

Session S4-GE

APP6

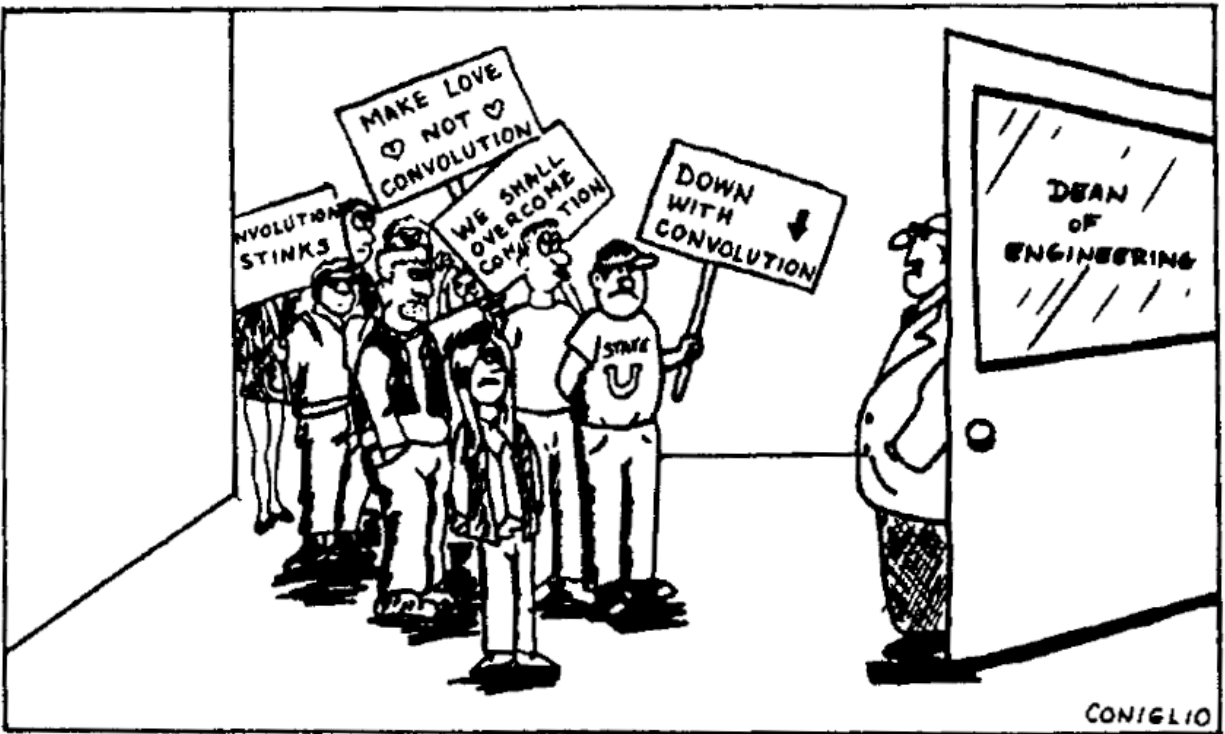
Traitement du signal sur μC

GUIDE DE L'ÉTUDIANT

**Département de génie électrique et de génie informatique
Faculté de génie
Université de Sherbrooke**

Hiver 2024

**Tous droits réservés © 2024 Département de génie électrique et de génie informatique,
Université de Sherbrooke**



Note : En vue d'alléger le texte, le masculin est utilisé pour désigner les femmes et les hommes.

Document GuideEtudiant_S4GE_APP6_H2024.docx

Conçu et rédigé par : Paul Charette

Tous droits réservés © 2024 Département de génie électrique et de génie informatique,
Université de Sherbrooke.

Reproduction permise sans autorisation à des fins d'enseignement au Département de génie
électrique et de génie informatique de l'Université de Sherbrooke seulement.

Table des matières

1	Éléments de compétences visés par l'unité.....	4
1.1	Éléments de compétences	4
1.2	Évaluation de l'unité	4
1.3	Qualités de l'ingénieur.....	5
2	Problématique	6
2.1	Énoncé.....	6
2.2	Annexe A – Organigramme du fichier S4-GE-APP6.c	8
2.3	Annexe B – Carte MX3 et modules Pmods.....	10
2.4	Annexe C – Filtrage FIR dans le domaine fréquentiel	12
2.5	Annexe D – Trucs Python.....	14
2.6	Annexe E - L'effet de fuite	17
2.7	Annexe F - Plan de vérification (développement du code C).....	19
3	Connaissances nouvelles à acquérir.....	26
3.1	GEL412 - Traitement numérique des signaux	26
3.2	GEL452 - Microcontrôleurs.....	26
4	Références.....	27
4.1	Livre de référence	27
4.2	Logiciels utilisés	27
4.3	Préparation aux activités (Lectures dans le livre de référence)	27
4.3.1	Procédural 1	27
4.3.2	Laboratoire.....	27
4.3.3	Procédural 2	27
4.3.4	Problématique	27
5	Formation à la pratique procédurale 1	28
6	Formation à la pratique en laboratoire	31
7	Formation à la pratique procédurale 2	38
8	Évaluation de la Problématique	42
	Modalités de l'évaluation de la solution à la problématique par équipe:	43
9	Mots clefs suggérés pour le schéma de concept au tutorat 2	45

1 Éléments de compétences visés par l'unité

1.1 Éléments de compétences

GEL412 - Traitement numérique des signaux

- C1 : Analyser des signaux à temps discret dans les domaines temporel et fréquentiel;
- C2 : Déterminer la réponse d'un filtre numérique linéaire à une excitation périodique et apériodique;
- C3 : Concevoir un filtre numérique selon des spécifications de tolérance, en vue d'une application donnée.

GEL452 - Microcontrôleurs

- C1 : Programmer et déployer un microcontrôleur;
- C2 : mettre en œuvre et employer une méthodologie de développement de systèmes embarqués à microcontrôleur et ses applications, en utilisant des outils de développement physique et logiciel.

1.2 Évaluation de l'unité

	Activité et éléments de compétence	Validation	Sommatif théorique	Sommatif pratique
GEL412-1	Analyser des signaux à temps discret dans les domaines temporel et fréquentiel	33	33	
GEL412-2	Déterminer la réponse d'un filtre numérique linéaire à une excitation périodique et apériodique	33	33	
GEL412-3	Concevoir un filtre numérique selon des spécifications de tolérance, en vue d'une application donnée	34	19	15
Total GEL412		100	85	15
		200		
GEL452-1	Programmer et déployer un microcontrôleur	20	30	
GEL452-2	Mettre en œuvre et employer une méthodologie de développement de systèmes embarqués à microcontrôleur et ses applications, en utilisant des outils de développement physique et logiciel	20		30
Total GEL452		40	30	30
		100		

1.3 Qualités de l'ingénieur

Les qualités de l'ingénieur visées par cette unité d'APP sont les suivantes. D'autres qualités peuvent être présentes sans être visées ou évaluées dans cette unité d'APP. Pour une description détaillée des qualités et leur provenance, consultez le lien suivant :

<https://www.usherbrooke.ca/genie/etudiants-actuels/au-baccalaureat/bcapg/>

Qualités requises des diplômés		Touchée	Évaluée
Q01	Connaissances en génie	X	
Q02	Analyse de problèmes	X	
Q03	Investigation		
Q04	Conception		
Q05	Utilisation d'outils d'ingénierie	X	
Q06	Travail individuel et en équipe		
Q07	Communication		
Q08	Professionnalisme		
Q09	Impact du génie sur la société et l'environnement		
Q10	Déontologie et équité		
Q11	Économie et gestion de projets		
Q12	Apprentissage continu		

2 Problématique

2.1 Énoncé

Mélangeur audio

Vous venez d’être embauché par une compagnie qui fabrique des accessoires de musique électronique. Votre superviseur a identifié le PIC32 comme un candidat intéressant pour faire du traitement numérique de signal audio en temps réel dans leurs futurs produits. Pour ce faire, il a acheté une carte MX3 qui permet d’évaluer les capacités du PIC32.

Comme démonstration, votre patron vous demande de compléter un mélangeur (*mixer*) audio fonctionnant à une fréquence d’échantillonnage de 20 kHz. La conception du code est faite en langage C dans l’environnement de développement MPLABX que vous connaissez déjà. Vous devrez y ajouter les plugins MCC et DMCI afin d’accélérer le développement et le déverminage du code. Votre prédécesseur a déjà commencé le projet avant de quitter la compagnie pour changer de vocation professionnelle. Il vous a laissé un organigramme du programme principal en C (voir Annexe A – Organigramme du fichier S4-GE-APP6.c) où les sections en vert restent à compléter.

Le signal audio en entrée est numérisé par un convertisseur analogique à numérique et la sortie audio filtrée est synthétisée par PWM (*pulse width modulation*), tous deux des périphériques internes du PIC32. Deux modules Pmod vous sont fournis pour gérer les signaux en entrée et sortie (voir Annexe B – Carte MX3 et modules Pmod ainsi que les documents *Pmod - Filtre anti-repliement en ENTRÉE.pdf* et *Pmod - Filtre reconstruction en SORTIE.pdf*).

Le système de traitement du signal numérique programmé en C comportera 5 filtres FIR à 255 ou 256 coefficients, selon le type de filtre, qui pourront être actionnés indépendamment par les interrupteurs SW7 à SW3 de la carte MX3 avec feedback visuel sur l’écran LCD. Les caractéristiques des filtres sont:

- 1) Passe-bas de coupure à 500 Hz
- 2) Passe-bandes : 1000 ± 500 Hz, 2000 ± 500 Hz, 3500 ± 1000 Hz
- 3) Passe-haut de coupure à 4490 Hz

Les filtres FIR seront conçus par la méthode des fenêtres (Blackman). Comme vous avez une affection particulière pour la transformation de Fourier, vous choisissez une approche de filtrage par blocs dans le domaine fréquentiel. En effet, la combinaison des fonctions de transfert des différents filtres activés se fait par simple addition dans le domaine fréquentiel. La longueur des blocs traités sera en accord avec les recommandations minimales données par Richard Lyons pour la méthode *overlap-and-save* (voir Annexe C – Filtrage FIR dans le domaine fréquentiel).

Puisque le PIC32MX370F512L utilise un processeur MIPS à calcul sur nombres entiers, vous savez qu’il n’est vraiment pas optimal du point de vue des ressources (mémoire disponible, vitesse d’exécution) d’écrire du code C qui fonctionnerait en point flottant, vous choisissez donc de faire tous les calculs dans le code en point fixe sur 32 bits. Pour minimiser les chances d’erreurs de saturation et de retournement dans les calculs, vous choisissez d’utiliser les versions à 32 bits des fonctions de transformation de Fourier rapide dans la librairie DSP du PIC32. Même si la librairie ne contient pas explicitement de fonction de transformation de Fourier inverse, vous n’êtes nullement perturbé(e), vous savez qu’il est tout à fait possible de faire une transformée inverse avec une fonction de transformée de Fourier normale.

Dans un bar de traitement du signal numérique sur la rue Wellington sud à Sherbrooke, vous avez entendu dire pendant la semaine de relâche que les filtres IIR sont plus puissants que les filtres FIR, mais que ceux-ci peuvent être instables (aucun rapport avec l'alcool), en raison de quoi ils sont déconseillés pour le contrôle du niveau d'eau dans les toilettes du bar. Cependant, pour une application audio, comme le danger vous semble acceptable, vous décidez d'ajouter un filtre IIR coupe-bande à votre système afin d'éliminer une composante fréquentielle à 1 kHz dans le signal d'entrée (le son dérangent qui émane de la bouilloire *old-school* de votre patron). Le filtre sera de type elliptique d'ordre 4 avec une bande passante de 950 à 1050 Hz (ronflement maximal de 1 dB dans la bande passante et atténuation minimale de 70 dB dans la bande atténuée).

Afin d'effectuer le filtrage IIR du signal d'entrée en temps réel, votre code de filtrage IIR devra être ajouté à la fonction de gestion d'interruption (*interrupt handler*) du convertisseur analogique à numérique du PIC32, soit à la fonction *ADC_1()* dans le fichier *adc1.c* généré par le plugin MCC. Toujours selon les conseils entendus au bar, il semblerait que la structure la plus efficace et robuste pour le filtrage IIR relativement aux erreurs de retournement soit une structure en cascade de filtres de second ordre (SOS : *second order sections*). Tristement, vous n'avez pas le droit d'utiliser la fonction *mips_iir16()* de la librairie DSP du PIC32, il vous faudra coder explicitement l'algorithme de filtre IIR en forme Direct II *Transposed* en cascade.

