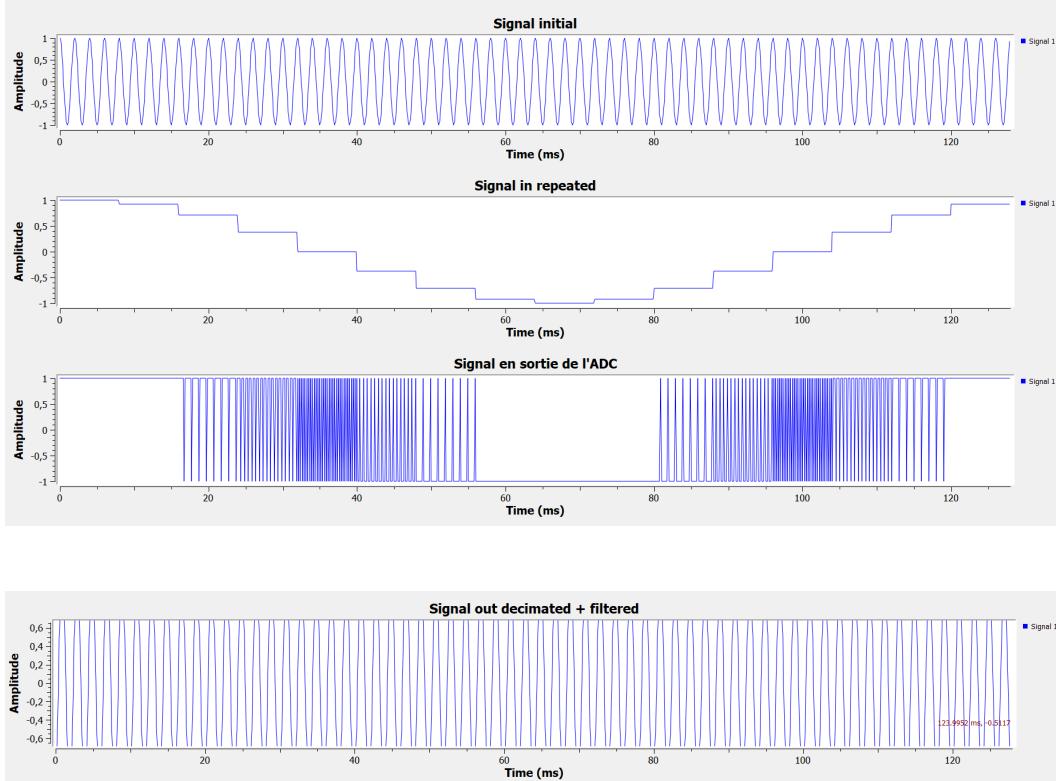
3.2.2 Bloc python pour l'ADC en entrée sinusoïdale

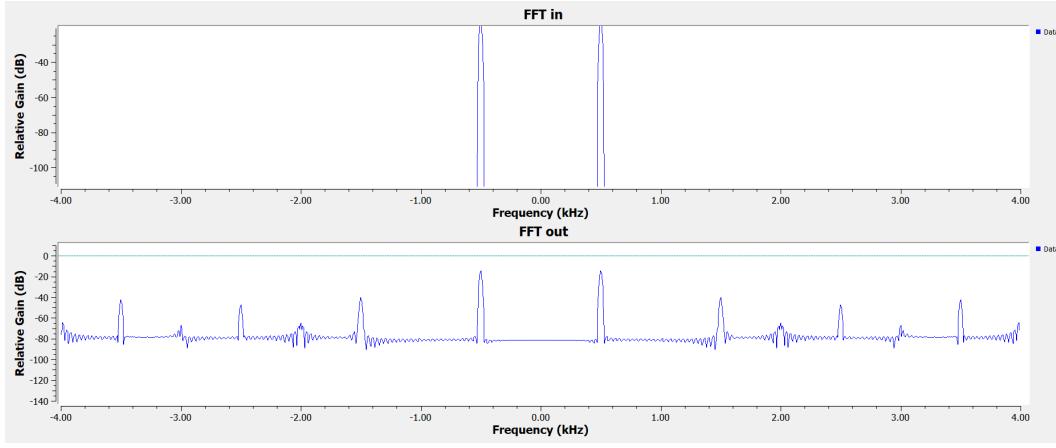
Le script grc correspondant à cette section est "`ADCsinus.grc`".

Nous allons maintenant créer notre bloc correspondant à l'ADC. Pour ce faire, nous reprenons notre code python initial pour une entrée sinusoïdale (`Premiere_partie_sinus.py`), et nous le transposons pour le rendre compatible dans GNU radio.



Ici, le suréchantillonnage est fait avant d'entrer dans le bloc python de l'ADC. En sortie de bloc, nous appliquons un filtre passe-bas qui réalise la décimation dans le même temps. Nous visualisons les différentes FFT et signaux temporels tout au long de la chaîne et obtenons les résultats suivants :



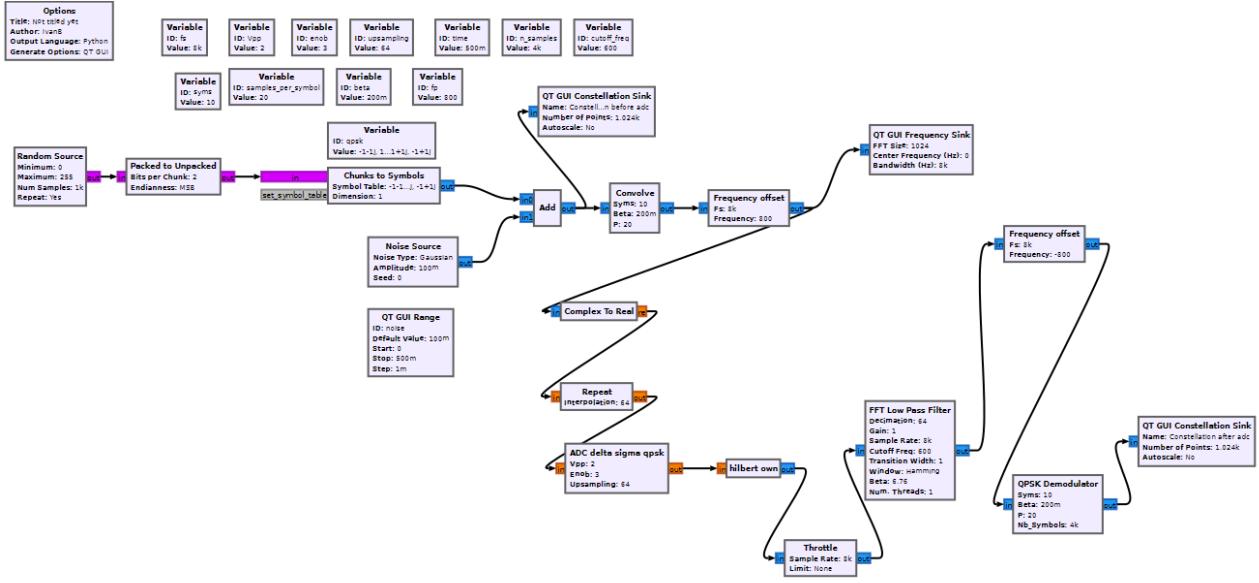


Nous retrouvons donc bien toutes les caractéristiques obtenues dans notre code python de base.

