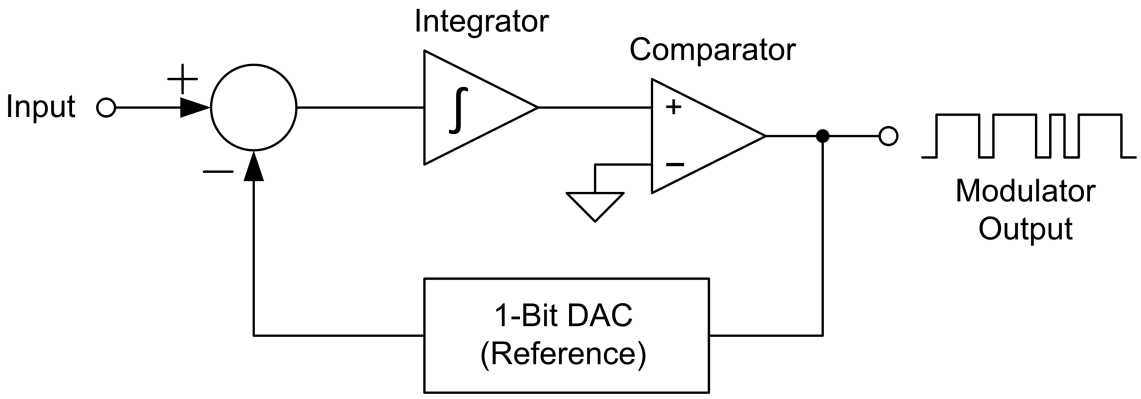
Satconsult
Member of Euroconsult Group

ADC $\Delta\Sigma$ et Antenne Active



Bourlatchka Ivan
2024

Table des matières

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 mégahertz. De plus, ils possèdent une plus grande largeur de bande que les ADC classiques et offrent une dynamique accrue.

Dans ce rapport, nous allons essayer de comprendre le fonctionnement de ce convertisseur, ses principales caractéristiques, et son implémentation sur une antenne active.

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 :

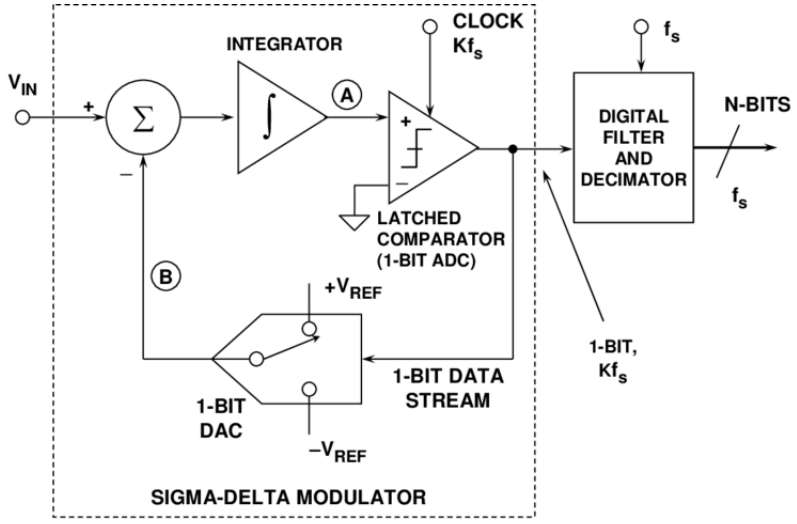


FIGURE 1 – Schéma basique d'un ADC $\Delta\Sigma$ d'ordre 1

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.

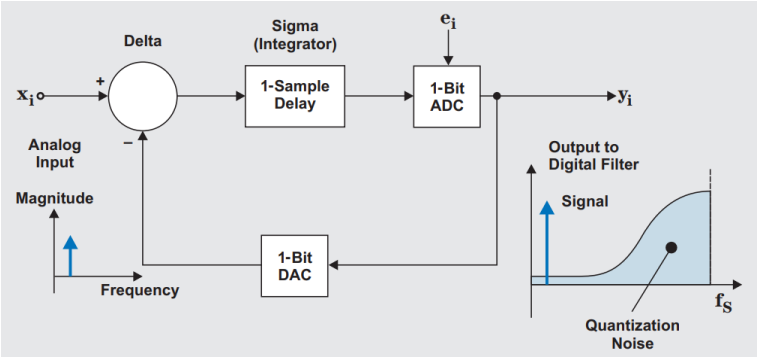


FIGURE 2 – ADC $\Delta\Sigma$ du premier ordre dans le domaine fréquentiel

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.

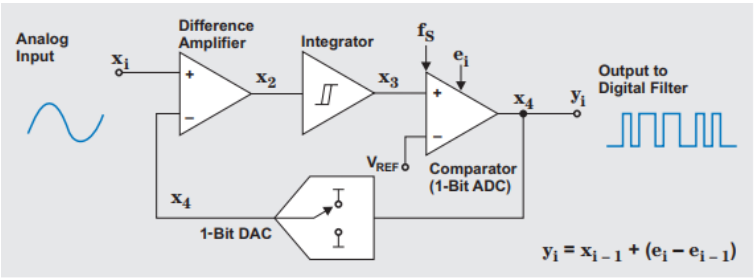


FIGURE 3 – ADC $\Delta\Sigma$ du premier ordre dans le domaine temporel

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 :