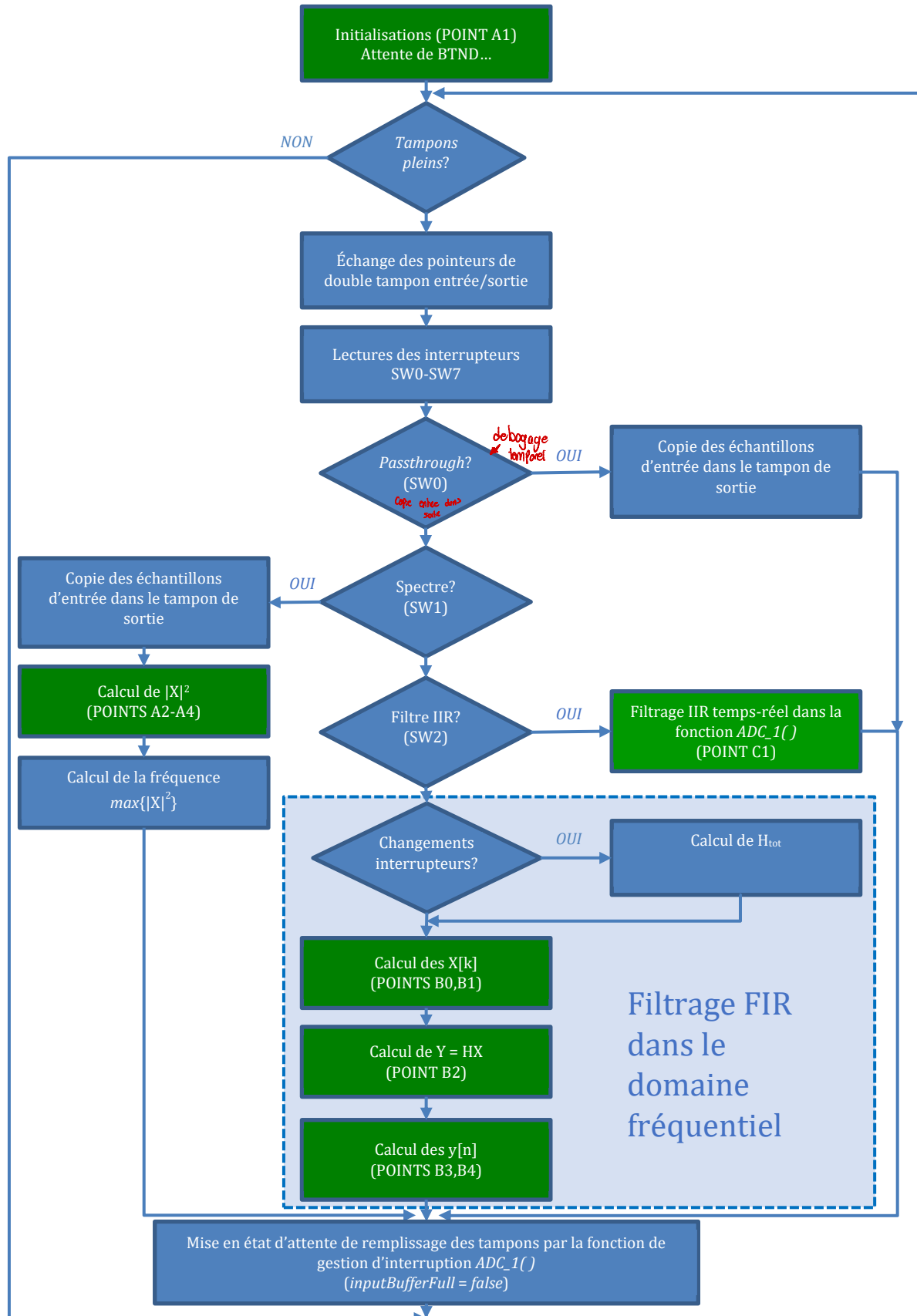
Tous les filtres seront conçus dans un environnement Python. Pour les FIR, utilisez la fonction *scipy.signal.firwin* (voir Annexe D – Trucs Python). Par soucis de flexibilité dans la conception du projet en C, les coefficients des filtres seront incorporés au code dans des fichiers « .h ». Comme les calculs dans le code C seront fait en point fixe sur 32 bits, vous aurez convertis les coefficients de vos filtres en format Q2.13. Enfin, pour démontrer que vous comprenez bien les conséquences des limites de ressources de calcul sur la qualité du filtrage numérique, vous faites la démonstration de votre filtre IIR mais cette fois-ci avec les coefficients encodés en format Q2.5.

Comme le code C développé sera relativement complexe (combinaison d'exécution en temps réel et de traitement du signal numérique dans le domaine fréquentiel sur un microcontrôleur à ressources limitées), vous décidez d'adopter des bonnes pratiques de développement et de déverminage inspirées par le « développement piloté par les tests » (*Test-Driven Development*, ou TDD), une méthodologie largement utilisée en milieu industriel. Vous procéderez donc de manière structurée au développement et déverminage du code C en suivant un plan de vérification (voir Annexe F - Plan de vérification (développement du code C)).

Notes importantes:

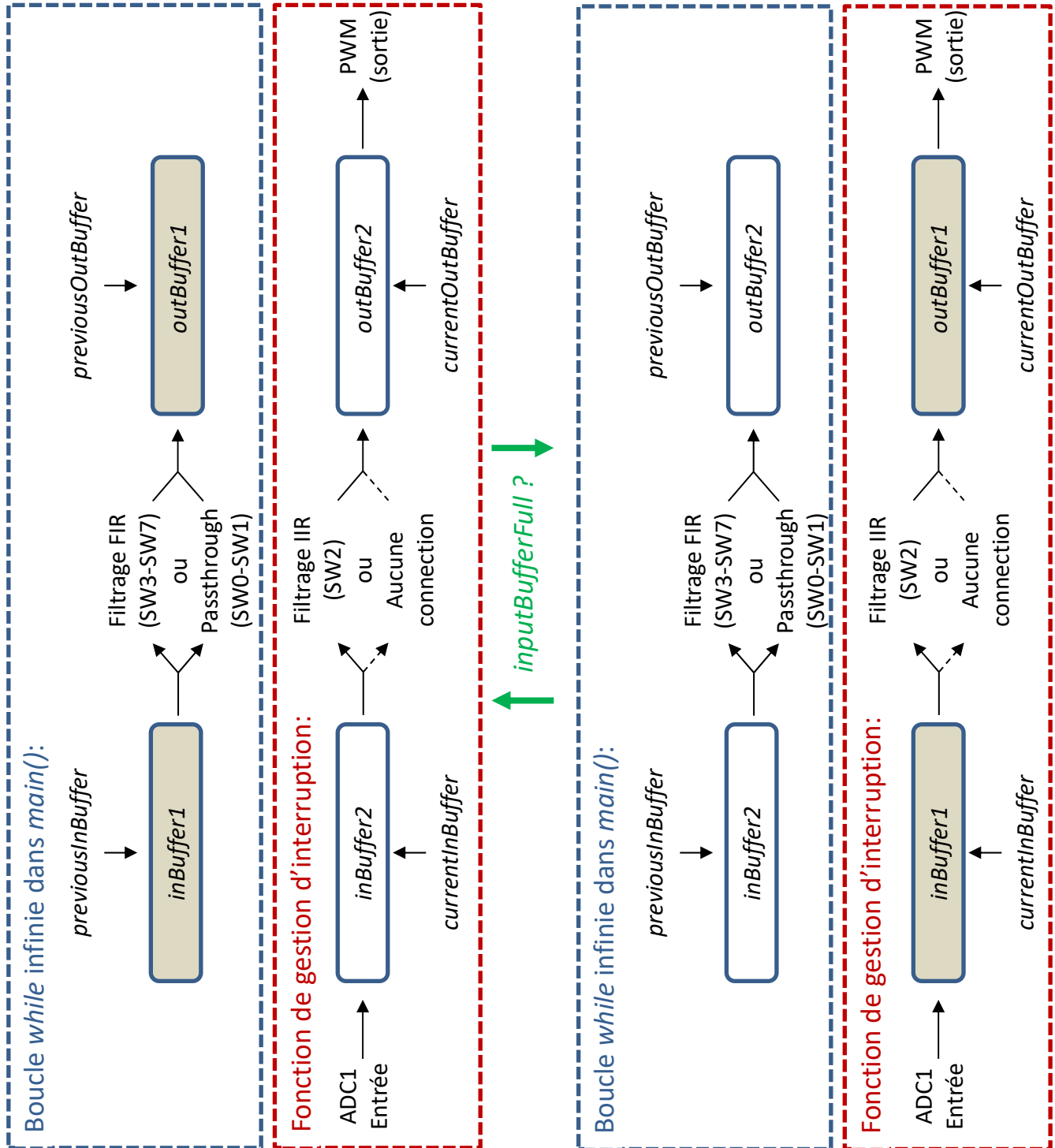
- 1) Un *Guide de résolution de la problématique* sera affiché sur le site web de l'APP après le 1^{er} tutorat.
- 2) La très grande majorité des points pour la résolution de la problématique sont attribués à l'évaluation de votre document de plan de vérification dans lequel vos résultats Python et MPLAB/DCMI seront consignés, voir l'Annexe F - Plan de vérification (développement du code C).
- 3) Votre plan de vérification ainsi que vos projets MPLAB et PyCharm doivent être remis avant la validation en laboratoire.
- 4) Toute demande d'aide au tuteur pour le déverminage de votre code C devra faire référence à votre plan de vérification.
- 5) Même s'il s'agit ici de signaux audios, il est complètement inutile d'y aller « à l'écoute » pour déverminer votre projet en C, l'oreille n'a vraiment pas assez de bits dans sa FFT...

2.2 Annexe A – Organigramme du fichier S4-GE-APP6.c

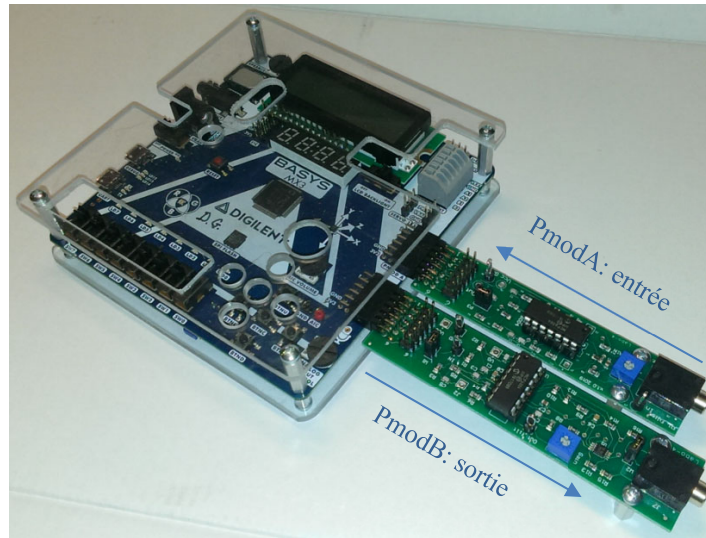


Double mise en mémoire tampon (« *double buffering* ») :

À chaque déclenchement de la boucle principale avec *inputBufferFull* = TRUE, les paires de pointeurs *currentInBuffer/currentOutBuffer* et *previousInBuffer/previousOutBuffer* vont intervertir leurs cibles (les tampons *inBuffer1/outBuffer1* & *inBuffer2/outBuffer2*).



2.3 Annexe B – Carte MX3 et modules Pmods



Pin-out du connecteur AD2 à l'endos de la carte MX3 pour accès à BIN1/BIN2:

Pin-out du connecteur Pmod :

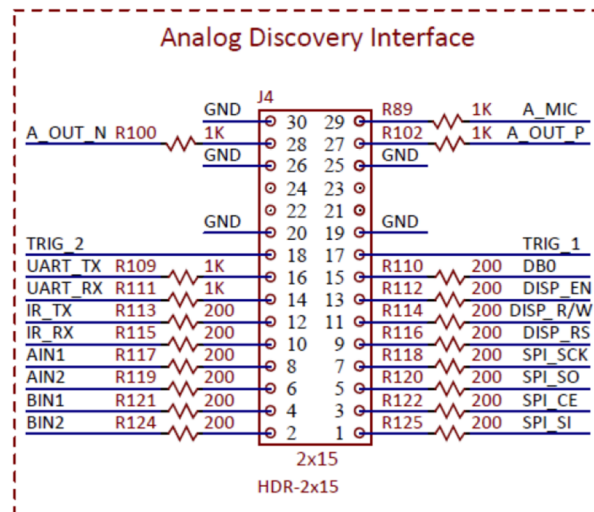
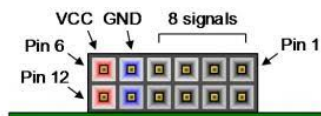


Figure 21.1. Debug header schematic diagram.

PmodA – Module d'interface pour le signal d'entrée (la MX3 n'a pas d'entrée audio externe):

- Entrée audio (mono) 3.5 mm.
- Test point *In* : signal d'entrée centré à 1.6V.
- Filtre anti-repliement (filtre elliptique actif d'ordre 4, fréquence de coupure ~10 kHz).
- Connecteur 2x6 qui permet de prendre des lectures avec l'Analog Discovery 2 (AD2):
 - Broche 8 : signal filtré qui sera numérisé par la carte MX3.
 - Broches 5 et 11 : GND (référence globale).

NB : l'amplitude du signal d'entrée doit être d'au maximum 3.2 Vpp, validez via la broche 8 du connecteur 2×6 avec l'AD2 qu'il n'y a pas saturation.

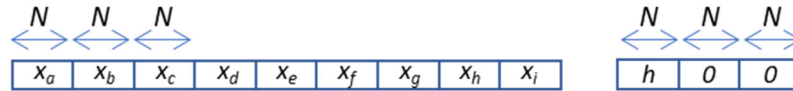
PmodB – Module d'interface pour le signal de sortie (filtre de reconstruction supérieur à celui de la carte MX3):

- Le connecteur 2×6 expose les signaux suivants :
 - Broche 4: signal du PWM issu de la carte MX3 avant le filtre de reconstruction (onde carrée à 142.9 kHz avec modulation du *duty cycle*, il est peu utile de regarder ce signal directement avec l'AD2).
 - Broches 5 et 11: GND (référence globale).
- Filtre de reconstruction (filtre elliptique actif d'ordre 4, fréquence de coupure ~10 kHz).
- Test point *Out Filt* : signal filtré et centré à 1.6V (la sortie peut être lue à cet endroit seulement avec l'AD2, la sortie sur le connecteur audio n'étant pas référencée à la masse).
- Sortie audio 3.5 mm pour casque d'écoute.

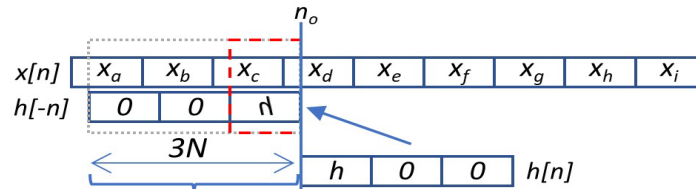
NB : ajustez d'abord le gain via le potentiomètre à son minimum, puis augmentez jusqu'à obtenir une amplitude au test point *Out Filt* du même niveau que l'entrée (Test point *In* sur le PmodA).