3.2.3 Modélisation pour une QPSK

Le script grc correspondant à cette section est "ADCqpsk.grc".

Essayons de faire la même chose avec un signal QPSK en entrer cette fois-ci. À l'image de ce que nous avons fait précédemment, nous allons générer un signal QPSK et l'envoyer dans notre bloc ADC avant de démoduler et retrouver la constellation initiale. Voici le flowgraph :



3.2.4 Bloc python personnalisé

Pour créer un bloc personnalisé sous GNU Radio, deux approches principales existent : les blocs Python simples ou les blocs Python OOT (Out-Of-Tree).

Un bloc Python simple s'intègre directement dans le flux de travail de GNU Radio. Il est défini et utilisé dans le même script Python. Sa création est rapide pour des tâches simples ou des prototypes, mais ces blocs sont moins modulaires et réutilisables, limitant leur utilisation ailleurs

sans copie de code. Nous ne pouvons donc pas réutiliser un script python simple pour d'autres projets. Le seul moyen est de copier/coller le code python sur un autre projet.

Un OOT bloc Python, conçu pour être indépendant du script principal, est compilé et installé dans GNU Radio, accessible comme tout autre bloc prédéfini. Il offre une approche plus robuste, adaptée aux projets de grande envergure ou nécessitant des blocs réutilisables. Cependant, sa création nécessite une configuration beaucoup plus complexe, avec une courbe d'apprentissage plus élevée, nécessitant pas mal de temps pour créer un simple bloc. Il faut en effet utiliser des serveurs du style XLaunch et rentrer de nombreuses commandes sur le terminal avant de pouvoir créer un simple bloc.

4 Antenne active

4.1 Fonctionnement d'une antenne active

En communications spatiales, la gestion des transmissions de données est complexe en raison de la grande distance entre l'émetteur et le récepteur. Pour améliorer les performances de communication, il est crucial de gérer la directivité et le gain de l'antenne afin de transmettre le maximum de données avec une puissance minimale, tout en maximisant le rapport signal/bruit (SNR). Cela implique un filtrage spatial en contrôlant les faisceaux émis, une technique connue sous le nom de beamforming.

Avec le beamforming, nous pouvons aisément transmettre des données à un utilisateur ciblé. Cependant, il est également nécessaire de le faire pour plusieurs utilisateurs simultanément. Il est peu pratique d'avoir un satellite dédié à chaque utilisateur, ce qui nécessite une solution plus flexible. Modifier physiquement l'antenne pour changer la direction de rayonnement est une solution, mais elle présente plusieurs inconvénients :

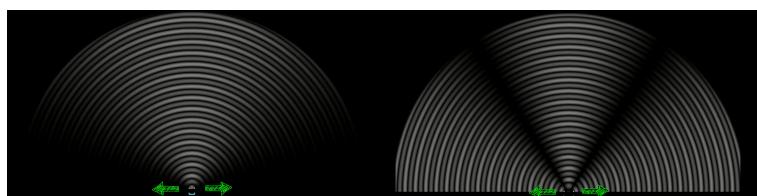
- Les antennes sont difficiles à déplacer.
- La fréquence de changement de direction est beaucoup trop élevé par rapport à la capacité de déplacement physique d'une antenne.
- La consommation d'énergie est trop importante.

Une solution consiste à changer la direction du faisceau électroniquement. En ajustant la phase de chaque petite antenne, nous pouvons manipuler les interférences pour former un faisceau adapté à l'utilisation. Ces antennes sont appelées "antennes à réseau phasé" ou "antennes actives".

Premièrement, pour créer un faisceau, plusieurs paramètres entrent en compte :

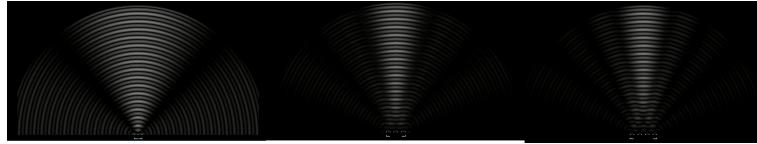
- **L'espacement entre les antennes.**

Plus nous écartons l'espace entre les éléments rayonnants, plus le faisceau principal se réduit, mais nous formons des lobes secondaires indésirables.



— Nombre d'antennes.

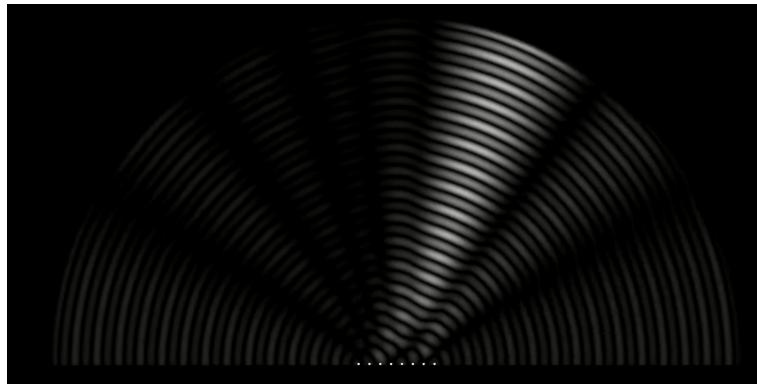
Plus nous augmentons le nombre d'éléments rayonnants, plus le faisceau devient étroit, mais des lobes secondaires faibles apparaissent. En général, les différentes antennes sont séparées d'une demi-longueur d'onde.



— La disposition spatiale des antennes.

La disposition spatiale des éléments rayonnants va également modifier le schéma des lobes de réseau et des lobes secondaires.