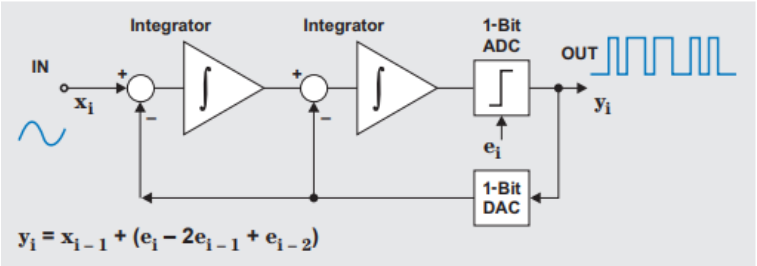


FIGURE 4 – ADC $\Delta\Sigma$ du second ordre

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 :

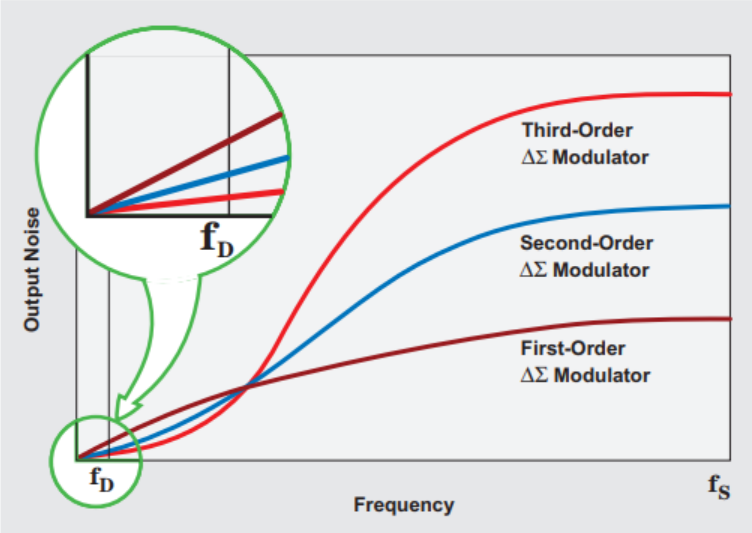


FIGURE 5 – Noise shaping en fonction de l'ordre du modulateur

Finalement, le modulateur de l'ADC ΔΣ 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 ΔΣ.

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.

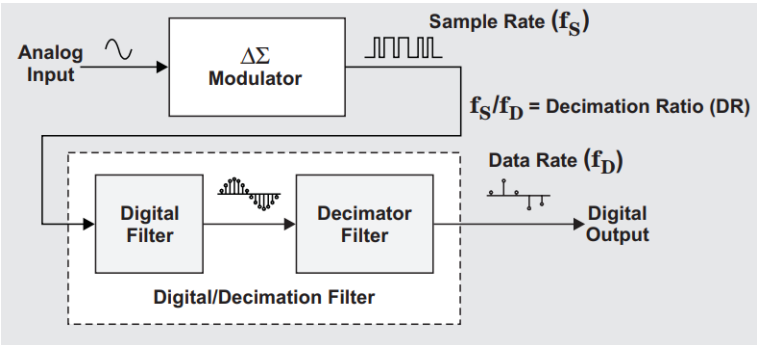


FIGURE 6 – Diagramme bloc complet

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 ΔΣ. 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 ΔΣ. 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.

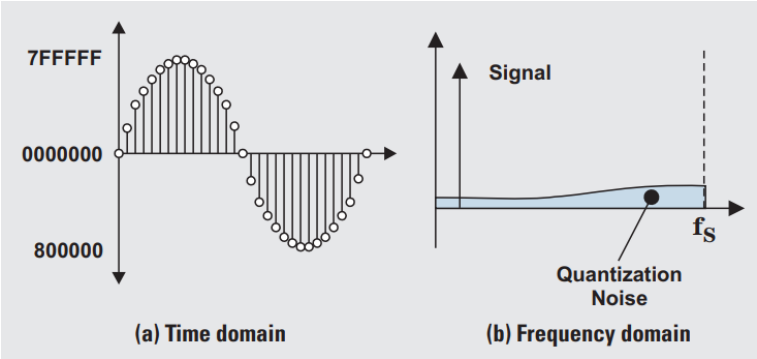


FIGURE 7 – Signal en sortie du filtre (Entrée sinusoïdale dans le modulateur)

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.

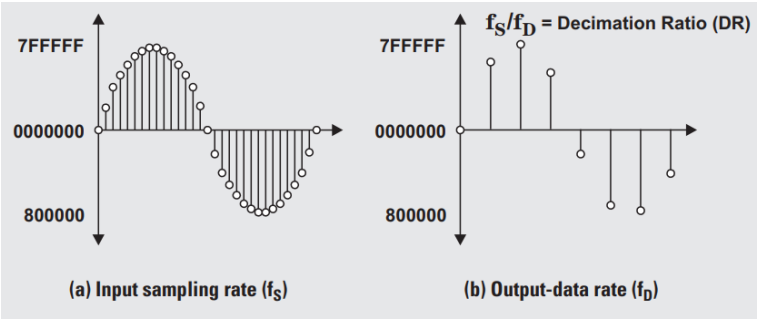


FIGURE 8 – Principe de décimation

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.
— **Coût :** 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

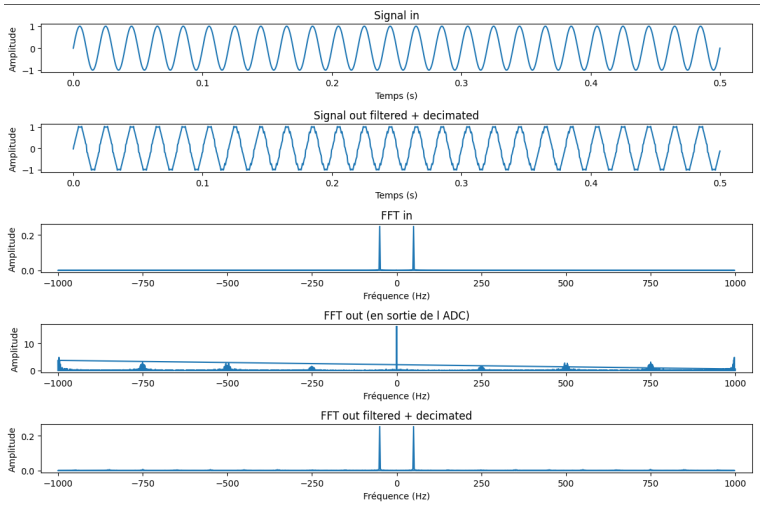
Le script python correspondant à cette section est `"Premiere_partie_sinus.py"`.