2.4 Annexe C – Filtrage FIR dans le domaine fréquentiel

On désire filtrer un signal $x[n]$ groupé en blocs x_a à x_i , chacun de longueur N , par trames de 3 blocs avec un filtre dont la réponse impulsionnelle $h[n]$ est de longueur N . Pour filtrer dans le domaine fréquentiel avec $Y = X \times H$, on doit rallonger $h[n]$ à longueur $3N$ par *zero-padding* pour que H et X soient de même longueur :

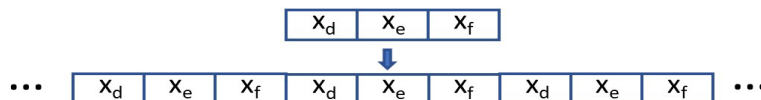


L'équivalent par convolution *linéaire* dans le domaine temporel est montré plus bas pour le calcul d'un échantillon $y[n]$ à $n = n_o$, où $h[n]$ est inversé et aligné sur l'échantillon n_o dans le bloc x_d . La sortie $y[n_o]$ sera égale à la somme des produits point-à-point $x[n_o-k] \times h[k]$ avec $k = [0..3N-1]$:

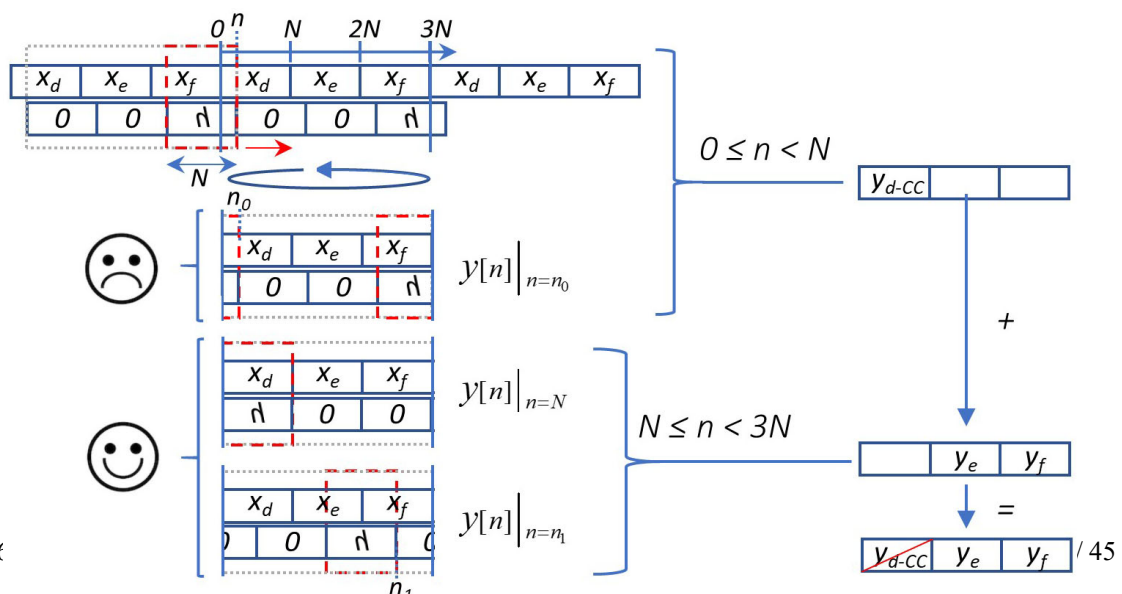


$$y[n_o] = \sum_{k=0}^{k < 3N} x[n_o-k]h[k] = \sum_{k=0}^{k < N} x[n_o-k]h[k], \text{ parce } h[k]=0 \text{ pour } k \geq N \text{ (rectangle rouge)}$$

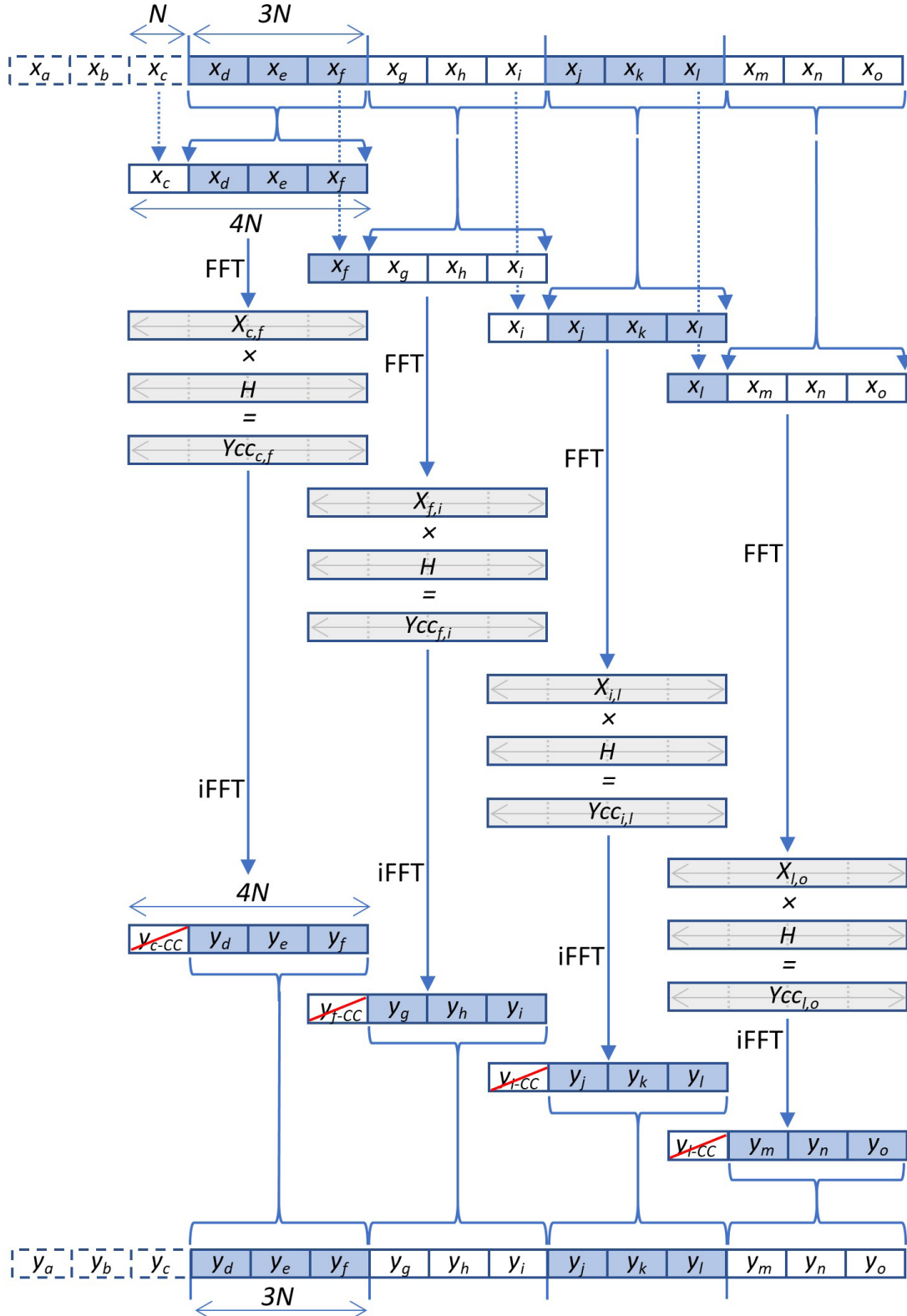
MAIS, le traitement du signal dans le domaine fréquentiel *discret* rend le signal temporel d'entrée $x[n]$ *périodique*. Pour le filtrage de la trame $[x_d, x_e, x_f]$, le traitement fréquentiel discret transforme $x[n]$ en un signal périodique de période $3N$, i.e. ce n'est *plus le même signal* :



Or $x[n]$ étant « devenu » périodique, la convolution devient *circulaire*, voir le rectangle rouge « replié » dans le premier cas plus bas. La sortie $y[n]$ pour $0 \leq n < N$ sera différente de la convolution linéaire parce qu'elle est calculée entre $h[n]$ et les blocs x_f et x_d au lieu de x_c et x_d . Les premières N valeurs de $y[n]$ (y_{d-cc} plus bas) doivent être donc rejetées dans le filtrage. La sortie $y[n]$ sera correcte pour $n \geq N$ parce que seules les portions nulles de $h[n]$ seront « repliées » dans le tampon (NB : $y[n]$ est correct à partir de $n \geq N-1$, mais on y va avec $n \geq N$ pour simplifier).



La méthode *Overlap & Save* (Lyons, 13.10) apporte une solution au problème de convolution circulaire dans le filtrage fréquentiel par recouvrement (*overlap*) entre les trames successives filtrées, où la longueur du recouvrement est celle de la réponse impulsionnelle du filtre, N (1 bloc). Dans l'exemple plus bas, les cases x_a à x_o et y_a à y_o sont des blocs de longueur N des signaux d'entrée/sortie. Dans la problématique, $N = 256$, donc les FFTs sont faites sur des tableaux de $4N = 1024$ échantillons, ce qui permet de filtrer une trame de $3N = 768$ échantillons à la fois (3 blocs).



2.5 Annexe D – Trucs Python

- Exemples avec structure plus avancée qu'à l'APP5, notez les déclarations de types des variables dans le code et dans les définitions des fonctions (*type hinting* ou *type annotations*).

```
# Libraries requises
from matplotlib import pyplot as plt, rcParams as mpl_rcParams
import numpy as np
from scipy import signal

def calcul_des_filtres(n: int, fc: float, fe: float):
    """
    Réponses impulsionnelles et fonctions de transfert des filtres FIR et IIR
    """

    # Sinusoïde à 200 Hz, rallongée à longueur 2N par "zero-padding"
    f_sig: float = 200
    x200: np.ndarray = np.sin(2 * np.pi * np.arange(n) * f_sig / fe)
    x200_zero_pad: np.ndarray = np.append(x200, np.zeros(len(x200)))
    plt.figure()
    plt.plot(np.arange(len(x200_zero_pad)) / fe, x200_zero_pad)
    plt.plot(np.arange(len(x200)) / fe, x200)
    plt.title(f"Signal sinusoïdal à {f_sig} Hz doublé en longueur par zero-padding")
    plt.xlabel("Temps (s)")
    plt.ylabel("Amplitude normalisée")

    # Filtre FIR: réponse impulsionnelle d'ordre N-1 à fréquence de coupure fc
    fir_h: np.ndarray = signal.firwin(
        numtaps=n, cutoff=fc, pass_zero="lowpass", window="hamming", fs=fe
    )

    # Filtre FIR: fonctions de transfert harmoniques ("h_dft_*") calculées
    # par freqz() et explicitement par TF pour comparer. Dans l'appel de freqz()
    # pour un FIR, notez les bi qui sont les coefficients de la réponse
    # impulsionnelle et qu'il n'y a qu'un seul coefficient aj (a0 = 1),
    # ainsi que l'usage du paramètre worN pour augmenter la résolution du spectre.
    fir_freq_fz, fir_h_dft_fz = signal.freqz(b=fir_h, a=1, worN=10000, fs=fe)
    fir_h_dft_tf: np.ndarray = np.fft.fft(fir_h)

    # Filtre IIR d'ordre 4: coefficients a & b de l'équation aux différences
    # et calcul de sa réponse impulsionnelle tronquée (N premiers échantillons)
    [b, a] = signal.butter(N=4, Wn=fc, btype="low", fs=fe, output="ba")
    imp: np.ndarray = signal.unit_impulse(n)
    iir_h: np.ndarray = signal.lfilter(b=b, a=a, x=imp)

    # Filtre IIR: fonctions de transfert harmoniques ("h_dft_*") du filtre IIR
    # par freqz() et explicitement par TF (h tronqué) pour comparer
    freq_iir_fz, h_dft_iir_fz = signal.freqz(b=b, a=a, worN=10000, fs=fe)
    h_dft_iir_tf: np.ndarray = np.fft.fft(iir_h)

    # Graphiques des réponses impulsionnelles des filtres. Remarquez que
    # le filtre FIR est causal versus le filtre IIR qui est non-causal et déphasé
    plt.figure()
    plt.subplot(3, 1, 1)
    plt.plot(fir_h, label="FIR")
    plt.plot(iir_h, label="IIR")
    plt.title("Réponses impulsionnelles des filtres")
    plt.xlabel("n")
    plt.ylabel("Amplitude normalisée")
    plt.legend()
```

```

# Graphiques des fonctions de transfert harmoniques du filtre FIR (freqz & TF).
# Dans le cas de la TF, afin d'extraire les valeurs du spectre aux fréquences
# non-négatives seulement, remarquez l'usage de la division avec
# troncature "/" pour générer un nombre entier tel que requis pour des
# indices de tableaux (une simple division avec "/" aurait causé une erreur).
# Notez aussi la résolution accrue avec freqz() grace au paramètre worN.
f_nn: np.ndarray = np.arange(0, fe / 2, fe / n)
plt.subplot(3, 1, 2)
plt.semilogx(fir_freq_fz, 20 * np.log10(np.abs(fir_h_dft_fz)), label="freqz()")
plt.semilogx(f_nn, 20 * np.log10(np.abs(fir_h_dft_tf[0 : n // 2])), label="TF")
plt.ylim(top=10, bottom=-200)
plt.title("Fonctions de transfert harmoniques du filtre FIR")
plt.xlabel("Fréquence [Hz]")
plt.ylabel("Gain [dB]")
plt.legend()

# Graphiques des fonctions de transfert harmoniques du filtre IIR (freqz & TF)
# aux fréquences non-négatives seulement.
plt.subplot(3, 1, 3)
plt.semilogx(freq_iir_fz, 20 * np.log10(np.abs(h_dft_iir_fz)), label="freqz()")
plt.semilogx(f_nn, 20 * np.log10(np.abs(h_dft_iir_tf[0 : n // 2])), label="TF")
plt.ylim(top=10, bottom=-200)
plt.title("Fonctions de transfert harmoniques du filtre IIR")
plt.xlabel("Fréquence [Hz]")
plt.ylabel("Gain [dB]")
plt.legend()
plt.show()

def code_exemple_guide_etudiant():
    """
    Fonction principale appelant les fonctions secondaires (e.g. main() en C++)
    """

    # Propriétés par défaut de matplotlib
    plt.rcParams.update(
        {
            "backend": "TkAgg",
            "axes.grid": True,
            "axes.grid.which": "both",
            "figure.autolayout": True,
            "interactive": True,
        }
    )

    # Appel des fonctions spécifiques
    n: int = 1024
    fc: float = 100
    fe: float = 22100
    calcul_des_filtres(n, fc, fe)

    # Breakpoint pour mettre l'interpréteur en pause afin d'afficher les figures
    print("Done!")

if __name__ == "__main__":
    code_exemple_guide_etudiant()

```

- Ligne « `if __name__ == "__main__":` »

L'exemple de code précédant utilise la structure recommandée pour un script Python en bonne et due forme qui, grâce à la ligne de code ci-dessus (voir https://www.geeksforgeeks.org/what-does-the-if-__name__-__main__-do/), permet de :

- 1) Exécuter le script au complet directement à partir de l'interpréteur Python OU utiliser le script comme librairie de fonctions qui peuvent être appelées depuis un autre script.
- 2) Forcer la définition explicite des variables globales (pas recommandé de toute façon) à l'extérieur des fonctions et ainsi éviter des erreurs causées par des conflits cachés de portée (*scope*) entre des variables locales et des variables autrement définies implicitement comme globales.