Ensuite, pour pouvoir changer la direction de ce faisceau, les antennes actives ajustent individuellement le retard ou la phase de chaque antenne :



En ajustant les retards ou les avances de phase de chaque signal reçus par les différentes antennes, nous pouvons modifier la direction du faisceau pour maximiser le signal émis ou reçu.

Pour résumer, la géométrie du réseau d'antennes crée le motif du faisceau, tandis que le décalage de phase oriente ce faisceau.

Le poids (weight), ou facteur w , représente à la fois le gain et directement la phase de chaque antenne. Ainsi, nous avons la possibilité de former un faisceau en fonction de ce facteur de poids pour chaque antenne. Habituellement, la forme du faisceau et sa taille sont définies par les amplitudes des éléments rayonnants, et l'orientation du faisceau s'obtient par la création d'un plan de retard ou de phase.

4.2 Modélisation d'une antenne active

Nous allons modéliser ici l'antenne active « phased array ». Le script python correspondant à cette section est "ModelisationAntenneActiveDash.py".

Pour ce faire, nous allons utiliser Dash.

Dash est une bibliothèque en Python développée par Plotly pour créer des applications web analytiques interactives. Cela nous permettra de créer et de visualiser des tableaux et des graphiques en temps réel. Nous commençons d'abord par générer les positions des éléments de l'antenne. Ensuite, nous savons que chaque élément rayonnant émet un signal identique en amplitude, mais avec des phases contrôlables individuellement. Cela permet de diriger le faisceau émis dans une direction spécifique.

Le signal émis par chaque élément peut être décrit par une onde sinusoïdale de la forme :

$$s(t) = A \cos(2\pi ft + \phi)$$

Dans le cas d'un phased array, la fréquence f et l'amplitude A sont généralement les mêmes pour tous les éléments (bande passante étroite), tandis que la phase ϕ est ajustée individuellement.

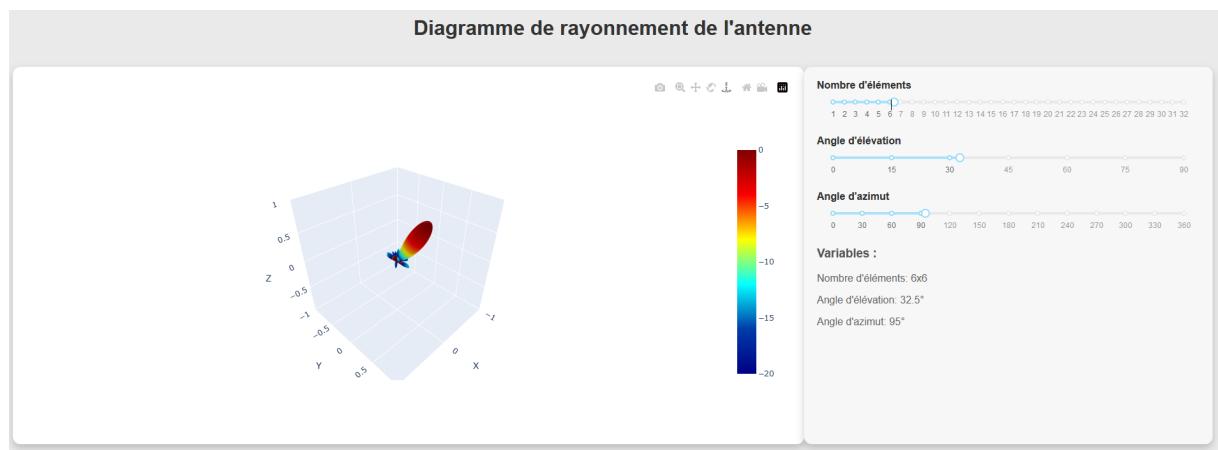
Pour notre géométrie d'antenne, qui est un carré de 16 par 16 (Cette valeur est ajustable dans le code), les éléments sont espacés par une demi longueur d'onde entre eux. La phase supplémentaire $\Delta\phi$ introduite pour chaque élément situé à une position (x, y) est calculée comme suit :

$$\Delta\phi(x, y) = k \cdot (x \sin \theta \cos \phi + y \sin \theta \sin \phi)$$

où λ est la longueur d'onde du signal et k le nombre d'onde, défini par :

$$k = \frac{2\pi}{\lambda}.$$

Pour finir, il nous suffit juste d'afficher le diagramme de rayonnement à l'aide de Dash. Voici à quoi ressemble l'application web :



En modifiant en temps réel, à l'aide des curseurs, les valeurs des angles d'attaque, le diagramme de rayonnement s'actualise. Nous pouvons aussi modifier le nombre d'éléments actifs sur l'antenne.

4.3 Steering en temps / Steering en phase

Dans le code précédent, nous ajustons les phases de chaque signal pour diriger le faisceau. Cependant, pour certains signaux à large bande, ce contrôle de phase peut introduire des distorsions de phase à différentes fréquences, compromettant ainsi la fidélité du signal. Une méthode alternative pour diriger le faisceau consiste à utiliser le contrôle du retard en temps (time delay) plutôt que le déphasage (phase shift). Le retard en temps permet de manipuler les signaux de manière uniforme sur toutes les fréquences.

Dans une approche par retard en temps, au lieu de décaler la phase des signaux émis par chaque élément de l'antenne, nous retardons ou avance chaque signal dans le temps. Ce retard est ajusté pour chaque élément en fonction de sa position et de la direction désirée du faisceau. Chaque élément de l'antenne émet un signal du type :

$$s(t) = A \cos(2\pi f(t - \tau(x, y)))$$

Pour diriger le faisceau dans une direction spécifique, définie par un angle d'élévation θ et un angle d'azimut φ , le retard en temps $\tau(x, y)$ pour chaque élément situé à la position (x, y) est calculé de la manière suivante :

$$\tau(x, y) = \frac{x \sin \theta \cos \varphi + y \sin \theta \sin \varphi}{c}$$

L'approche du retard en temps possède de nombreux avantages comparés au phase shift, mais c'est toutefois plus complexe et plus coûteux à implémenter.

4.4 Le phénomène de beamsquint

Le beamsquint est un phénomène rencontré dans les antennes actives lorsqu'on utilise des déphasages pour orienter un faisceau d'ondes avec des signaux large bande.

Lorsqu'une onde arrive sur une antenne, il y a des décalages temporels dans la réception de l'onde entre les différents éléments en fonction de l'angle d'incidence. Pour diriger le faisceau dans une direction spécifique, nous compensons ce décalage par un déphasage entre les signaux des différents éléments. Ce procédé fonctionne bien pour des signaux à bande étroite. Cependant, avec des signaux large bande, le déphasage requis pour orienter le faisceau varie avec la fréquence, ce qui fait que la direction du faisceau change en fonction de la fréquence du signal, causant le phénomène de beam squint.