Dans cette section, on va 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. On commence par le suréchantillonner, puis ensuite on rentre dans la boucle. Dans cette boucle on applique le même traitement pour tous les échantillons de données : on différencie avec l'ancienne sortie, on intègre, on quantifie et on compare avant de le rebalancer dans un DAC pour qu'il retourne dans la boucle. Voici notre fonction ADC Delta Sigma en python :

```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

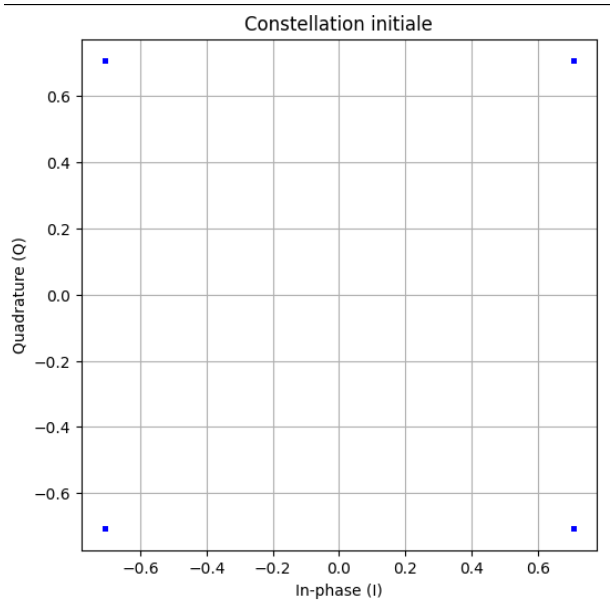
Le script python correspondant à cette section est "`Premiere_partie_QPSK.py`".

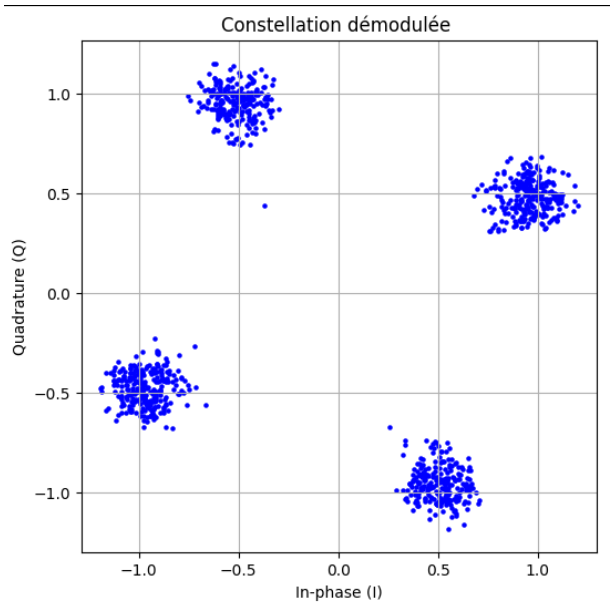
Dans cette section on va essayer de modéliser simplement le passage d'un signal QPSK dans un adc delta sigma et comparer les différentes constellations afin de vérifier la bonne modélisation mise en place.

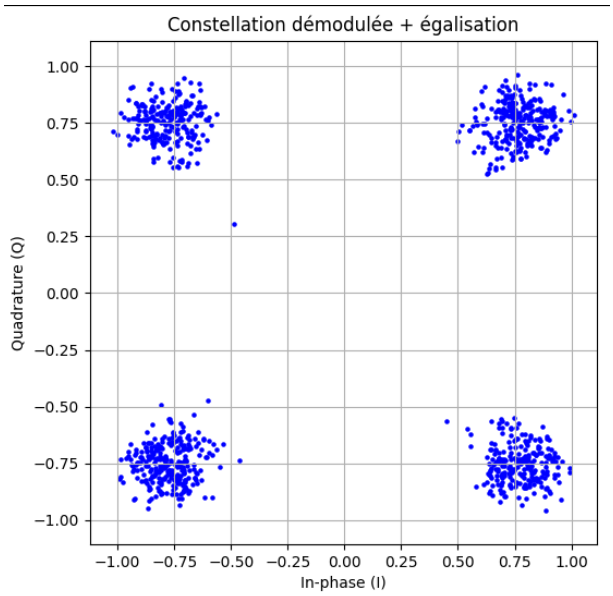