- Fonction `scipy.signal.firwin` : paramètres *numtaps* et *pass-zero*

Type de filtre	Nombre de coefficients, <i>N</i> (paramètre <i>numtaps</i>)	Paramètre <i>pass-zero</i>
Passe-bas	pair	'lowpass'
Passe-haut	impair	'highpass'
Passe-bande	pair	'bandpass'
Coupe-bande	impair	'bandstop'

- Pour construire une chaîne de texte formatée (*formatted text string*), utilisez la méthode « *f-strings* » introduite dans Python 3 (<https://realpython.com/python-f-strings/>). Il suffit de mettre la lettre « f » devant la chaîne et d'insérer les noms des variables entre "{}":

```
p1 = "Bob"
p2 = "Alice"
t = 32.56735
print(f"{p1} et {p2} vont à l'école. Il fait {t:.2f} oC")
```

Console :

```
>> Bob et Alice vont à l'école. Il fait 32.57 oC
```

- Code pour écrire un tableau de nombres entiers complexes dans un fichier .h en C (notez les dédoublements du caractère "{}" pour insérer des "{" dans la chaîne *f-strings*):

```
h_dft_qxy: np.ndarray = np.array([3 + 6j, 1 + 9j, 55 + 6j])
with open("bob.h", "w") as fd:
    fd.write(f"int32c H[{len(h_dft_qxy)}] = {{\n")
    for c in h_dft_qxy:
        fd.write(f"{{{int(c.real)}},{int(c.imag)}}},\n")
    fd.write("};\n")
```

- Ajouts recommandés à PyCharm ou autre IDE de votre choix:
 - Dictionnaire français (<http://www.winedt.org/dict.html>), <Settings><Editor><Spelling>
 - Plugin « Black »: (voir <https://black.readthedocs.io/en/latest/> et <https://plugins.jetbrains.com/plugin/10563-black-pycharm>)
 - Plugin « Mypy »: (voir <https://github.com/leinardi/mypy-pycharm>)

2.6 Annexe E - L'effet de fuite

Références :

- Notes de cours GEL412 de l'APP5 par François Grondin
- Sections 3.8 *DFT Leakage* et 3.9 *Windows* dans le livre de référence (Lyons, 2011)

Mathématiquement, en traitement du signal, les signaux sont toujours de longueur infinie. Cependant, le seul moyen d'exprimer mathématiquement un signal infini est avec une équation, par exemple avec $y(t) = \sin(2\pi ft)$.

Dans le traitement numérique du signal, les signaux sont représentés par une série de valeurs, donc nécessairement de longueur finie, en tout cas pour un APP limité à 2 semaines. *Mathématiquement*, un signal $x[n]$ de longueur finie N aura été obtenu en multipliant le signal original de longueur infinie, $x_{inf}[n]$, par une fenêtre, $w[n]$, aussi de longueur infinie qui a des valeurs non-nulles pour $n = [0, N-1]$ seulement et nulles ailleurs. Donc, du point de vue mathématique, quand vous traitez un signal $x[n]$ de longueur N , celui-ci provient d'un signal fictif de longueur infinie $x_{inf}[n]$ qui a été auparavant multiplié par une *fenêtre rectangulaire* (un signal de longueur infinie qui vaut 1 pour $n = [0, N-1]$ et zéro ailleurs). Le fenêtrage rectangulaire est donc "implicite" en traitement du signal numérique et sa conséquence est l'*effet de fuite*. **Morale de l'histoire : quitte à fenêtrer, faisons-le correctement (i.e. explicitement)!**

L'*effet de fuite* dans le domaine spectral se produit quand un signal contient des composantes fréquentielles à des fréquences qui ne sont pas représentées explicitement dans un spectre discret, ce qui est généralement le cas en pratique.

Si par exemple un signal $x[n]$ de longueur $N = 10$ est une sinusoïdale pure à 5 Hz échantillonnée à $f_c = 25$ Hz, les échantillons du spectre discret $X[k]$ seront séparés par $\Delta f = f_c/N = 2.5$ Hz:

- 1) Le spectre aura exactement et seulement deux raies à -5 Hz et +5 Hz, soit à $k = -2$ et $k = 2$ respectivement.
- 2) Le signal $x[n]$ sera composé d'une succession de périodes complètes de sinusoïdes à 5 Hz (2 périodes complètes dans cas-ci), le nombre de périodes étant dépendant de la longueur N et de la période d'échantillonnage, $1/f_c$.

Il n'y aura donc pas d'effet de fuite dans cet exemple et prendre $x[n]$ intégralement sans fenêtrage explicite pour calculer le spectre, i.e. faire un fenêtrage rectangulaire pour « isoler » $x[n]$ à partir d'un signal hypothétique infini, $x_{inf}[n]$, composé d'une répétition sans fin de $x[n]$, ne modifiera pas le spectre parce la fenêtre contiendra un nombre entier de périodes de $x_{inf}[n]$.

Si $x[n]$ est une sinusoïde pure à une fréquence qui n'est pas un multiple de 2.5 Hz, disons 6 Hz:

- 1) Les raies spectrales qui devraient être à ± 6 Hz vont « tomber entre deux chaises » et leur énergie sera répartie sur des raies de part et d'autre de ± 6 Hz, c'est l'*effet de fuite*.
- 2) Le signal $x[n]$ ne sera pas composé d'une succession de périodes complètes de la sinusoïde à 6 Hz, la première et/ou la dernière période seront incomplètes.

Le théorème de convolution dit qu'une multiplication dans un domaine équivaut à une convolution dans l'autre. Donc, multiplier $x_{inf}[n]$ par une fenêtre pour en « extraire » $x[n]$ équivaut à convoluer

X_{inf} par la TF de la fenêtre, ce qui aura pour effet dans de remplacer chaque raie pure de X_{inf} par un groupe de raies (la TF de la fenêtre!), i.e. le spectre de X_{inf} est « modifié » (sauf dans le cas exceptionnel plus haut à 5 kHz, qui n'arrive jamais en pratique). La fenêtre qui fait le plus de « dommage » est la fenêtre rectangulaire (i.e. fenêtrage *implicite*, simplement utiliser $x[n]$) : puisque la TF d'une fenêtre rectangulaire est un $\sin(x)/x$ répété (en anglais : *periodic sinc* ou *aliased sinc function*), chaque raie spectrale de X_{inf} devient un $\sin(x)/x$ centré sur la raie en question, puis échantillonné aux fréquences discrètes $\Delta f = f_c/N$.

NB : pour des opérations de filtrage, on ne fait pas de fenêtrage, le signal $x[n]$ « est ce qu'il est » et une fréquence d'échantillonnage la plus élevée possible sera la stratégie principale pour minimiser les dommages causés par l'effet de fuite. Le fenêtrage est utilisé uniquement pour évaluer le plus exactement possible le contenu spectral d'un signal en minimisant l'effet de fuite qui pourrait autrement donner la fausse impression que le signal a plus de diversité dans son contenu fréquentiel qu'il n'en a réellement.

2.7 Annexe F - Plan de vérification (développement du code C)

Comme le code C temp-réel développé est relativement complexe, il sera impératif de procéder au développement et au déverminage de manière structurée, i.e. en suivant un plan de vérification. Ce plan sera composé d'une série d'étapes dans un ordre logique de déverminage (voir l'exemple de gabarit plus bas, vous pouvez élaborer votre propre structure), les étapes ayant les informations suivantes :

- Titre descriptif « haut niveau » qui permet de comprendre le but de l'étape.
- Description du résultat attendu faisant références à des variables/tableaux spécifiques, où vous préciserez au besoin le type des variables (*datatype*, longueur des tableaux, etc.) ainsi que les valeurs et plages dynamiques attendues (amplitude, fréquence, etc.) **avec des graphiques générés avec Python dans le cas de tableaux.**
- Description de(s) test(s) unitaire(s) par *breakpoint* dans le code ou *test point* matériel qui permettra de valider le résultat, où vous ferez notamment bon usage du plugin DMCI dans MPLAB pour visualiser des tableaux.
- Consignation des résultats observés à date.

Consignes/conseils :

- *Avant* de débiter le développement du code, vous devez faire une *première ébauche complète* du plan avec le plus d'informations précises possibles.
- Vous procéderez ensuite au développement *incrémental* du code à chaque étape et validerez son bon fonctionnement avec le(s) test(s) unitaire(s) de l'étape. Il est inutile de procéder aux étapes subséquentes du plan si le code de l'étape actuelle n'est pas fonctionnel, un problème qui se propage à travers le système devient rapidement beaucoup plus difficile à déverminer.
- Le plan de vérification sera un document en évolution durant la résolution de la problématique, vous pourrez y faire des ajouts/améliorations au fur et à mesure en fonction de l'expérience acquise et de la meilleure compréhension de la problématique.
- **Tous les graphiques doivent avoir un titre et leurs axes doivent être identifiés avec les unités appropriées. De plus, les graphiques doivent être décrits adéquatement**, par exemple: « Le graphique ci-haut montre la réponse du filtre H3 à une sinusoïde de 300 Hz, l'amplitude est atténuée à 200 tel qu'attendu parce que le filtre coupe à... »
- **Les graphiques des spectres et des fonctions de transfert doivent montrer les amplitudes en dB. De plus, les graphiques générés avec Python doivent montrer les données aux fréquences non-négatives seulement, en Hz, sur une échelle logarithmique.**

Pour toute demande d'aide au tuteur pour le déverminage de votre code C, vous devrez :

- **Avoir soumis votre plan en avance** pour que le tuteur ait pu en prendre connaissance.
- Formuler toute question en vous référant à un item spécifique de votre plan de vérification, avec résultats actuels à l'appui. Les étapes précédentes du plan devront bien entendu avoir été complétées avec succès, avec résultats à l'appui.

NB : cette procédure pour vous sembler lourde, mais soyez assuré(e)s que ça vous permettra d'avancer beaucoup plus rapidement dans la résolution de la problématique.

Exemples de stratégies pour des tests unitaires:

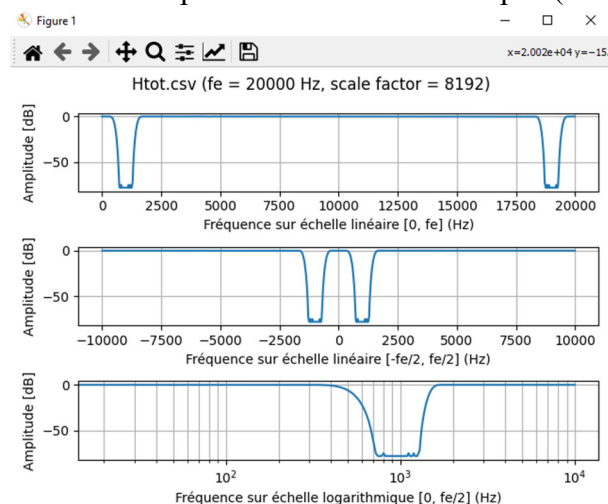
- 1) Vous ne pouvez pas visualiser des tableaux complexes dans DMCI, ex : $X[k]$, parce que ceux-ci sont composés d'une succession de paires de valeurs réelles et imaginaires, i.e. $X[0].re, X[0].im, X[1].re, X[1].im, \dots$ donc un graphique montrera les parties réelles et imaginaires intercalées de $X[k]$, ce qui n'est pas utile. Il faut plutôt visualiser le tableau de la norme des valeurs $|X[k]|^2$ (purement réel), calculé avec la fonction `calc_power_spectrum()`.

Attention: dans le filtrage FIR, `calc_power_spectrum()` est à utiliser pour le déverminage seulement. Comme la fonction utilise des *double*, elle va bousiller le fonctionnement en temps réel du code parce que son temps d'exécution sera très long. **Elle doit donc être mise en commentaire pour l'exécution en temps-réel.**

- 2) Quoique le plugin DMCI soit un bon outil pour visualiser les tableaux, l'axe horizontal n'indique que les indices des valeurs dans le tableau, ce qui est adéquat pour des tableaux de données temporelles. Pour visualiser des tableaux de données du domaine fréquentiel, il peut être utile de montrer ces données sur un graphique en fonction d'un axe fréquentiel en Hz, sur une échelle linéaire ou logarithmique, ce qui n'est pas possible avec DMCI.