Cela pose des défis pour les systèmes large bande, car le faisceau ne reste pas stable dans la direction souhaitée, réduisant ainsi l'efficacité du système. Pour pallier à ce problème, nous utilisons un steering en temps. En effet, un retard temporel est indépendant de la fréquence. En appliquant un retard temporel pour compenser le décalage de temps entre les éléments d'antenne, la direction du faisceau reste cohérente pour toutes les fréquences du signal large bande, éliminant ainsi le problème de beam squint. Essayons de mathématiser tout ça pour comprendre un peu mieux :

Si le signal est représenté par une fonction temporelle $s(t)$, après un retard temporel τ , le signal devient $s(t - \tau)$. La transformée de Fourier du signal retardé $s(t - \tau)$ est donnée par :

$$S(f) = S_0(f) e^{-j2\pi f \tau}$$

Le terme $e^{-j2\pi f \tau}$ représente un déphasage qui dépend linéairement de la fréquence. Cependant, le retard temporel τ reste constant, peu importe la fréquence. Cela signifie que le retard temporel τ est le même pour toutes les fréquences, ce qui garantit que le faisceau reste aligné pour toutes les fréquences. Maintenant, voyons ce que donne un steering en phase. Après l'application d'un déphasage ϕ , le signal devient $s(t)e^{j\phi}$. Si ϕ est un déphasage constant, alors la relation entre ce déphasage et la fréquence est donnée par :

$$\phi(f) = 2\pi f \cdot \tau_{\text{equiv}}$$

car la transformée de Fourier de ce signal sera :

$$S(f) = S_0(f) e^{-j2\pi f(\frac{\phi}{2\pi f})}$$

Le terme τ_{equiv} n'est pas un vrai retard temporel appliqué directement, mais il peut être compris comme le retard temporel que produirait un déphasage de ϕ pour une fréquence spécifique. Par exemple, pour une fréquence plus élevée, un même déphasage correspondrait à un retard temporel plus court, et vice versa. Ainsi, pour une fréquence différente, le même déphasage ϕ se traduit par un angle de phase différent, ce qui peut provoquer un décalage de faisceau (beam squint) lorsque des signaux large bande sont utilisés.

La formule pour calculer cette déviation est :

$$\Delta\theta = \arcsin\left(\frac{f_0}{f} \sin \theta_0\right) - \theta_0$$

où :

- θ_0 est l'angle de direction du faisceau pour la fréquence de référence f_0 .
- f_0 est la fréquence de référence.
- f est la nouvelle fréquence pour laquelle nous calculons la déviation du faisceau.
- $\Delta\theta$ est la déviation angulaire du faisceau, ou beam squint.

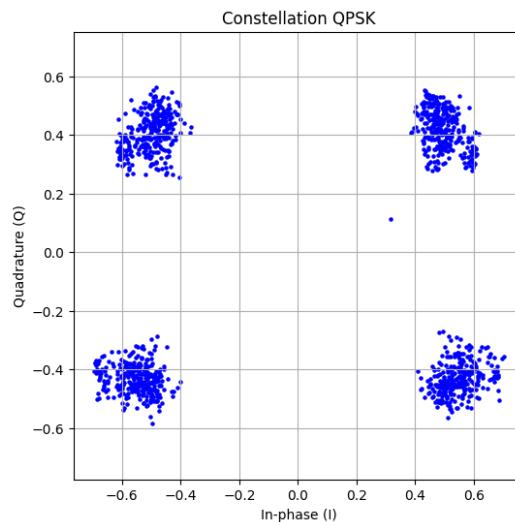
En résumé, le beam squint est un effet indésirable dans les systèmes large bande car il peut entraîner une mauvaise directionnalité du faisceau, affectant ainsi la performance du système. Dans la section suivante nous allons essayer de mettre en avant ce phénomène de beamsquint.

4.5 Simulation beamsquint

Le script python correspondant à cette section est "Beamsquint.py".

Le but de ce code est de montrer qu'un steering en phase est affecté par le phénomène du beamsquint. Pour ce faire, le code de cette partie va simuler une antenne active sous deux forme : une avec un steering en phase, et une autre avec un steering en temps.

Premièrement, nous devons créer le signal QPSK que va envoyer l'antenne. Pour cela nous reprenons simplement les codes des sections précédentes. Nous le numérisons ensuite avec un ADC $\Delta\Sigma$ et voici sa constellation suivi de son SNR :



```
In [10]: M2M4(symbols_out_demod_egal,1)
Out[10]: 18.411406602357978
```

Il nous faut maintenant faire un décalage en phase et en temps pour les différents éléments à un angle fixé. Pour la phase, nous multiplions simplement le signal par un facteur correspondant aux bons angles et à la géométrie de l'antenne. Pour le signal en temps, nous le retardons / avançons en fonction des paramètres de la simulation.

Ensuite, nous choisissons un point au sol qui est défini par les variables dU et dV, donnant par la suite les valeurs des angles theta et phi. Cette partie GRD va permettre de visualiser la QPSK à ce point précis :

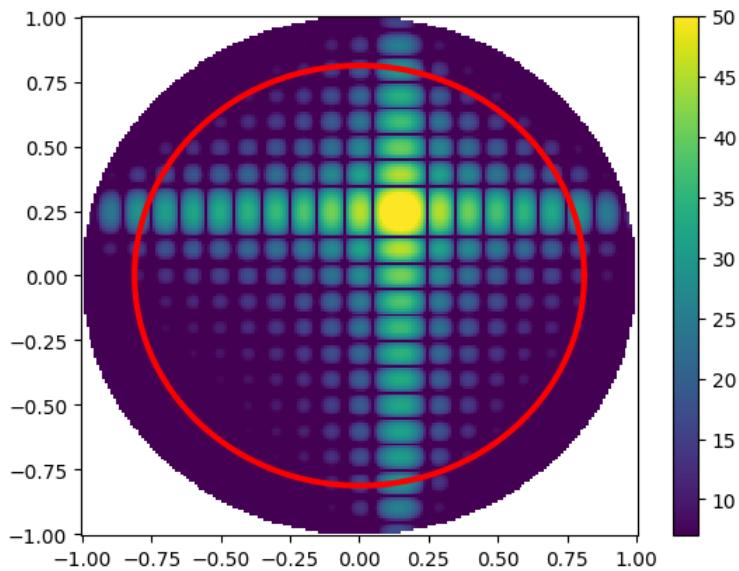


FIGURE 10 – GRD avec $dU = 0.2$ et $dV = -0.5$

Enfin, pour visualiser la QPSK au point ciblé, nous devons retarder/avancer nos signaux des différents éléments de l'antenne pour simuler le fait que les signaux n'arrivent pas tous en même temps. Nous implémentons alors un retard en temps pour les deux formes : que ce soit en phase, ou en temps.