On commence par créer un signal QPSK disponible dans usual.py, que l'on bruite avant de le mettre sur une porteuse. Ensuite on fait comme dans la partie précédente, on fait rentrer le signal

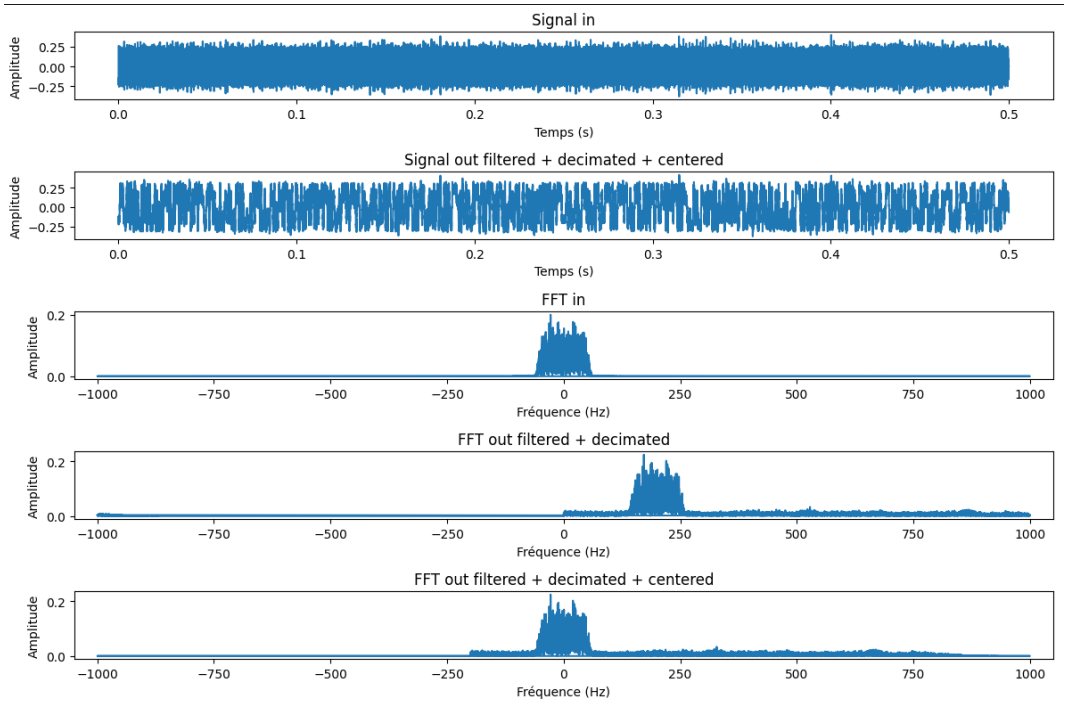
dans l'adc, et on récupère la sortie dans la variable `signal_out`. nous appliquons par la suite tout le traitement nécessaire à la démodulation du signal afin de retrouver la constellation initiale. La fonction `demodulator_non_data_aided` ne fonctionnant pas sur mon code, il a fallu démoduler à la main en trouvant le premier échantillon qui allait maximiser le SNR. Il a aussi fallu rajouter un peu d'égalisation.

Voici les résultats :





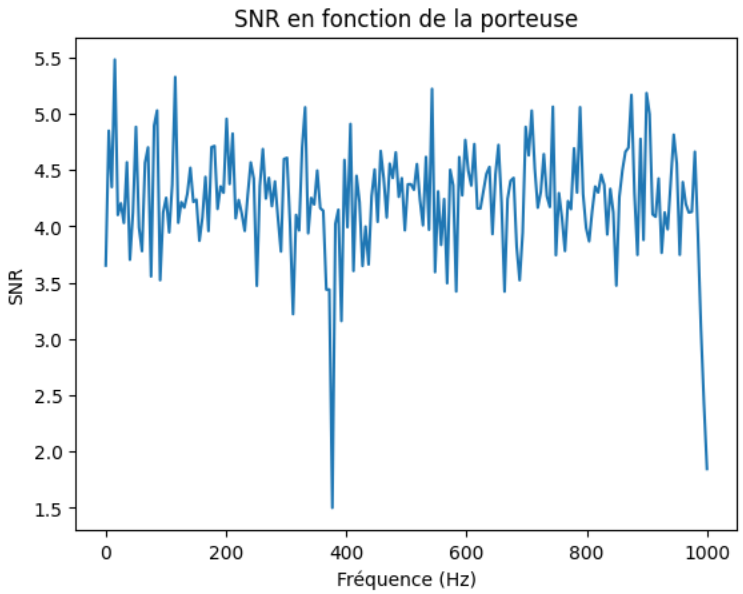




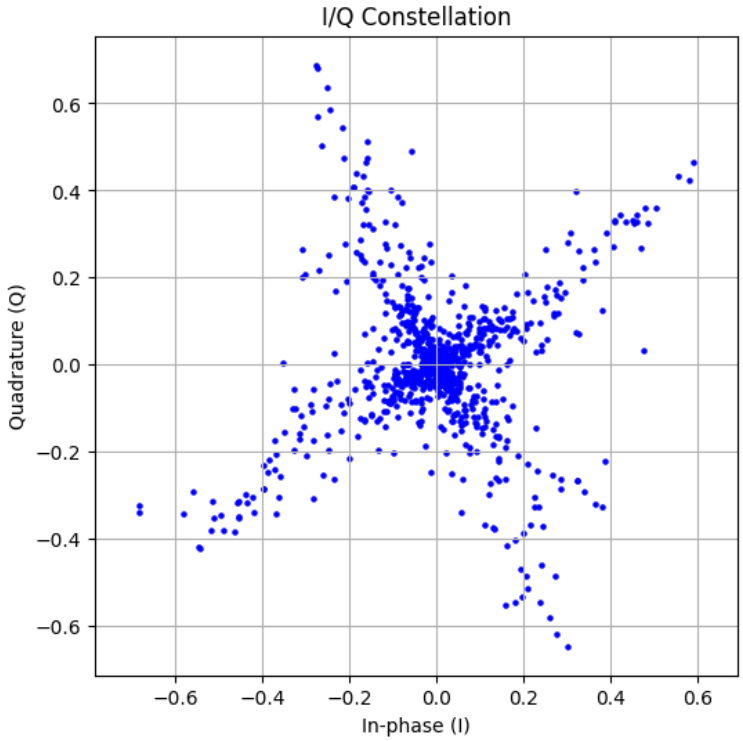
3.1.3 SNR en fonction de la porteuse

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

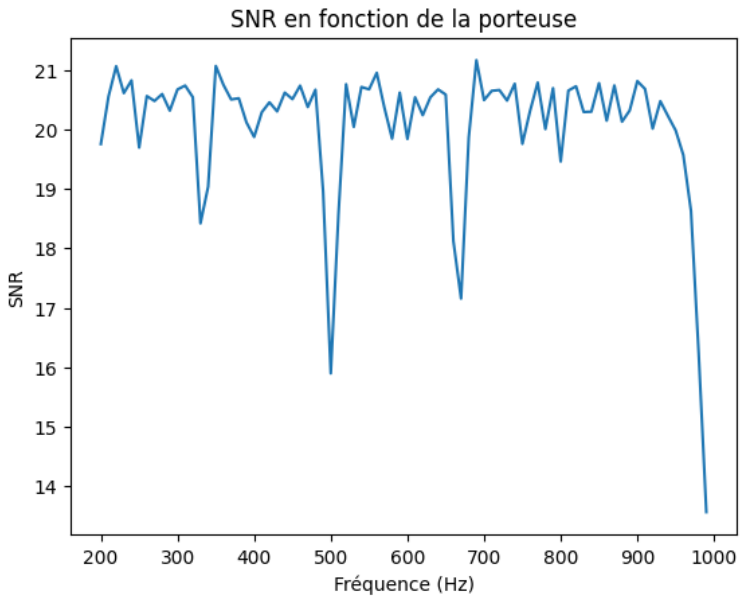
On essaye maintenant de tracer la courbe du SNR en fonction de la porteuse utilisée. On commence par diminuer la taille de la bande passante du signal et pour ce faire, on augmente simplement le nombre d'échantillons par symbole. On trace la courbe, et voici le résultat :