Pour ce faire, MPLAB permet d'exporter des tableaux dans des fichiers .csv pour les traiter avec un logiciel externe comme Excel: dans la fenêtre "Variables", faites un right-droit sur la variable en question, puis *Export Data -> CSV File -> Hexadecimal Format*. Si la variable n'apparaît pas dans la fenêtre « Variables », il faut l'y rajouter en faisant un click-droit sur la variable et sélectionner « New Watch... ».

Pour des tableaux de valeurs complexes (ex : résultat d'une FFT), vous pouvez utiliser le script Python *S4GE-APP6-Plot-Complex-Csv.py* disponible sur le site web de l'APP. Le script lit un fichier .csv contenant un tableau de valeurs complexes exporté en **hexadécimal** de MPLAB et affiche la norme (amplitude) en fonction des fréquences. Voici un exemple pour le tableau *Htot* avec la bande passante du filtre H6 coupée (SW6 vers le bas) :



Exemple de gabarit de plan de vérification (les premières étapes sont **partiellement complétées** à titre d'exemples)

Plan de vérification

Membres de l'équipe :

A1 - Calcul de la résolution fréquentielle du spectre

- Description : calcul de la valeur de la variable *spectralResolution* (type double)
- Résultat attendu : $\Delta f = \dots$
- Test unitaire à exécuter :
- Résultat obtenu :

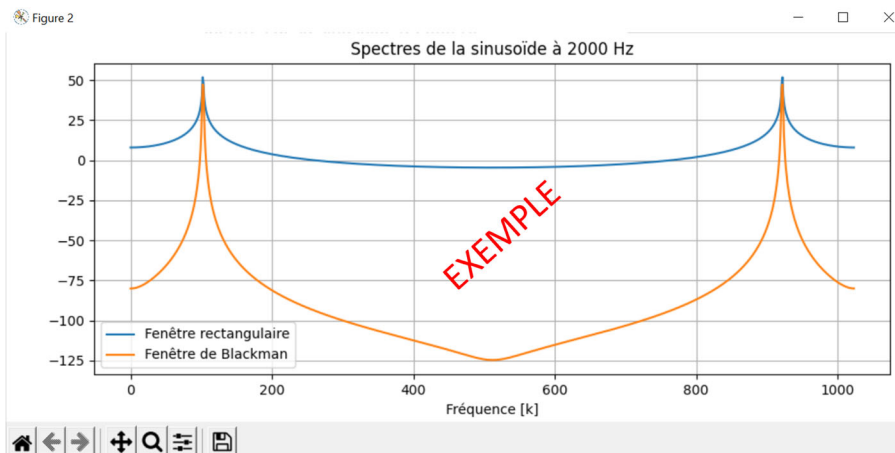
A2 & A3 – Calcul du spectre d'entrée, effet du fenêtrage

- Description : calcul de la TF du signal d'entrée avec la fonction *mips_????()*, calcul du module carré du spectre $|X[k]|^2$ en dB, stockage du résultat dans le tableau *debugBuffer1*, affichage avec DMCI.
- Résultats attendus, conditions du test: montrer des graphiques Python pour un signal test à 2 kHz, avec fenêtres rectangulaire et fenêtre de *Hanning*, sur des échelles de fréquence en log. Pour le cas de la fenêtre de *Hanning*, montrer aussi un graphique sur une échelle linéaire pour comparer avec le résultat affiché dans MPLAB avec DMCI.

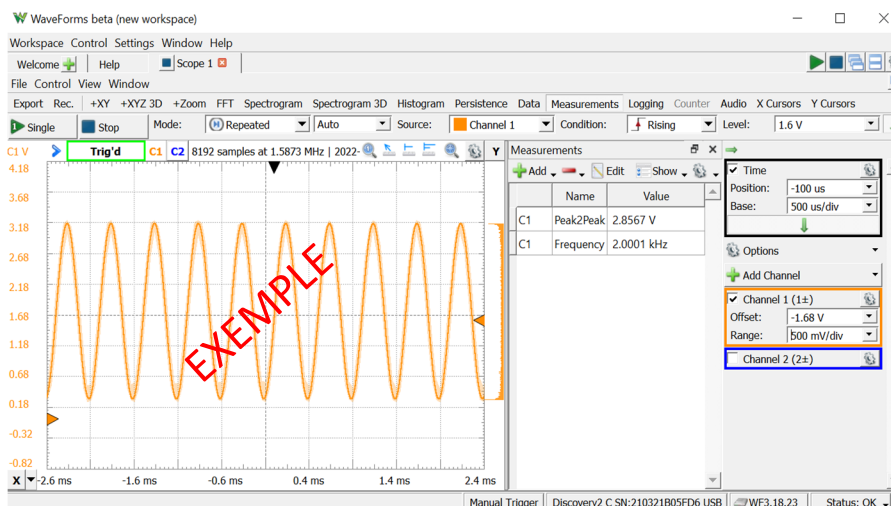
Progression suggérée et exemples de résultats pour rédiger, puis faire progresser, cette étape exemple du plan de test, au fur et à mesure que vous avancez :

- 1) Croquis à la main du spectre en amplitude d'un signal de 2 kHz, avec identification des indices k des composantes à fréquence positive et négative.
- 2) Simulation dans Python du spectre en amplitude, avec et sans fenêtrage : vérifiez que le fenêtrage a bien l'effet escompté (abaisser significativement le niveau de bruit).
- 3) Validation du signal d'entrée avec l'AD2.
- 4) Code C embarqué (sans fenêtrage, puis recompilation avec fenêtrage) :
 - a. Validation du signal d'entrée $x[n]$ avec DMCI, avant la FFT.
 - b. Validation du spectre $|X[k]|^2$ avec DMCI après la FFT, calculé avec la fonction *calc_power_spectrum()*.

- 2) Simulation dans Python du résultat attendu pour le spectre du signal sinusoïdal à 2 kHz, sans et avec fenêtrage :



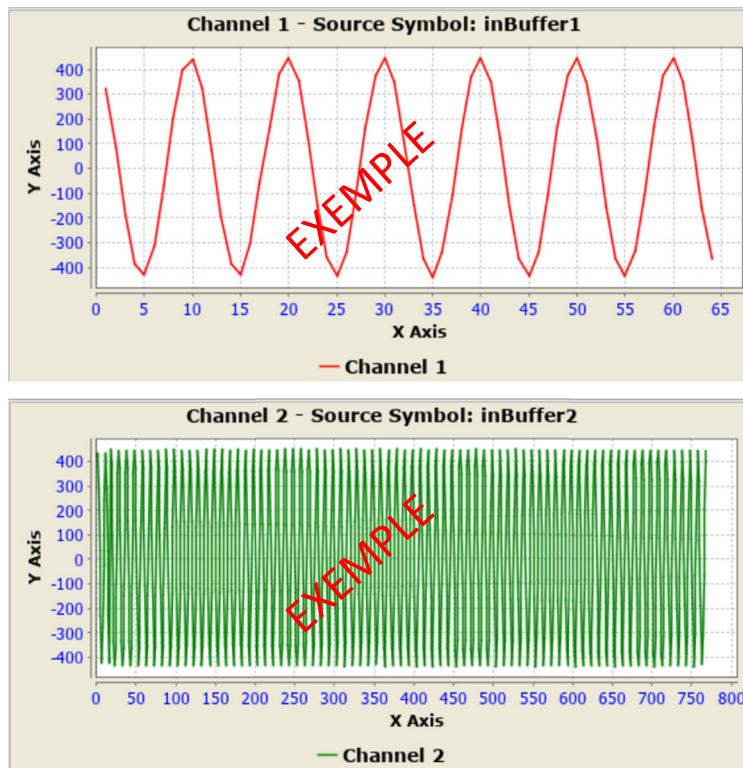
- 3) **Test unitaire** : validation du signal d'entrée avec l'AD2



Résultat : le signal a une apparence sinusoïdale et n'est pas saturé.

4) Code C embarqué (sans fenêtrage, avec fenêtrage) :

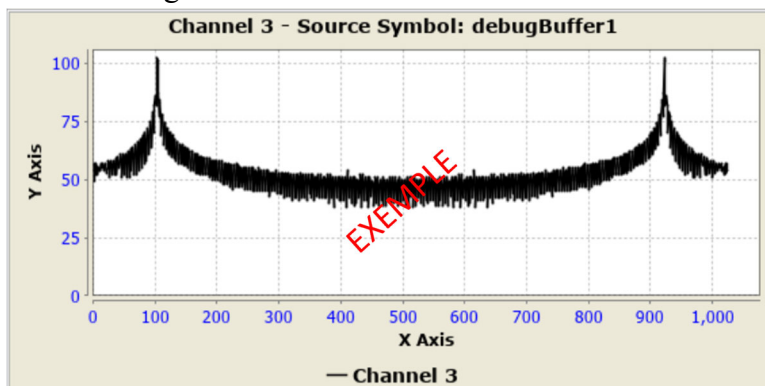
Test unitaire : validation du signal d'entrée dans DMCI (breakpoint à « DEBUG C »)



Résultat : le signal a une apparence sinusoïdale et n'est pas saturé (le graphique de *inBuffer1* « zoomé » sur les premiers 64 échantillons permet de confirmer ceci).

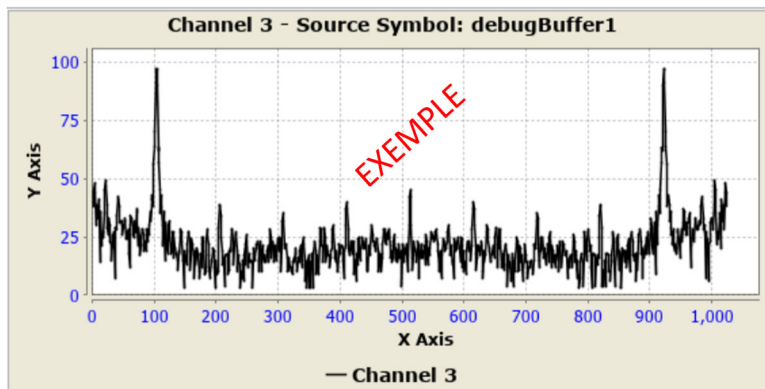
Test unitaire : avec un breakpoint à « DEBUG C » et SW1 vers le bas, calcul du spectre et résultat dans DMCI pour le signal de 2 kHz, comparaison avec le résultat de Python.

Sans fenêtrage :



Résultat de la comparaison avec la simulation en Python : les positions des pics coïncident ($k \approx 102$ pour la fréquence positive, $k \approx 922$ pour la fréquence négative) ainsi que le delta entre le pic et le niveau de bruit (~ 50 dB).

Avec fenêtrage :



Résultat de la comparaison avec la simulation en Python : les positions des pics coïncident, le delta entre le pic et le niveau de bruit est plus grand (~100 dB), mais quand même moins que Python qui fonctionne en *float*.

A4 – Calcul de l'indice de la fréquence à amplitude maximale

- 5) Description : calcul de la valeur de la variable *maxAmplFreq* (type int)
- 6) Résultat attendu, conditions du test (fréquence du signal d'entrée, etc.):
- 7) Test unitaire à exécuter :
- 8) Résultat obtenu :

B0 & B1 – Calcul de la FFT du signal d'entrée rallongé à 4N (méthode *overlap & save*)

- 9) Description : signal d'entrée rallongé à 4N dans le tableau *inFFT* (type : int32c, longueur : 4N), calcul de la FFT avec la fonction *mips_????()* dans le tableau *outFFT* (type : int32c, longueur : 4N).
- 10) Résultats attendus, conditions du test:

Test unitaire à exécuter :
- 11) Résultat obtenu :

B2 – Filtrage dans le domaine fréquentiel

12) Description : filtrage dans le domaine fréquentiel par la multiplication des transformées de Fourier

13) Résultats attendus, conditions du test:

14) Test unitaire à exécuter :

15) Résultat obtenu :

B3 & B4 – FFT inverse, extraction de la trame filtrée

...

C1 – Filtres IIR

...

3 Connaissances nouvelles à acquérir

3.1 GEL412 - Traitement numérique des signaux

Connaissances procédurales : QUOI

- Algorithme de la transformée de Fourier rapide.
- Transformée en z, propriétés.
- Filtres numériques linéaires : pôles et zéros, stabilité.
- Conception de filtres numériques IIR, familles de filtres.

Connaissances déclaratives : COMMENT

- Comprendre/utiliser les outils de transformées de transformée en z.
- Concevoir un filtre numérique linéaire et choisir parmi les familles de filtres.
- Analyser et comprendre la notion de stabilité de filtres numériques linéaires.
- Comprendre les différences entre le filtrage temporel et fréquentiel.

Connaissances conditionnelles : QUAND/POURQUOI

- Faire un choix judicieux de type (FIR/IIR) et de famille de filtre selon l'application.
- Savoir faire un choix judicieux de filtrage dans le domaine temporel ou fréquentiel.

3.2 GEL452 - Microcontrôleurs

Connaissances déclaratives : QUOI

- Périphériques internes et externes.
- Environnement et outils de développement logiciel; bonnes pratiques de programmation.
- Intégration logiciel-matériel.
- Traitement numérique des signaux sur microcontrôleur.

Connaissances procédurales – COMMENT

- Visualiser et modifier des variables.
- Exploiter les périphériques internes d'un microcontrôleur.
- Réaliser du calcul à point fixe.
- Utiliser une librairie.