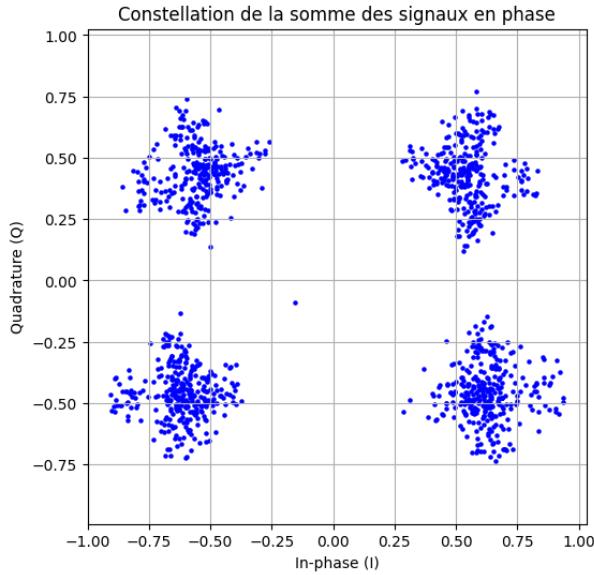


FIGURE 11 – Constellation du signal recu en phase : SNR = 13.22

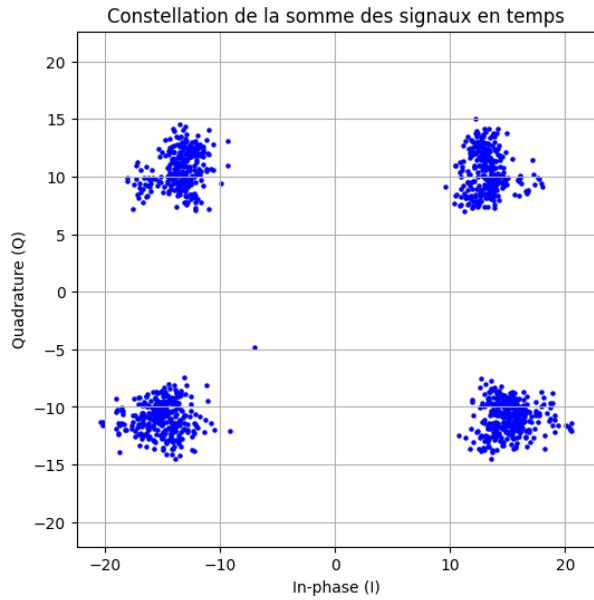
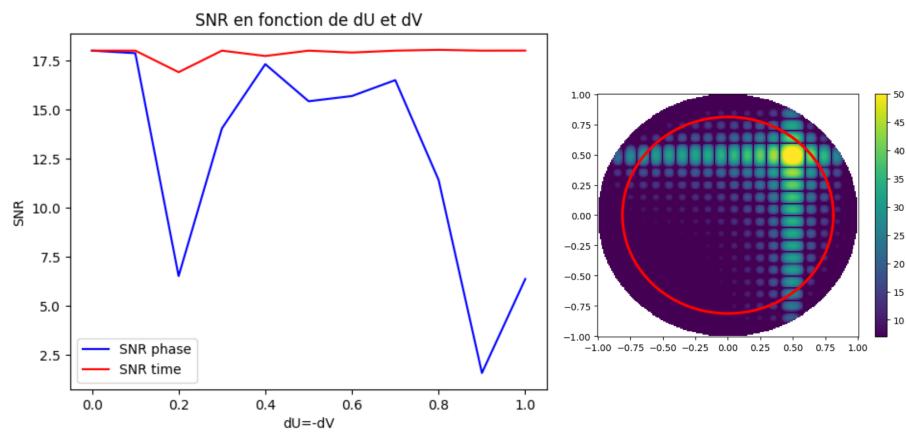
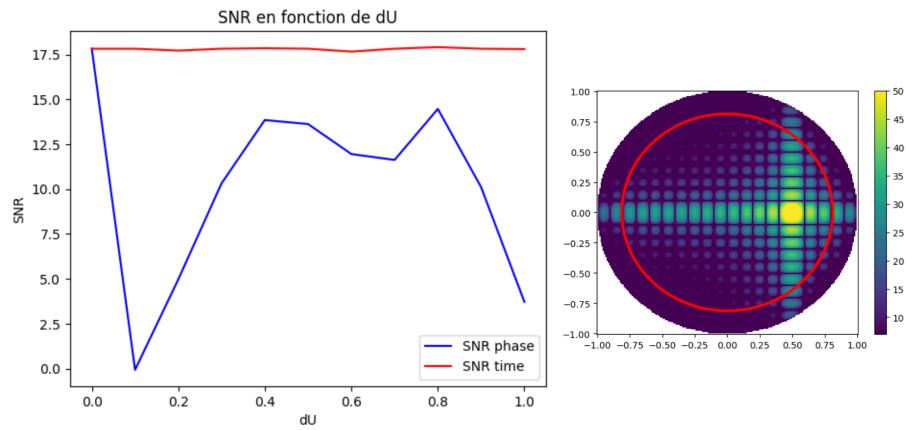
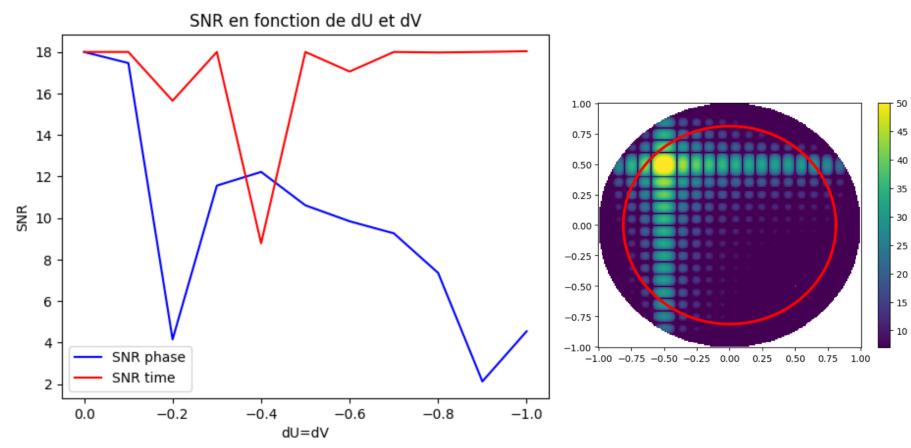
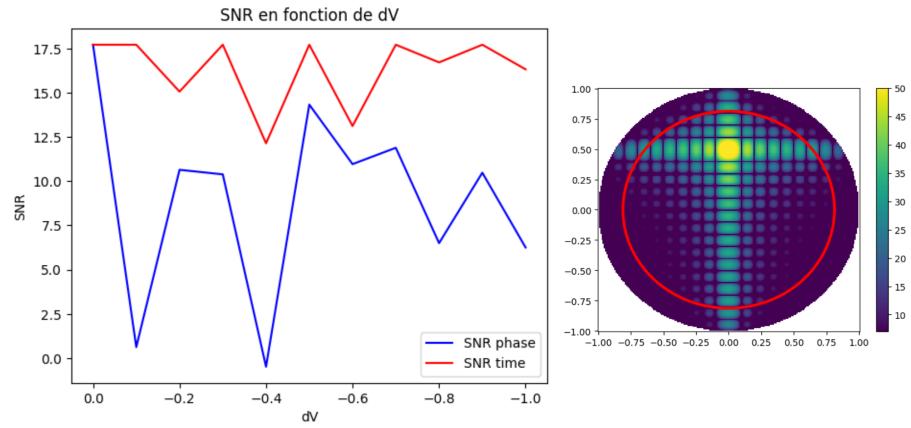