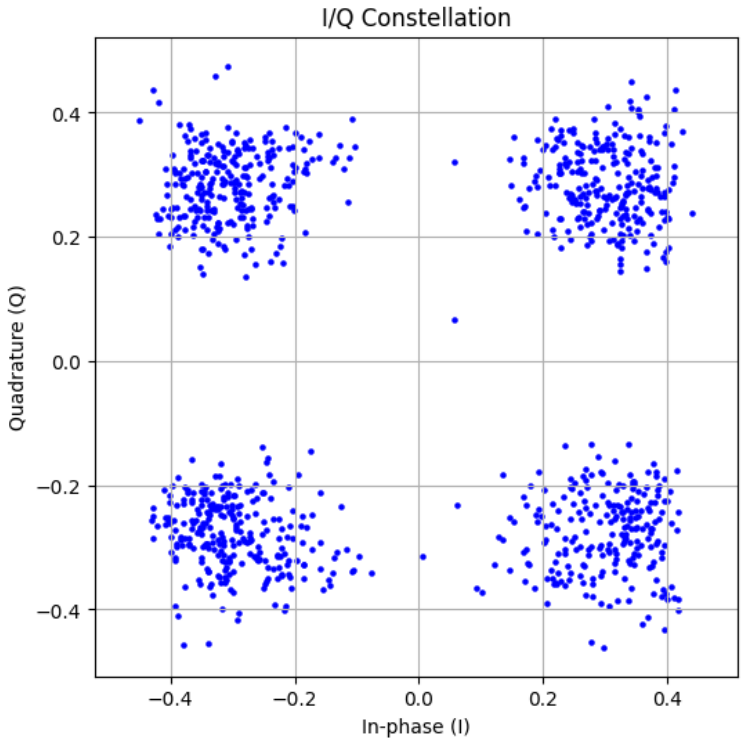


On remarque que le SNR est vachement bas. Alors en traçant la constellation pour vérifier le bon fonctionnement de l'adc, on obtient ceci :



En fait, ceci est dû au fait que la quantification n'est pas assez précise. En passant de enob = 3 à enob = 4 on obtient un meilleur résultat :





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 .

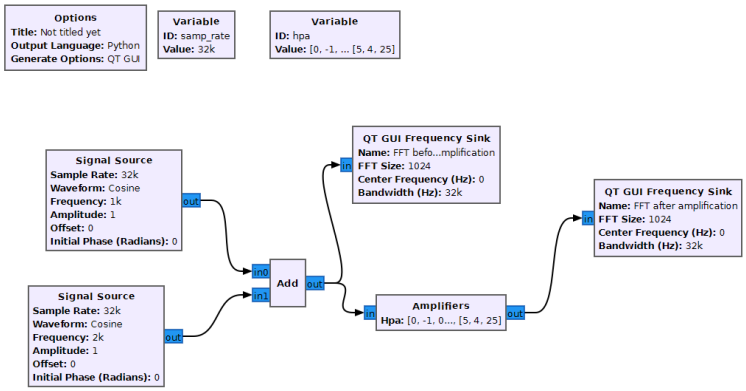
3.2 Modélisation GNU radio

3.2.1 Premiers pas : la fonction amplifiers

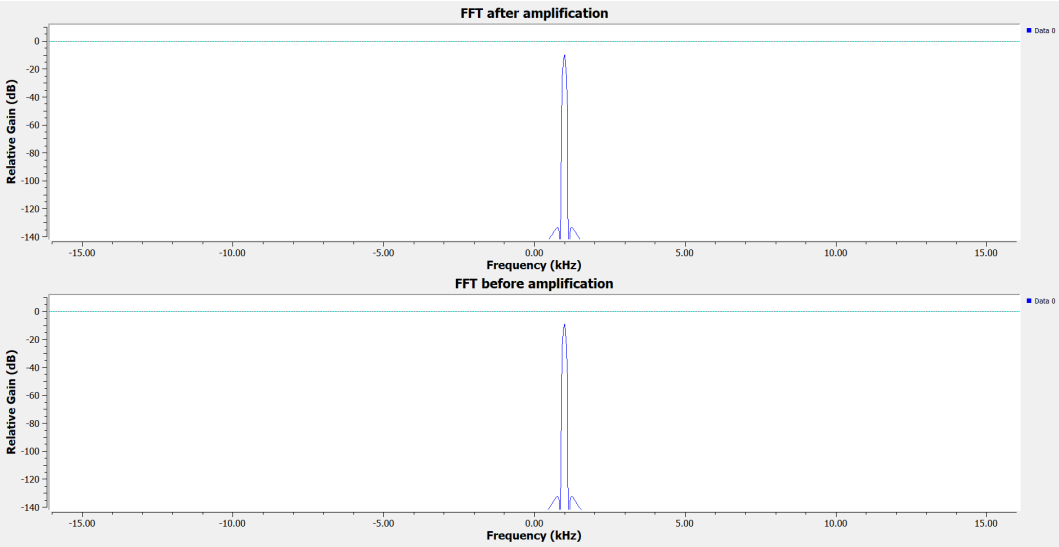
Nous allons maintenant essayer de modéliser tout ça dans GNU radio. Mais pour l'instant, il faut d'abord se familiariser avec l'outil. Il se compose de plusieurs blocs déjà présents dans le logiciel, mais il permet aussi de créer nos propres blocs. Pour ce faire, nous allons essayer d'implémenter une 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" :