Connaissances conditionnelles : QUAND/POURQUOI

- Répartition du calcul entre les interruptions et le programme principal

4 Références

4.1 Livre de référence

Understanding Digital Signal Processing, 3rd Edition, Richard G. Lyons (2010)

4.2 Logiciels utilisés

PyCharm (IDE Python), MPLAB X

4.3 Préparation aux activités (Lectures dans le livre de référence)

4.3.1 Procédural 1

- CHAPTER 6. INFINITE IMPULSE RESPONSE FILTERS
 - 6.1 An Introduction to Infinite Impulse Response Filters
 - 6.3 The Z-Transform
 - 6.4 Using the z-Transform to Analyze IIR Filters
 - 6.6 Alternate IIR Filter Structures
 - 6.8 Improving IIR Filters with Cascaded Structures
- CHAPTER 12. Digital Data Formats and Their Effects
 - 12.1.6 Fractional Binary Numbers

4.3.2 Laboratoire

- CHAPTER 4. THE FAST FOURIER TRANSFORM
 - 4.1 Relationship of the FFT to the DFT
 - 4.2 Hints on Using FFTs in Practice
- CHAPTER 13. DIGITAL SIGNAL PROCESSING TRICKS
 - 13.10 Fast FIR Filtering Using the FFT

4.3.3 Procédural 2

- CHAPTER 6. INFINITE IMPULSE RESPONSE FILTERS
 - 6.5 Using Poles and Zeros to Analyze IIR Filters
 - 6.11 Bilinear Transform IIR Filter Design Method
 - 6.13 A Brief Comparison of IIR and FIR Filters
- CHAPTER 4. THE FAST FOURIER TRANSFORM
 - 4.3 Derivation of the Radix-2 FFT Algorithm
 - 4.4 FFT Input/Output Data Index Bit Reversal

4.3.4 Problématique

- 13.6 Computing the Inverse FFT Using the Forward FFT
- 13.10 Fast FIR Filtering Using the FFT
- 13.41 Avoiding Overflow in Magnitude Computations

5 Formation à la pratique procédurale 1

Problème 1 – « Mise en bouche » (rencontre de retour sur l'APP5)

- Expliquez le lien entre l'équation à différence d'un filtre FIR, la réponse impulsionnelle du filtre, les coefficients de filtres b_j , et la convolution.
- Pourquoi programmer un tel filtre par convolution revient-il à programmer un calcul de moyenne pondérée des échantillons passés (filtre causal)?
- Quelle différence entre un filtre causal vs non-causal?
- Pourquoi faire des transformations (ex : Laplace, Fourier, Tz)?
- Que représente une exponentielle complexe et pourquoi les utiliser en ingénierie? Que représentent des fréquences négatives dans le domaine spectral (Fourier)?
- Quelle est la conséquence dans le domaine de Fourier d'échantillonner un signal dans le domaine temporel (i.e. le signal est « discret ») et inversement?
- Comment interpréter l'axe fréquentiel suite à l'algorithme FFT, où est le tableau des valeurs fréquentielles qui correspondent aux indices k ? à quoi sert la fonction `fftshift()`?
- Quelle est l'importance de la paire de transformées « rectangle $\leftrightarrow \sin(x)/x$ » et quel rapport avec l'effet de fuite? À quoi sert le fenêtrage?
- Qu'est le théorème de convolution et lien avec le filtrage en temps et en fréquence?
- Pourquoi filtrer dans le domaine temporel vs fréquentiel? Plus rapide? Plus précis?
- Que sont les indices n , N , k , K , vus dans l'APP5?

Problème 2 - Analyse de filtres IIR

- (a) Quelle est la définition de la transformée Tz, quelles sont ses deux utilités, et quelle est sa propriété la plus utile?
- (b) À partir de l'équation aux différences générale d'un filtre IIR et en utilisant la propriété de décalage de la Tz, trouvez la fonction de transfert du filtre en puissance négatives de z , puis en puissances positives de z .
- (c) Quelles est la différence entre les conventions de signe des équations aux différences des filtres IIR dans la plupart des livres de référence versus celle utilisée dans les outils numériques comme Python ou Matlab et quelle est l'implication pour la programmation? Si $y[n] = x[n] + 3x[n-1] + 5y[n-1]$, quelle sera la valeur de a_1 selon les deux conventions ?
- (d) Donnez la fonction de transfert $H(z)$ du filtre suivant :

$$y[n] = x[n] + 0.9 y[n-1]$$

- (e) Affichez le lieu des pôles et des zéros de ce filtre, en incluant le cercle de rayon unitaire dans le plan complexe, est-ce que le filtre est stable ?
- (f) Obtenez la fonction de transfert *harmonique*, $H(e^{j\bar{\omega}})$, à partir de $H(z)$, où $\bar{\omega} = 2\pi f/f_c$

- (g) En utilisant la réponse de la question précédente, donnez la réponse de ce filtre en régime permanent à la sinusoïde $x[n] = 10 \cos(0.5 \pi n)$

Problème 3 – Structure alternative de filtre IIR

- a) À partir de l'équation aux différences d'un filtre IIR d'ordre 2 ayant le même nombre de termes en x et en y , dessinez la structure en forme Direct I. Quel est l'avantage principal de cette forme sur un CPU à calcul sur nombres entiers comme le PIC32?
- b) À partir de la fonction de transfert ci-dessous d'un filtre IIR d'ordre 2 où $N = M = 2$ (même nombre de termes en x et en y), simplifiez la structure du filtre.

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{j=0}^N b_j z^{-j}}{1 - \sum_{i=1}^M a_i z^{-i}} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 - a_1 z^{-1} - a_2 z^{-2}}$$

- 1) Commencez par trouver l'expression de $\frac{Y(z)}{W(z)}$ et $\frac{W(z)}{X(z)}$ dans l'équation suivante :

$$\frac{Y(z)}{X(z)} = \frac{Y(z)}{W(z)} \times \frac{W(z)}{X(z)} = \text{num}\{H(z)\} \times \frac{1}{\text{den}\{H(z)\}}$$

Trouvez ensuite les équations aux différences correspondantes ($y[n]=\dots$ et $w[n]=\dots$) puis dessinez la structure qui en résulte.

- 2) Quel est le nom de cette structure et quels sont ses avantages et désavantages?
- c) Le livre de référence décrit la forme Direct II *Transpose* de l'IIR issue de l'application du théorème de transposition (théorie des graphes) à la forme Direct II « de base ».
- 1) Dessinez la structure IIR de forme Direct II *Transpose* d'ordre 2 et montrez qu'elle correspond aux équations suivantes :

$$\begin{aligned} y[n] &= b_0 x[n] + v[n-1] \\ v[n] &= b_1 x[n] + a_1 y[n] + u[n-1] \\ u[n] &= b_2 x[n] + a_2 y[n] \end{aligned}$$

où $u[n]$ et $v[n]$ sont des signaux intermédiaires résultant de sommations.

- 2) Prouvez que ces équations correspondent bien à l'équation aux différences pour un IIR d'ordre 2.
- 3) Pourquoi la structure Direct II *Transpose* est-elle préférable à la forme Direct II « de base » de l'exercice précédent?
- d) Expliquez comment implémenter un filtre d'ordre élevé en une cascade de filtres IIR Biquad. Pourquoi cette forme est-elle préférée ? Quel est le lien avec problématique?

Problème 4 - Notation QX.Y à point fixe en compléments à 2


- a) Expliquer ce qu'est la notation de complément à 2 et ce qu'est que la « répétition du bit de signe », notamment pour un décalage vers la droite (*arithmetic shift right*). Donnez les représentations de 256 et -256 sur 8 bits et 16 bits.
- b) À quoi sert la notation QXY? Expliquer comment convertir un nombre décimal en notation QXY. Définir les plages dynamiques et la précision pour les représentations fractionnaire signée à point fixe en complément à 2 sur 16 bits suivantes: Q3.12 et Q0.15 (cette dernière est habituellement indiquée par Q15).
- c) Convertir le nombre décimal 2.5695 dans les représentations QX.Y suivantes sur 16 bits et exprimer le résultat en hexadécimal :
- Q12.3
 - Q5.10
 - Q1.14
- d) Expliquer les procédures requises et caractéristiques particulières pour les opérations d'addition et de multiplication sur les nombres en notation QX.Y.
- e) Donnez le format QX.Y résultat des opérations suivantes sur 16 bits :
- $Q3.12 + Q1.14$
 - $Q3.11 + Q1.13$
 - $Q2.5 \times Q1.7$
 - $Q2.5 \times Q1.8$
- f) Dans un calcul à étapes multiples en format QXY, il faut tenir compte du format du résultat à chaque étape pour, d'une part, s'assurer d'éviter les dépassements à chaque étape, et d'autre part, de savoir comment convertir le résultat final dans le format désiré.
- 1) Dans les étapes de calcul sur 32 bits d'un filtre IIR avec la méthode DIRECT II *transposée* ci-dessous, où $x[n]$ et $y[n]$ sont en format Q15 et les coefficients en Q2.13, indiquez le format de chacun des termes et celui du résultat.
- $$y[n] = b_0 x[n] + v[n-1]$$
- $$v[n] = b_1 x[n] + a_1 y[n] + u[n-1]$$
- $$u[n] = b_2 x[n] + a_2 y[n]$$
- Pourquoi les calculs de $y[n]$, $v[n]$, et $u[n]$ doivent-ils *absolument* se faire dans cet ordre dans votre programme en C?
 - À laquelle des trois étapes doit-on diviser le résultat par 2^{13} ?
- 2) Si on considère qu'un convertisseur A/N de 16 bits numérise les données en format normalisé sur la plage dynamique $[-1, 1[$, dans quel format sont les données?
- 3) Dans la problématique, pourquoi considère-t-on que les échantillons d'entrée soient en format Q15 malgré le fait que le convertisseur A/N du PIC32 n'ait que 10 bits?

6 Formation à la pratique en laboratoire

Problème 1 : conception et analyse de filtre IIR avec un logiciel de simulation (Python, environnement PyCharm)

- Pour cet exercice avec Python, faites référence au besoin à la documentation sur le site web de la session sous la rubrique « Manuels, logiciels », en particulier : *Matlab-style IIR filter design* sur le site de *scipy.signal*.
- Les librairies (*packages*) Python les plus utiles pour le traitement du signal sont *numpy*, *scipy*, et *matplotlib*, en particulier les "subpackages" *matplotlib.pyplot* et *scipy.signal*, installés avec :