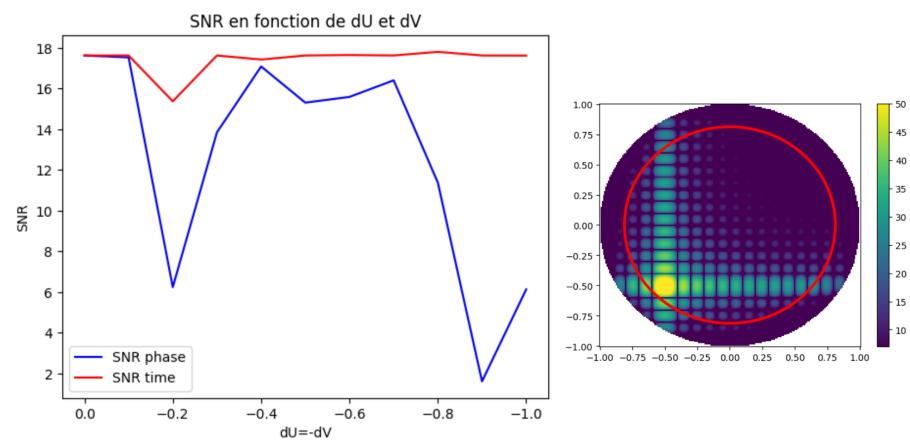
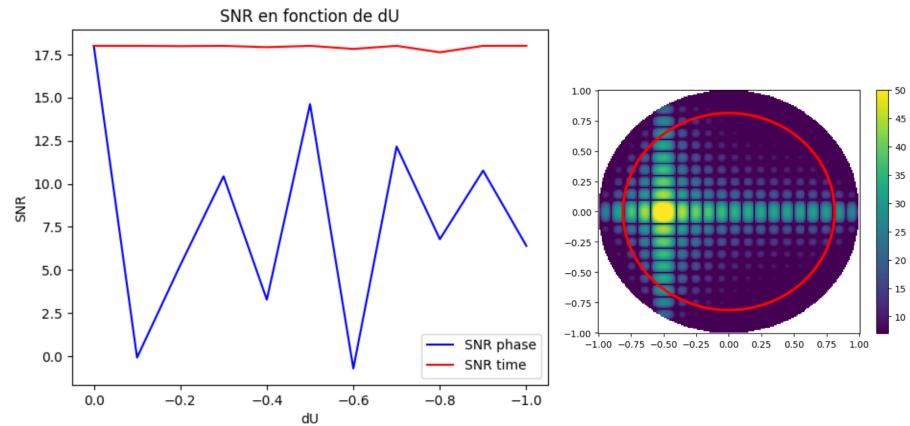
FIGURE 12 – Constellation du signal recu en temps : SNR = 16.98

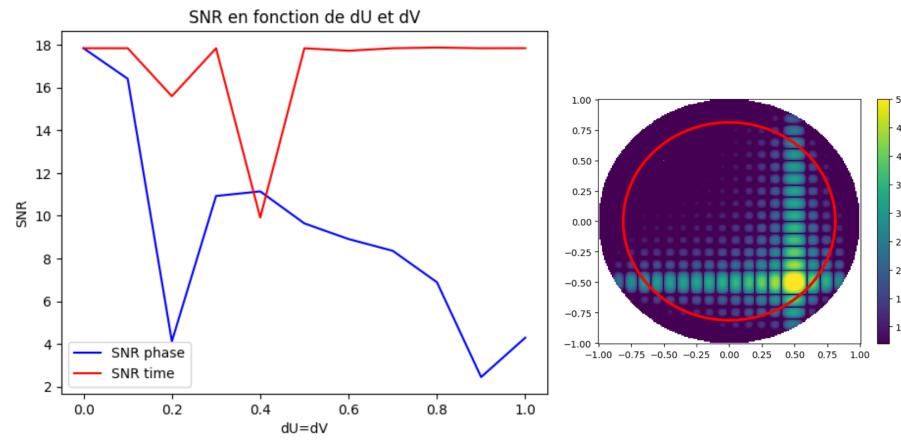
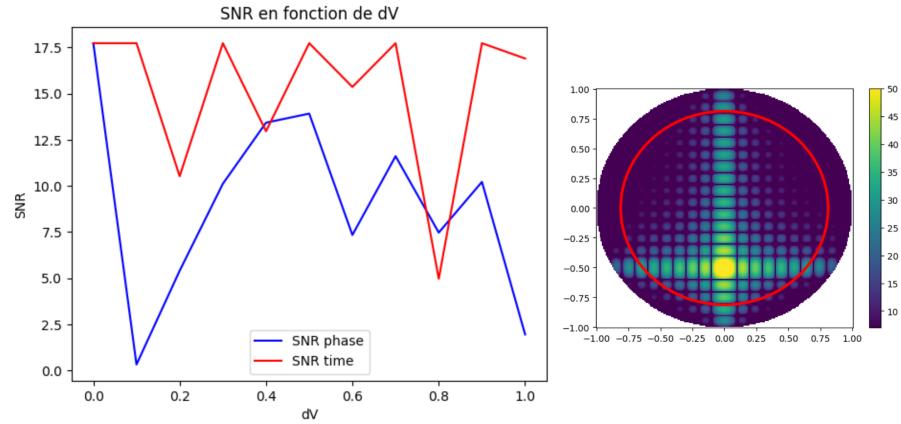
On remarque déjà que le steering en phase est moins bon que celui en temps. Pour vérifier cela, nous allons tracer la courbe du SNR en fonction des positions dU et dV. La cible à droite de chaque courbe représente le dernier point calculé de la courbe. Il permet de visualiser l'axe/la direction dans lequel nous calculons le SNR.