```python
import numpy as np
from gnuradio import gr

class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self, hpa=1):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Amplifiers',   # will show up in GRC
            in_sig=[np.complex64],
            out_sig=[np.complex64]
        )
        self.hpa = np.array(hpa)

    def amplifier(self, signal_in):
        ibo_range = self.hpa[:, 0]
        obo_range = self.hpa[:, 1]
        phase_range = self.hpa[:, 2]
        signal_in_am = 20 * np.log10(np.abs(signal_in))
        signal_in_pm = np.angle(signal_in)
        signal_out_am = np.interp(signal_in_am, ibo_range, obo_range)
        signal_out_pm = np.interp(signal_in_am, ibo_range, phase_range) * np.pi / 180
        signal_out = 10**(signal_out_am / 20) * np.exp(1j * (signal_in_pm + signal_out_pm))

        return signal_out

    def work(self, input_items, output_items):
        signal_in = input_items[0]
        signal_out = self.amplifier(signal_in)

        output_items[0][:] = signal_out

        return len(output_items[0])
```

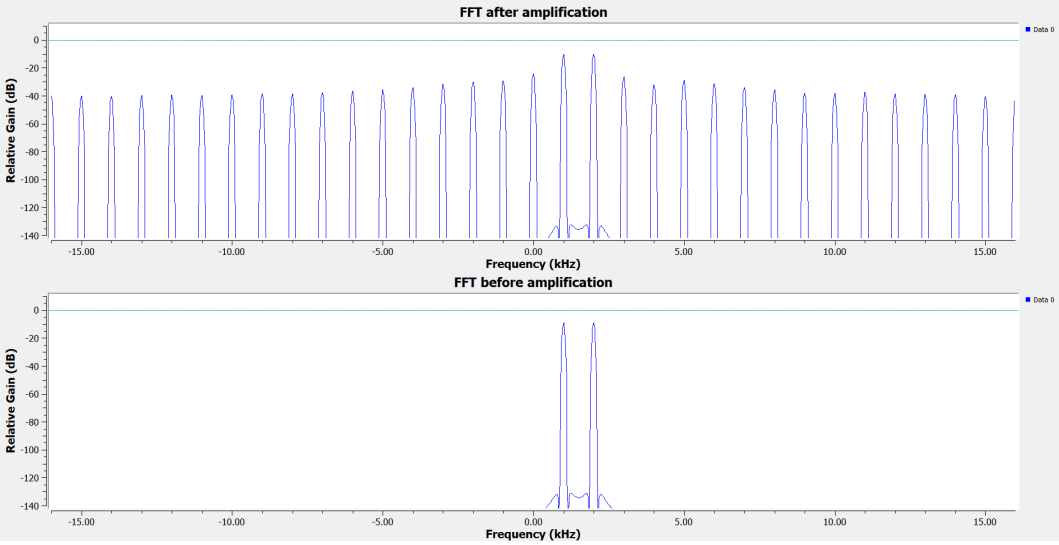
Et voici les blocs correspondants :



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 :



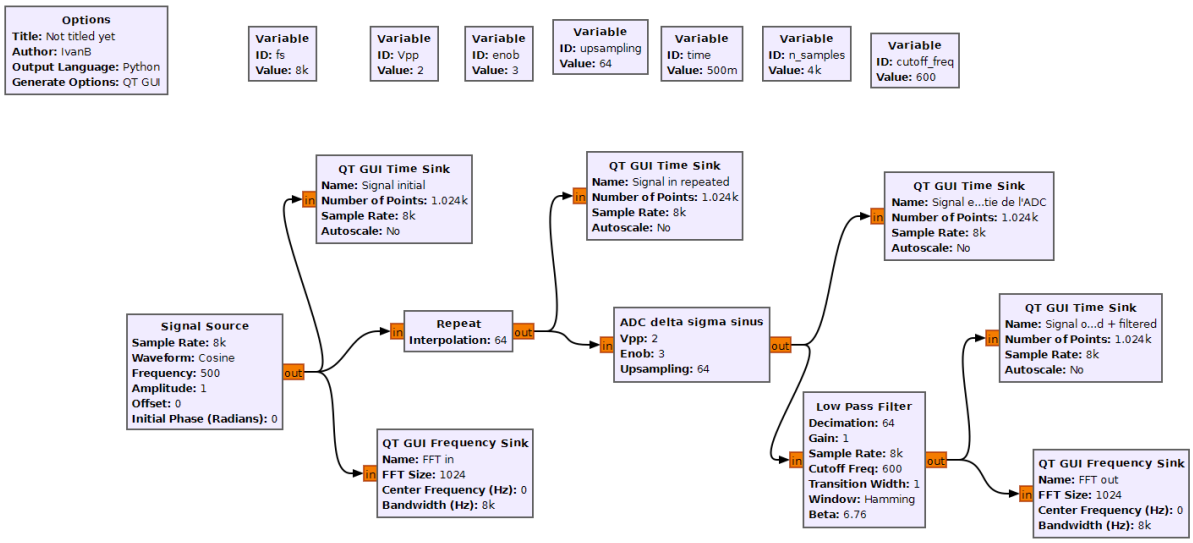
Et maintenant, voici les résultats que nous obtenons pour une somme de deux signaux :



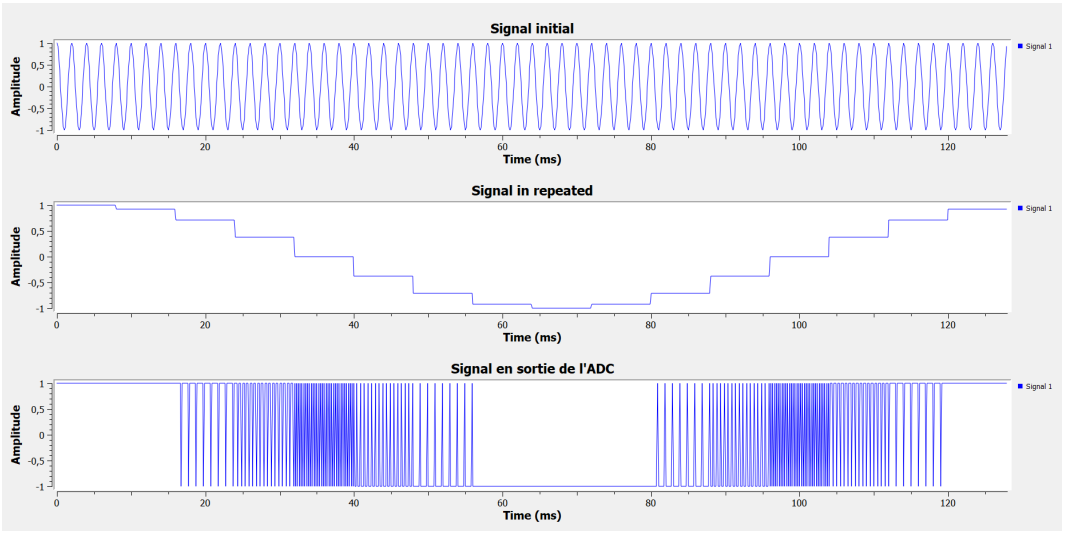
Notre bloc "Amplifiers" fonctionne donc correctement. Nous avons donc maintenant les bases de la construction d'un bloc GNU radio.

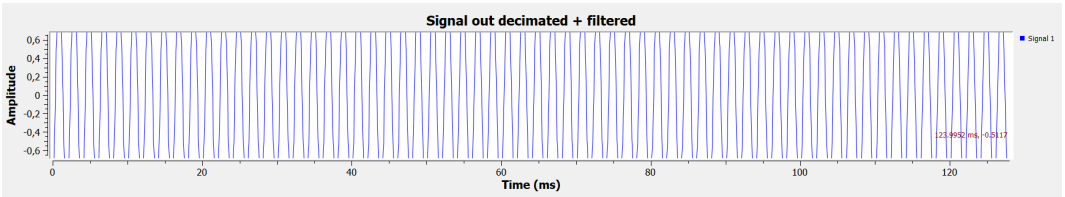
3.2.2 Bloc python pour l'ADC en entrée sinusoïdale

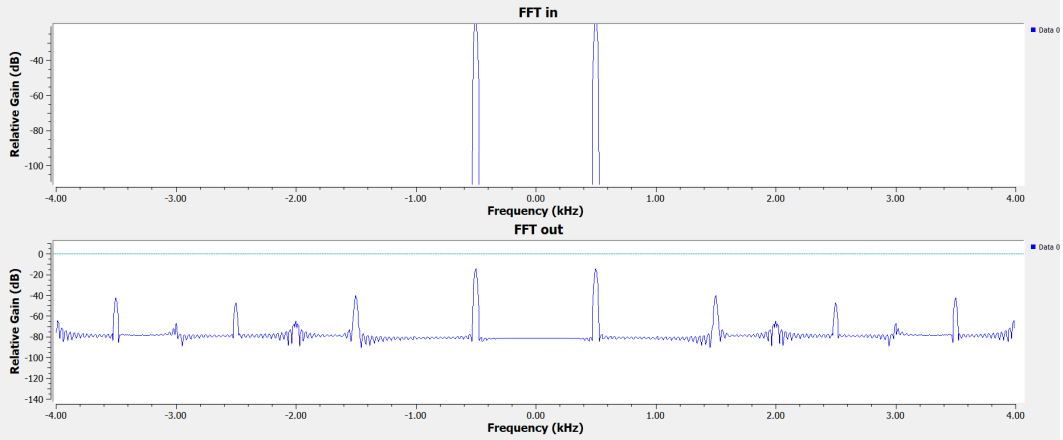