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import signal
```

- PyCharm comprend un puissant outil de déverminage (*debugger*) qu'il est impératif d'apprendre à utiliser. L'exécution d'un script Python avec le débogueur est déclenchée avec l'icône  (si l'icône n'apparaît pas initialement dans l'interface PyCharm, il faut cocher l'option *Navigation bar* dans le menu *View/Appearance*). Les icônes ci-dessous dans la barre de navigation du débogueur permettent de démarrer/poursuivre/arrêter l'exécution selon les *breakpoints* :



- Les *breakpoints* sont activés/désactivés en faisant un *click-gauche* dans la marge de gauche des lignes de code et sont indiqués par un point rouge dans la marge. *Aux pauses d'exécution, les valeurs de toutes les variables calculées à date sont indiquées en gris à mêmes les ligne de code.* Vous pouvez aussi visualiser les variables dans la fenêtre *Variables* au bas de la fenêtre PyCharm.
- Si vous faites une installation de PyCharm sur votre ordinateur personnel, il faut désactiver le « *Scientific Mode* » de PyCharm afin de visualiser les graphiques dans des fenêtres indépendantes et interactives:

View | Scientific Mode, *Disable*

Settings | Tools | Python Scientific | Show plots in tool window, *Disable*

Astuces et conseils :

- Consultez l'Annexe D – Trucs Python.
- Fonctions utiles pour afficher graphiquement des résultats : *plt.figure()*, *plt.plot()*, *plt.subplot()*, *plt.semilogx()*, *plt.title()*, *plt.xlabel()*, *plt.ylabel()*, *plt.grid()*, *plt.tight_layout()*, *plt.xlim()*, *plt.ylim()*, *plt.legend()*.
- La fonction *plt.show()* est requise pour afficher les figures, i.e, les fonctions comme *plt.plot()* construisent un objet de type « figure » mais ne les affichent pas explicitement.
- Insérez la fonction *plt.ion()* au début des scripts afin de visualiser des figures lors de l'exécution avec le debugger. Cette fonction permet à l'exécution de continuer après une fonction d'affichage de figure, ces fonctions étant normalement « bloquantes » (le script est normalement mis en pause).
- Quand l'exécution d'un script Python se termine, toutes les figures sont fermées automatiquement. Afin de garder à l'écran les figures générées par un script, prenez l'habitude de mettre un *breakpoint* à la dernière ligne du script contenant une fonction bidon, par exemple « *print("Whoa!")* ».

a) **Filtrage IIR dans le domaine temporel** : conception d'un filtre IIR passe-bande de type elliptique qui répond aux caractéristiques suivantes (fonction *scipy.signal.ellip*):

- i. Fréquence d'échantillonnage = 20 kHz
- ii. Bande passante : 900 Hz – 1100 Hz
- iii. Ordre : 2 (le passe-bande aura un ordre 4)
- iv. Ronflement (*ripple*) maximum dans la bande passante : 1 dB
- v. Atténuation dans la bande stop : 40 dB

NB : Il n'y a pas de définition universelle de la fréquence de coupure d'un filtre, f_c . Dans le cas de filtres IIR comme les filtres elliptiques ou Tchebychev (type I), f_c est définie comme la fréquence à laquelle le gain dans la bande de transition passe en deçà de la valeur du ronflement sous le gain unitaire dans la bande passante.

Téléchargez le script Python de départ fourni S4GE-APP6-Laboratoire-FiltreElliptique.py du site web de l'APP.

1) Affichez le module (dB) de la réponse en fréquence du filtre (fonctions *scipy.signal.freqz*, *matplotlib.semilogx*, *numpy.log10*).

NB : Quand vous montrez la réponse en fréquences avec *matplotlib.semilogx()*, n'oubliez pas de spécifier le tableau des fréquences correspondantes sur l'axe horizontal comme premier paramètre de la fonction, par exemple le tableau des fréquences renvoyé par la fonction *signal.sosfreqz()*. Autrement, la fonction *matplotlib.semilogx()* utilisera par défaut la suite d'indices des échantillons, i.e. 0, 1, 2, 3, sur l'axe horizontal... dans ce cas, les fréquences de coupure du filtre sembleront être erronées, mais c'est l'axe des fréquences qui sera incorrect.

- 2) Calculez ses pôles et ses zéros (commande *numpy.roots*). Sont-ils consécutifs avec la forme de la réponse en fréquence? Essayez aussi le programme *zplane.py* disponible sur le site web de l'APP (copiez le fichier *zplane.py* dans le même répertoire que votre script et ajoutez à votre script la ligne « *from zplane import zplane* ») dans les déclarations, puis « *zplane(b, a)* », précédé au besoin par *plt.figure()* pour créer une nouvelle figure.
- 3) Affichez la réponse impulsionnelle « tronquée » (la réponse impulsionnelle d'un filtre IIR est normalement de longueur infinie): créez une impulsion de longueur de $N = 1000$, puis calculez la réponse correspondante du filtre laquelle sera de longueur N (commandes *signal.unit_impulse* et *signal.lfilter*)
- 4) Calculez les coefficients du filtre en format SOS (*second order sections*) en réutilisant la commande *signal.ellip()*, mais cette fois en spécifiant explicitement la valeur appropriée pour le paramètre *output*, puis affichez le module (dB) de sa réponse en fréquence (commande *signal.sosfreqz*).

NB : Pour montrer un graphique avec des *peaks* très fins avec *signal.sosfreqz()*, il faut avoir suffisamment de points dans le graphique pour bien représenter les *peaks*, sans ça ils risquent de tomber entre deux points et sembler ne pas être au bon endroit ou ne pas avoir la bonne profondeur. Le paramètre *worN* permet de spécifier le nombre de points dans le graphique. La valeur par défaut de *worN* est souvent très basse (512). Si vous avez un graphique avec des *peaks* fins, spécifiez une valeur beaucoup plus élevée, par exemple « *worN = 100000* ».

- 5) Convertissez les coefficients a et b et SOS du filtre en point-fixe (format Q2.13) en arrondissant à l'entier près (commandes *numpy.power*, *numpy.round*). Comparez la réponse en fréquence avec le cas précédant sur un même graphique.

NB : en multipliant les coefficients du filtre par 2^Y dans la conversion en format QXY, les coefficients a_0 des différentes sections SOS seront passés à 2^Y , ce qui est différent de la situation normale où $a_0 = 1$. Mathématiquement ça ne pose pas de problème parce les coefficients b_i et a_j aux numérateurs et dénominateurs des différentes sections SOS auront tous été multipliés par 2^Y , le filtre « demeure le même » sauf pour l'information perdue par arrondissement ([[]]) des coefficients :

$$H_{QXY}(z) = \frac{\sum_{i=0}^N \llbracket 2^Y b_i \rrbracket z^{-i}}{\sum_{j=0}^N \llbracket 2^Y a_j \rrbracket z^{-j}}$$

Cependant, la fonction *signal.sosfreqz()* n'aime pas des valeurs de $a_0 \neq 1$ donc il faut rediviser les coefficients SOS par 2^Y pour utiliser cette fonction (l'effet de perte de précision qu'on veut démontrer demeurera parce l'info perdue des coefficients par arrondissement n'est pas restaurée en divisant par 2^Y).

b) **Filtrage FIR « rapide » par blocs dans le domaine fréquentiel** (brique de base de la méthode *overlap-and-save*)

Avec la commande `signal.firwin` (voir Annexe D – Trucs Python) pour une fréquence d'échantillonnage 20 kHz, faites la conception avec la méthode des fenêtres (*Hamming*) de deux filtres aux caractéristiques suivantes de longueur N ou $N-1$ selon le cas ($N = 512$):

- i. Filtre passe-bas, fréquence de coupure = 1000 Hz
- ii. Filtre passe-haut, fréquence de coupure = 950 Hz

NB : Tel qu'écrit plus haut, il n'y a pas de définition universelle pour la fréquence de coupure d'un filtre, f_c . Dans plusieurs cas, f_c est définie comme la fréquence à laquelle le gain en *puissance* est de $\frac{1}{2}$, soit le point de -3 dB. Dans d'autres, comme c'est le cas pour la commande `signal.firwin()`, f_c est définie comme la fréquence à laquelle le gain en *amplitude* est de $\frac{1}{2}$, soit le point de -6 dB.

- 1) Affichez les réponses impulsionnelles des deux filtres.
- 2) Ajoutez des zéros à la fin des réponses impulsionnelles (*zero padding*) pour que la longueur totale de celles-ci soit de $4N$ (fonction `np.append`), calculez les fonctions de transfert des 2 filtres (fonction `numpy.fft.fft`). Affichez sur un même graphique le module des réponses en fréquence en dB des 2 filtres aux fréquences non-négatives seulement (fonctions `numpy.arange`, `numpy.abs`, `numpy.log10`). Sur un graphique séparé, affichez le module de la *somme* des réponses en fréquence des 2 filtres.

Quel est l'intervalle d'échantillonnage en fréquence, Δf ?

- 3) Créez deux sinusoïdes de longueur $4N$ de fréquences $f_1 = 200$ Hz et $f_2 = 2000$ Hz (fonctions `numpy.pi`, `numpy.sin`) et affichez la norme de leurs transformées de Fourier en dB sur un graphique aux fréquences non-négatives seulement. Quand faut-il multiplier le `log()` par 10 versus par 20 pour le calcul en dB?
- 4) Filtrez chacun des signaux dans le domaine fréquentiel en multipliant leurs transformées de Fourier par celles des filtres, calculez ensuite les transformées de Fourier inverses (`numpy.fft.ifft`) et affichez les signaux filtrés.

NB : comme le résultat de la FFT^{-1} est complexe, la composante réelle des tableaux `numpy` résultants sera le résultat (signal filtré) et la composante imaginaire contiendra du « bruit numérique ». Donc, pour montrer les signaux filtrés sur des graphiques, extrayez la composante réelle des tableaux `numpy` avec « *.real* », par exemple `y.real`, où `y` est le tableau `numpy` résultant de la FFT^{-1} .

Que remarquez-vous après les premiers $N/2$ échantillons ? Comparez avec un signal dont la fréquence, f , est un multiple exact de Δf , par exemple à une fréquence de $20 \times$ l'intervalle d'échantillonnage en fréquence, soit $f = 20\Delta f$. Que remarquez-vous comme différence dans la représentation spectrale comparativement au signal à 200 Hz? Quel rapport avec l'*effet de fuite*?

Problème 2 : projet MPLAB sur la carte MX3, usage de DMCI

Élaboration d'un nouveau projet dans MPLAB avec le plugin MCC pour visualiser avec le plugin DMCI une sinusoïde numérisée par le Pmod A.

- a) Créez le projet en suivant les instructions dans le *Guide d'utilisation MCC* disponible sur le site web de l'APP.
- b) Créez un nouveau fichier *main.h* avec le contenu suivant, que vous ajoutez au projet (*click-droit* sur le sous-répertoire *Header Files*, <Add Existing Item...>):

```
#ifndef _MAIN_H
#define _MAIN_H

#define BUFSIZ 512
extern int32_t inbuffer[], outbuffer[];
extern bool bufferFull;

#endif //_MAIN_H
```

- c) Ajoutez des déclarations de *main.h* (**en gras**) aux endroits requis:

- *main.c*:

```
#include "mcc_generated_files/mcc.h"
#include "main.h"
```

- *adc1.c*:

```
#include "adc1.h"
#include "../main.h"
```

- d) Ajoutez le code **en gras** à la fonction *ADC_1* (*void*) dans le fichier *adc1.c*:

```
void __ISR(_ADC_VECTOR, IPL1AUTO) ADC_1(void)
{
    static int n = 0;

    // Load ADC sample into input buffer, scale from [-512,511]
    // to [0,PR2] and write to output buffer
    inbuffer[n] = ADC1BUF0;
    outbuffer[n] = (inbuffer[n] + 512) * PR2 >> 10;

    // Write output sample to the PWM
    OC1_PWMPulseWidthSet(outbuffer[n]);

    // If buffer full, reset counter, flag reset to main program
    if (n++ == BUFSIZ) {
        n = 0;
        bufferFull = true;
    }

    // clear ADC interrupt flag
    IFS0CLR= 1 << _IFS0_AD1IF_POSITION;
}
```

- e) Dans le fichier *main.c*, ajoutez les déclarations suivantes avant la fonction *main()* :

```
int32_t inbuffer[BUFSIZ], outbuffer[BUFSIZ];
bool bufferFull = false;
```

- f) Dans le fichier *main.c*, remplacez le contenu de la fonction *main()*:

```
int main(void)
{
    // initialize the device
    SYSTEM_Initialize();
    BMXCONbits.BMXWSDRM = 0;
    CHECONbits.PFMWS = 3;
    CHECONbits.PREFEN = 2;
    TMR2_Start();
    TMR3_Start();

    while (1)
    {
        if (bufferFull) {
            bufferFull = false;
        }
    }