Voici ce que nous obtenons :









Dans un premier temps, nous voyons que le steering en phase est dans chaque cas moins bon que celui en temps. De plus, nous pouvons vérifier que dans la plupart des cas, le SNR diminue avec la distance dU et dV. Nous remarquons une chute du SNR aux alentours de 0.2 pour dU et dV pour tous les cas. Nous n'avons pas eu suffisamment de temps pendant le stage pour déterminer la cause des points hors famille. Par exemple, la méthode M2M4 utilisée pour l'estimation du SNR

est assez sensible et la valeur obtenue peut être momentanément assez basse à cause de seulement quelques échantillons. L'utilisation de la corrélation avec les symboles émis pourrait permettre de stabiliser le système de mesure du SNR. Néanmoins la tendance générale est cohérente avec la théorie.

Vérifions maintenant la cohérence de notre code en traçant le SNR en fonction de la bande passante sur deux positions différentes. Voici ce que nous obtenons :

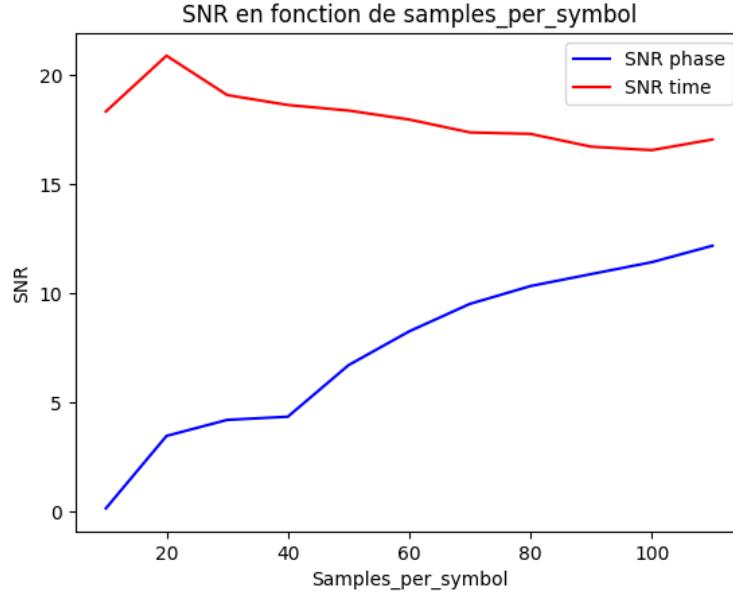


FIGURE 13 – $dU = 0.3$ et $dV = 0.5$

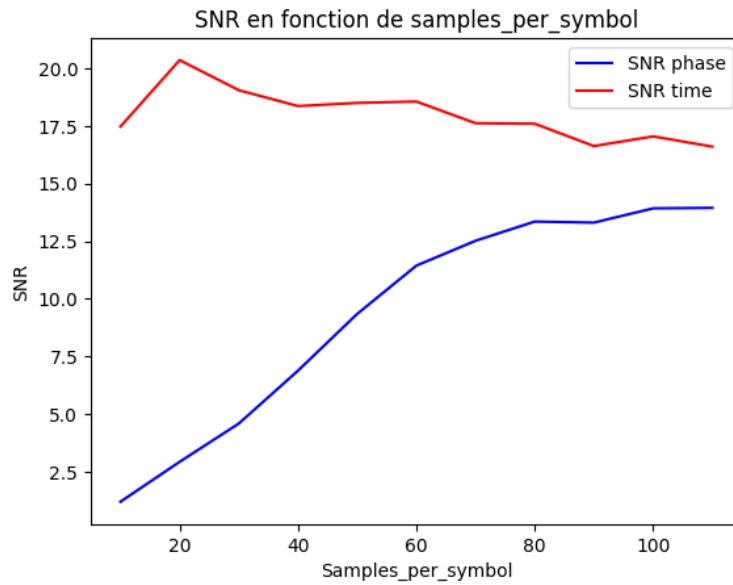


FIGURE 14 – $dU = -0.6$ et $dV = -0.3$

En ce qui concerne le steering en temps, celui-ci ne semble pas être affecté par la variation

de la bande passante. En revanche, pour le steering en phase, nous observons que l'augmentation du nombre d'échantillons par symbole, donc la réduction de la bande passante, conduit à une amélioration du SNR. Cela s'explique par le fait que, comme évoqué précédemment, le retard ajouté aux signaux dans le steering en phase dépend de la fréquence. Ainsi, une bande passante plus étroite réduit la variation de déphasage entre les différentes fréquences, limitant les distorsions et améliorant le rapport signal sur bruit.

5 Conclusion

Ce rapport a exploré en profondeur le fonctionnement et les applications des convertisseurs analogique-numérique Delta-Sigma ainsi que des antennes actives. Les ADC $\Delta\Sigma$ se distinguent par leur capacité à offrir une haute résolution et une réduction efficace du bruit dans la bande de base, bien qu'ils présentent des défis en termes de complexité et de latence. La modélisation pratique en Python et GNU Radio a permis de démontrer leur performance avec différents types de signaux.

En parallèle, l'étude des antennes actives a mis en lumière l'importance du beamforming pour optimiser les communications spatiales. La modélisation et la simulation ont montré les avantages du steering en temps par rapport au steering en phase, notamment pour éviter le phénomène de beamsquint.