Nous allons maintenant créer notre bloc correspondant à l'ADC. Pour ce faire, nous reprenons notre code python initial pour une entrée sinusoïdale (`Premiere_partie_sinus.py`), et nous la transposons dans pour le rendre compatible dans GNU radio.



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

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 :

4 Antenne active et ADC

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 :

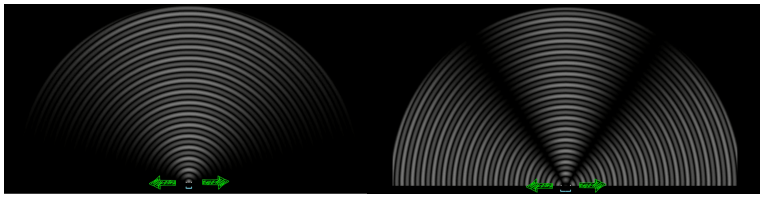
— Les antennes sont difficiles à déplacer.
— La fréquence de changement de direction est beaucoup trop élevé par rapport à la capacité de déplacement physique d'une antenne.
— La consommation d'énergie est trop importante.

Une solution consiste à changer la direction du faisceau électroniquement. En ajustant la phase de chaque petite antenne, on peut 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 :

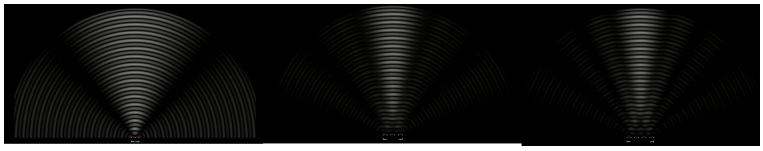
— **L'espacement entre les antennes.**

Plus on écarte l'espace entre les antennes, plus le faisceau principal se réduit, mais nous formons des lobes secondaires indésirables.



— **Nombre d'antennes.**

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



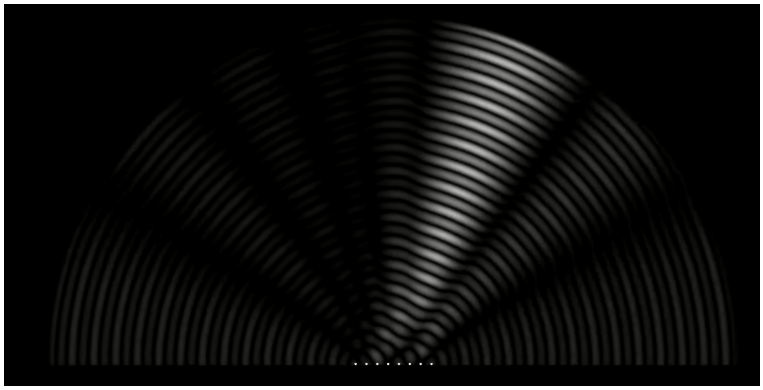
— **La disposition spatiale des antennes.**

La disposition spatiale des antennes va également modifier le schéma d'interférences.

— **La forme des signaux sur les différentes antennes.**

Enfin, les signaux émis par les différentes antennes vont également modifier la forme du faisceau.

Selon les applications envisagées, tous ces paramètres peuvent être modifiés pour ajuster le faisceau.
Ensuite, pour pouvoir changer la direction de ce faisceau, les antennes actives ajustent individuellement la phase de chaque antenne :