6 Sources

- Texas Instruments, "Understanding Data Converters," Application Report SLYT423A, accessed from :
<https://www.ti.com/lit/an/slyt423a/slyt423a.pdf>
- Texas Instruments, "Understanding Delta-Sigma Modulators," Application Report SLYT438, accessed from :
<https://www.ti.com/lit/an/slyt438/slyt438.pdf>
- Wikipedia, "Delta-sigma modulation," accessed from :
https://en.wikipedia.org/wiki/Delta-sigma_modulation
- Analog Devices, "Sigma-Delta ADC Tutorial," accessed from :
<https://www.analog.com/en/resources/interactive-design-tools/sigma-delta-adc-tutorial.html>
- Monolithic Power, "Delta-Sigma ADCs," accessed from :
<https://www.monolithicpower.com/en/learning/mpscholar/analog-to-digital-converters/detailed-analysis-of-adc-architectures/delta-sigma-adcs>
- Matlab youtube video, "What Are Phased Arrays?" accessed from :
https://www.youtube.com/watch?v=9WxWunOE-PM&list=WL&index=8&t=22s&ab_channel=MATLAB
- Analog Dialogue, "Phased Array Antenna Patterns—Part 2 : Grating Lobes and Beam Squint" accessed from :
<https://www.analog.com/media/en/analog-dialogue/volume-54/number-2/phased-array-antenna-patterns-part-2-grating-lobes-and-beam-squint.pdf>
- mwrf.com, "Phased-Array Antenna Patterns (Part 5)—Beam Squint" accessed from :
<https://www.mwrf.com/technologies/embedded/systems/article/21142402/analog-devices-phased-array-antenna-patterns-part-5beam-squint>

7 Annexe : Explication de la méthode M2M4 pour le calcul du SNR

La méthode M2M4 est une technique statistique utilisée pour estimer le rapport signal sur bruit (SNR) à partir de moments statistiques de second et de quatrième ordre des symboles reçus. Voici une explication détaillée de la fonction M2M4 implémentée en Python, accompagnée d'une explication des concepts mathématiques sous-jacents.

1. Moments statistiques

Dans le contexte de la méthode M2M4, deux moments statistiques jouent un rôle clé :

- **M2** : le moment d'ordre 2, correspondant à la puissance moyenne des symboles reçus, c'est-à-dire la moyenne du carré de la norme des symboles :

$$M2 = \mathbb{E}[|X|^2]$$

- **M4** : le moment d'ordre 4, qui donne une mesure de la "forme" des symboles reçus, via la moyenne de la quatrième puissance de la norme des symboles :

$$M4 = \mathbb{E}[|X|^4]$$

Ces moments sont calculés directement à partir des symboles reçus après suppression des pilotes (`RxSymbolsWithoutPilots`), qui sont utilisés pour estimer le SNR.

2. Kurtosis du signal

La **kurtosis** est un indicateur statistique mesurant la "pointedness" de la distribution d'une variable aléatoire. Pour les signaux numériques modulés, la kurtosis peut varier en fonction de la modulation utilisée. Dans la méthode M2M4, on utilise la kurtosis du signal, `signal_kurtosis`, une valeur caractéristique du type de modulation, comme une entrée importante pour l'estimation du SNR.

3. Principe de la méthode

L'estimation du SNR se base sur la séparation entre la composante de puissance du signal et celle du bruit, en utilisant les moments M2 et M4. Le bruit, dans un canal de communication, est supposé avoir une kurtosis de 2, qui est une caractéristique du bruit gaussien.

Voici les étapes détaillées du calcul :

1. **Calcul des moments M2 et M4** : La fonction commence par calculer la moyenne des moments d'ordre 2 et 4 des symboles reçus :

```
M2 = np.mean(np.abs(RxSymbolsWithoutPilots)**2)
M4 = np.mean(np.abs(RxSymbolsWithoutPilots)**4)
```

2. **Calcul de la puissance estimée du signal (Se)** : Le signal et le bruit sont séparés à partir d'une expression non-linéaire des moments M2 et M4, prenant en compte la kurtosis du signal et celle du bruit :

```
noise_kurtosis = 2
tmp = (4 - signal_kurtosis * noise_kurtosis) * M2**2 +
      M4 * (signal_kurtosis + noise_kurtosis - 4)
Se = np.sqrt(tmp) / (signal_kurtosis - 2)
```

3. **Estimation de la puissance du bruit (Ne)** : La puissance du bruit est ensuite estimée en soustrayant la puissance du signal (Se) de la puissance totale (M2).

```
Ne = np.abs(M2 - Se)
```

4. **Calcul du SNR estimé :** Enfin, l'estimation du SNR est obtenue en prenant le rapport entre la puissance du signal et celle du bruit, puis en l'exprimant en décibels (dB) :

```
EstimatedSNR = 10 * np.log10(Se / Ne)
```

4. Avantages de la méthode M2M4

La méthode M2M4 présente plusieurs avantages :

- Elle permet une estimation du SNR sans connaissance a priori de la puissance du bruit.
- Elle ne nécessite pas de symbole pilote ou d'information supplémentaire pour le calcul du SNR.
- Elle est relativement simple à implémenter et rapide à calculer à partir des symboles reçus. Cependant, elle repose sur certaines hypothèses, notamment la gaussienneté du bruit et la constance de la kurtosis du signal, qui peuvent limiter son efficacité dans certains environnements complexes.

Ainsi, la méthode M2M4 utilise des propriétés statistiques simples des symboles reçus pour estimer le rapport signal sur bruit, ce qui en fait une approche pratique et largement utilisée dans les systèmes de télécommunication.

```
"""
M2M4 SNR estimator
"""

def M2M4(RxSymbolsWithoutPilots,signal_kurtosis):
    """

    Parameters
    -----
    RxSymbolsWithoutPilots : TYPE
        DESCRIPTION.
    signal_kurtosis : TYPE
        DESCRIPTION.

    Returns
    -----
    EstimatedSNR : TYPE
        DESCRIPTION.

    """
    M2 = np.mean(np.abs(RxSymbolsWithoutPilots)**2)
    M4 = np.mean(np.abs(RxSymbolsWithoutPilots)**4)
    noise_kurtosis = 2
    tmp = (4 - signal_kurtosis * noise_kurtosis) * M2 * M2 + M4 * (signal_kurtosis + noise_kurtosis - 4)
    tmp = np.sign(tmp) * tmp
    Se = np.sqrt(tmp) / (signal_kurtosis - 2)
    Se = np.sign(Se) * Se
    Ne = np.abs(M2 - Se)
    EstimatedSNR = 10 * np.log10(Se / Ne)
    return EstimatedSNR
```

FIGURE 15 – Fonction M2M4 dans usual.py