En ajustant les retards ou les avances de phase de chaque signal reçu par les différentes antennes, on peut 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.

De plus, nous pouvons modifier le gain de chaque antenne pour ajuster la forme du faisceau. Généralement, nous réduisons le gain des antennes extérieures et augmentons le gain des antennes centrales pour réduire la largeur du faisceau. Cependant, le lobe principal est plus large qu'auparavant. C'est donc un compromis.
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, nous affinons le faisceau en ajustant le gain, et en décalons la phase pour orienter le faisceau.

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, tandis que la phase ϕ est ajustée individuellement.

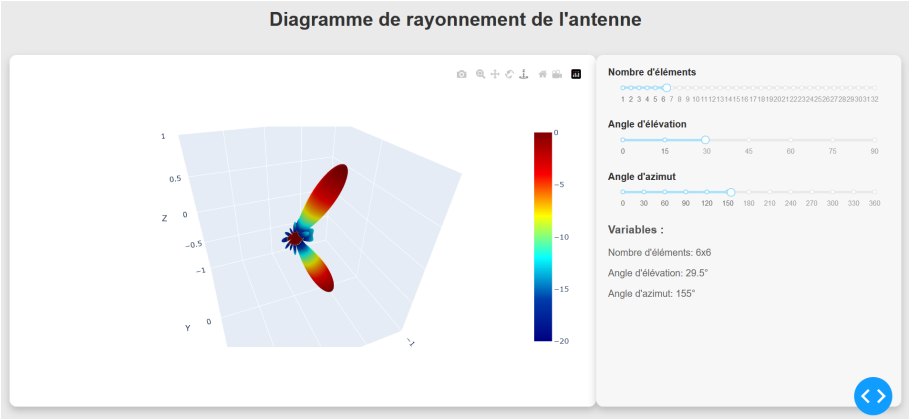
Pour notre géométrie d'antenne, qui est un carré 16 par 16 (Cette valeur est ajustable dans le code) dont les éléments sont espacés par une demi longueur d'onde entre eux, la phase supplémentaire $\Delta\phi$ introduite pour chaque élément situé à une position (x, y) est calculée comme suit :

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

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 Retard en phase et retard en temps

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, on retarde ou avance chaque signal dans le temps. Ce retard est ajusté pour chaque élément en fonction de la position de l'élément 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 :

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

Il est toutefois plus complexe et plus coûteux à implémenter.

4.4 Modélisation d'une antenne active avec des ADC $\Delta\Sigma$

5 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=9WxWun0E-PM&list=WL&index=8&t=22s&ab_channel=MATLAB