    return -1;
}
```

- g) Compilez le projet dans MPLABX, assurez-vous qu'il n'y a pas d'erreurs.
- h) Branchez le module de conditionnement de signal d'entrée fourni au connecteur PmodA et une source audio au connecteur 3.5 mm du module, par exemple depuis votre ordinateur avec le *Online Tone Generator* en ligne (lien sur le site web de l'APP).

NB : Assurez-vous de valider le signal d'entrée avec votre *Analog Discovery 2* selon les instructions dans l'Annexe B – Carte MX3 et modules Pmods.

- i) Avec votre programme en exécution continue, branchez le module de conditionnement de signal de sortie fourni au connecteur PmodB. Avec votre *Analog Discovery 2*, monitorisez le signal au test point *Out Filt* pour ajuster l'amplitude de sortie au même niveau que l'entrée avec le potentiomètre du module selon les instructions dans l'Annexe B – Carte MX3 et modules Pmods.

NB: pour mesurer un signal de sortie sur le PmodB, le programme doit forcément exécuter en continu sur la carte MX3 afin d'écrire des valeurs dans le PWM, sans être arrêté à un *breakpoint*.

j) Exécutez votre programme dans MPLABX en mode *Debug*. Avec un *breakpoint* judicieusement placé à la ligne "**bufferFull = false;**" dans la fonction *main()*, utilisez le plugin DMCI pour visualiser graphiquement le contenu des tableaux *inbuffer* et *outbuffer* :

1. Dans le menu *Tools*, sélectionnez <Embedded><DMCI><DMCI Window> pour faire apparaître la fenêtre DMCI, puis sélectionnez l'onglet *Dynamic Data View*
2. Avec un click-droit sur un des 4 graphique, sélectionnez *Configure Data Source*, puis dans la fenêtre *popup* qui apparaît, sélectionnez *Arrays Only* dans le menu déroulant des *Globals Symbols*, puis choisissez *inbuffer* dans le menu déroulant des variables.

NB : Pour pouvoir configurer DMCI, il faut avoir fait rouler le debugger au moins une fois pour que les variables du programme soient connues par le plugin.

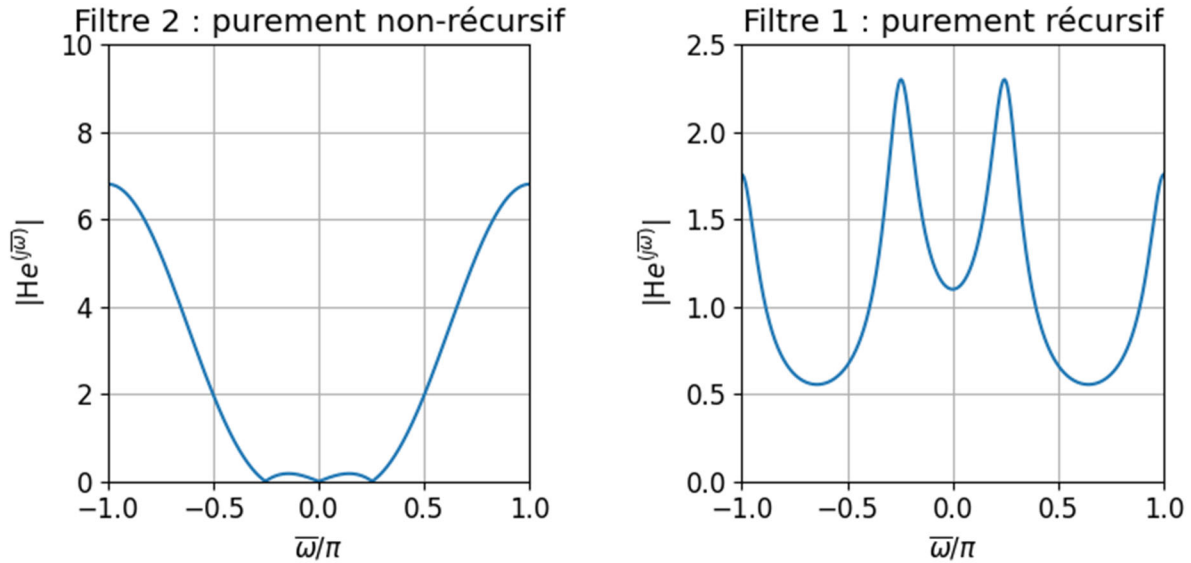
REMARQUE : la « distribution » des variables globales (ex : *inbuffer*, *bufferFull*) et des symboles (ex : *BUFSIZ*) via un fichier *.h* commun (ex : *main.h*) est un exemple de *bonnes pratiques* en programmation qui vous est fortement recommandé. Cette méthode permet au compilateur de travailler pour vous à valider la cohérence des références aux variables globales et d'assurer des définitions uniques pour les symboles partagés à travers tous les fichiers où ceux-ci sont utilisés. Autrement, par exemple, des définitions incohérentes multiples d'un même symbole dans différents fichiers (ex : *BUFSIZ*=128 dans l'un et *BUFSIZ*=256 dans l'autre) ne causeront pas d'erreurs de compilation mais causeront des problèmes d'écriture aberrante en mémoire qui se manifesteront seulement au moment de l'exécution et qui sont très difficiles à diagnostiquer.

7 Formation à la pratique procédurale 2

Problème 1 – Filtrage FIR rapide dans le domaine fréquentiel par la méthode *Overlap & save*

- a) Réchauffement...
 - 1) Si un signal est échantillonné dans un domaine, par exemple dans le domaine temporel, qu'est-ce que ça implique pour le domaine transformé, par exemple le domaine fréquentiel?
 - 2) Pourquoi est-ce qu'un spectre défini ou calculé dans un outil numérique comme Python ou Matlab représente-il nécessairement un signal périodique?
 - 3) Qu'est-ce que le théorème de convolution?
 - 4) Quelle est la longueur du signal de sortie $y[n]$ résultant de la convolution dans le domaine temporel entre un signal d'entrée $x[n]$ de longueur $3N$ et la réponse impulsionnelle d'un filtre $h[n]$ de longueur N ?
- b) Pourquoi le calcul de la convolution dans le domaine spectral avec un outil numérique ne donnera-t-il pas le même résultat qu'un calcul dans le domaine temporel?
- c) Expliquez la méthode *overlap-and-save*.
- d) Selon le livre de référence, à partir de quelle longueur de réponse impulsionnelle le filtrage dans le domaine fréquentiel devient-il plus efficace que le filtrage dans le domaine temporel?

Problème 2 - Analyse de filtres numériques par la méthode géométrique



La figure ci-dessus montre le module de la réponse en fréquence $|H(e^{j\bar{\omega}})|$ de deux filtres numériques. Notez bien l'échelle et les unités de l'axe des fréquences. On sait que chacun de ces filtres est d'ordre 3. On sait aussi que le filtre 1 est purement non-récurusif et que le filtre 2 est purement récurusif. Pour simplifier, on présume que dans un filtre donné, le module des tous les zéros est le même et le module de tous les pôles est le même.

Pour chaque filtre :

- Affichez les positions des pôles & zéros dans le plan complexe et donnez leurs formes mathématiques le plus précisément possible
- Donnez l'équations aux différences, sans valeurs numériques pour les coefficients
- À partir des expressions pour la fonction de transfert, calculez les valeurs numériques des pôles et les zéros (expliquez pourquoi un filtre aura toujours autant de pôles que de zéros et qu'un FIR a des pôles répétés « cachés » à l'origine)
- Donnez l'équations aux différences avec les valeurs numériques des coefficients

Problème 3 - Transformation bilinéaire

- Montrez que la transformation bilinéaire, définie ci-dessous, convertit tout point s dans le demi-plan gauche du plan complexe en un point z à l'intérieur du cercle de rayon unitaire.

$$s = \frac{2}{T} \frac{z - 1}{z + 1}$$

où T est la période d'échantillonnage, i.e. $T = 1/F_e$.

Quel est l'impact de ce « mapping » sur la stabilité du filtre numérique obtenu ? Commencez par trouver la fonction inverse $z(s)$, comme dans l'équation 6-106 du livre de référence, puis considérez l'expression pour $|z|$ pour $s = \sigma + j\omega$ avec $\sigma < 0$ et $\sigma > 0$.

b) Gauchissement des fréquences

Un filtre analogique passe-bande doit être transformé en un filtre numérique $H(z)$ par la transformation bilinéaire. Ce filtre a les caractéristiques suivantes :

- gain dans la bande passante = 0 dB \pm 3 dB
- bande passante allant de $f_{lp} = 4000$ Hz à $f_{hp} = 6000$ Hz
- gain maximum dans la bande coupée = -60 dB
- bande coupée pour $f_{lc} < 2000$ Hz et $f_{hc} > 8000$ Hz
- fréquence d'échantillonnage $f_e = 32000$ Hz

Appliquez le gauchissement des fréquences relatif à la transformation bilinéaire (équation 6-115 dans le livre de référence) pour obtenir les fréquences analogiques f_{lp}' , f_{hp}' , f_{lc}' et f_{hc}' qui serviront à définir la fonction de transfert analogique $H(s)$ à laquelle sera appliquée la transformation bilinéaire. Dessinez la réponse en fréquences des deux filtres.

NB : si on appliquait *directement* la transformation bilinéaire à une fonction de transfert $H(s)$ avec les caractéristiques ci-dessus, soit *sans appliquer préalablement le gauchissement des fréquences*, le filtre numérique $H(z)$ obtenu n'aurait pas les bonnes fréquences de coupure.

c) Conception d'un filtre IIR par la méthode de la transformation bilinéaire :

La fonction de transfert d'un filtre *Butterworth* analogique passe-bas d'ordre 1 et de fréquence de coupure de 500 Hz est donnée par:

$$H(s) = \frac{1}{1 + s/\omega_c}$$

où ω_c est la fréquence de coupure en rads/s. Notez que la fréquence de coupure correspond dans le cas d'un filtre *Butterworth* au point de -3 dB ($s = j\omega_c$, gain en puissance = $\frac{1}{2}$).

Appliquez la méthode de la transformation bilinéaire en tenant compte du gauchissement des fréquences, avec une fréquence d'échantillonnage de 8000 Hz pour obtenir l'équation à différences du filtre.

- Pourquoi s'assurer d'avoir un coefficient de « 1 » dans le premier terme au dénominateur?
- Où sont les pôles et les zéros de ce filtre ?

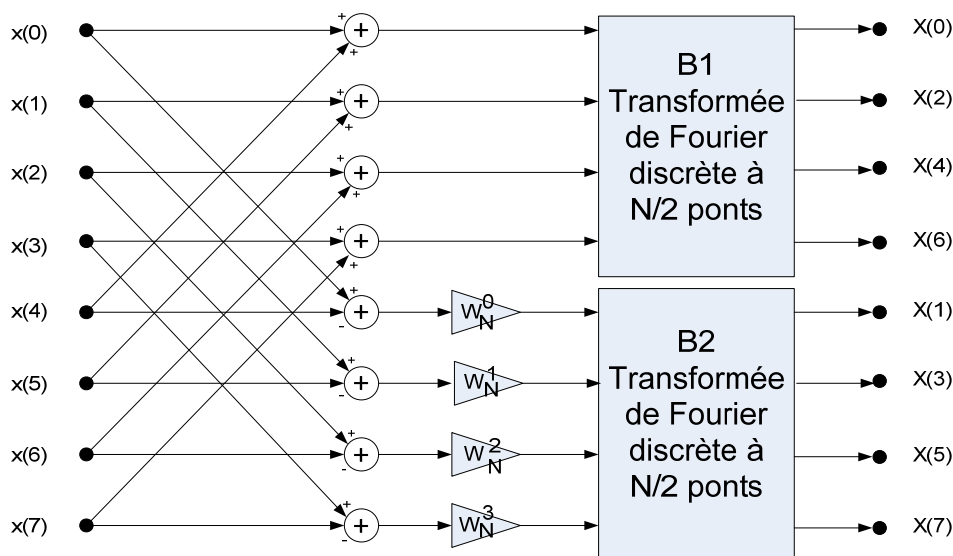
Problème 4 - FFT radix-2

Sachant que la FFT peut être réalisée par l'algorithme de décimation en fréquences (*decimation-in-frequency, DIF*) défini par les équations ci-dessous :

$$X(2k) = \sum_{n=0}^{\left(\frac{N}{2}\right)-1} \left[x(n) + x\left(n + \frac{N}{2}\right) \right] W_{N/2}^{nk} \quad (B1)$$

$$X(2k + 1) = \sum_{n=0}^{\left(\frac{N}{2}\right)-1} \left[x(n) - x\left(n + \frac{N}{2}\right) \right] W_N^n W_{N/2}^{nk} \quad (B2)$$

Et que ces deux équations sont utilisées comme suit pour réaliser une FFT sur un vecteur de 8 échantillons complexes :



Et sachant que les blocs B1 et B2 représentent la transformée de Fourier discrète (TFD) définie par l'équation :

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi nk/N} \text{ (TFD)}$$

- Que représente la variable $x(n)$ de la TFD pour le bloc B1 et le bloc B2 ?
- D'où vient l'augmentation de vitesse $O(N \log N)$ de l'algorithme FFT ?
- Qu'est-ce que l'ordre *bit reversed* ?
- Que sont les *twiddle factors* W ?

NB : la figure 4-2 à la page 144 du livre de référence représente une autre forme de l'algorithme FFT, soit la décimation en temps (*decimation-in-time, DIT*), qui donne le même résultat que le *DIF*. La seule différence est que l'ordre *bit reversed* est appliqué aux échantillons d'entrée dans le *DIT* au lieu des échantillons spectraux dans le cas de l'algorithme *DIF*.

8 Évaluation de la Problématique

		GEL412			GEL452	
		GEL412-C1	GEL412-C2	GEL412-C3	GEL452-C1	GEL452-C2
		Analyser des signaux à temps discret	Déterminer la réponse d'un filtre numérique	Concevoir un filtre numérique	Programmer un microcontrôleur	Méthodologie développement systèmes sur µC
	Description	33	33	34	20	20
Analyse de signaux dans le domaine fréquentiel	MPLAB (A1): Calcul de la résolution spectrale		8			
	PYTHON: Démonstration du fenêtrage sur l'effet de fuite dans le calcul du spectre: graphique de $ X ^2$ d'un signal sinusoïdal à une fréquence judicieusement choisie, avec et sans fenêtrage	23				
	MPLAB (A2, A3, A4): Graphique de $ X ^2$ d'un signal sinusoïdal fenêtré avec le plugin DMCI, identification de la fréquence en Hz de la composante à amplitude maximale				6	
	Qualité de la discussion dans le plan de test					7
Filtres FIR	PYTHON: Graphiques superposés des fonctions de transfert des filtres FIR (amplitude seulement), écriture des coefficients en notation QXY dans le fichier filterFIRcoeffs.h			17		
	MPLAB (B0-B4, 19/24 pts à la validation): Démonstration avec DMCI du fonctionnement des filtres FIR: montrez des graphiques de signaux filtrés par deux filtres FIR de votre choix (autre que H7) à 3 fréquences pertinentes pour chacun ($f < f_c$, $f = f_c$, $f > f_c$). NB: vous pouvez faire la démonstration du 2ième filtre avec l'AD2 au lieu de DMCI.	5	15		4	
	Qualité de la discussion dans le plan de test					7
Temps-réel	Analog Discovery: Calcul du temps disponible pour le traitement d'une trame dans la boucle principale et des proportions utilisées pour le filtrage FIR et le calcul du spectre				4	
Filtre IIR	PYTHON: Graphiques superposés des fonctions de transfert (gain seulement) du filtre IIR demandé pour des coefficients exacts et en format Q2.13, ainsi que pour le filtre "limité en ressources" (SOS en Q2.5)			17		
	MPLAB (C1, 14/19 points à la validation): Démonstration avec DMCI du fonctionnement du filtre IIR demandé à 3 fréquences pertinentes ($f < f_c$, $f = f_c$, $f > f_c$)	5	10		4	
	MPLAB: Démonstration d'un filtre IIR "limité en ressources" (Q2.5) qui cause de la saturation dans les calculs, avec visualisation dans DMCI des signaux d'entrée/sortie à différentes fréquences pertinentes				2	
	Qualité de la discussion dans le plan de test					6
		33	33	34	20	20
	TOTAUX		100		40	
Validation en laboratoire: résultats MPLAB + temps-réel		53	37.86%			
Plan vérif: résultats Python & MPLAB (résultats ET présentation)		67	47.86%			
Plan vérif: qualité de la discussion (structure, des tests unitaires, etc.)		20	14.29%			
		140	100.00%			

Modalités de l'évaluation de la solution à la problématique par équipe:

A- Plan de vérification et logiciels développés :

Votre plan de vérification (voir **Annexe F - Plan de vérification (développement du code C) pour les requis**) ainsi que vos projets MPLAB et PyCharm **doivent être remis avant la validation en laboratoire** dans une archive unique .zip par équipe nommée selon la convention "<CIP1 >_<CIP2>.zip" et contenant les fichiers suivants :

- Plan de vérification en format PDF.
- Archive .zip du répertoire complet de votre projet MPLAB. **ATTENTION, afin de réduire la taille de l'archive au minimum, déposez une version du projet sans code exécutable : dans MPLABX, click-droit sur le nom du projet + « Clean ».**
- Archive .zip de tous vos projets PyCharm

NB : Vous aurez ensuite jusqu'à 22h30, si vous le voulez, pour faire un 2^{ème} dépôt du plan de vérification pour finaliser votre document, par exemple pour compléter la description des graphiques, ajuster les axes, mettre des jolies couleurs, etc. Faites uniquement un dépôt du fichier PDF du plan de vérification (sans les projets MPLABX et PyCharm), nommé "<CIP1 >_<CIP2>_FINAL.pdf".

B- Validation en laboratoire : 20 minutes par équipes

Les éléments en gris dans le tableau précédant seront évalués à la période de validation en équipe. **ATTENTION, les graphiques évalués en validation doivent aussi apparaître dans votre plan de vérification.**

1) Préparation, avant le passage du tuteur pour la validation :

- a. Logiciels PyCharm et MPLABX ouverts.
- b. Les tampons *inBuffer1/inBuffer2* et *outBuffer1/outBuffer2* montrés dans DMCI.
- c. Les signaux BIN & BIN2 montrés avec l'AD2.
- d. Les graphiques demandés plus bas en PyCharm prêts à visualiser.

2) FIR:

- a. PyCharm : Montrer les graphiques des $|H|$ et $|H_{tot}|$ (NB : échelles verticales en dB).
- b. DMCI : Montrer le fonctionnement de 2 filtres (autres que H7) à des fréquences intéressantes.

3) IIR:

- a. PyCharm : Montrer sur une même figure les graphiques des $|H|$ pour les cas de coefficients bruts (double), Q2.13, et Q2.5.
- b. DMCI : Montrer le fonctionnement du filtre IIR avec coefficients Q2.13 et Q2.5 à des fréquences intéressantes.

4) Mesures temps-réel:

- a. Expliquer l'effet sur les signaux BIN1 et BIN2 du filtrage FIR, filtrage IIR, *passthrough*, et calcul de spectre. Ajoutez deux appels à `calc_power_spectrum()` dans le code FIR et expliquez le résultat.

NB :

- 1) Soyez prêts à montrer et expliquer votre code Python et code C sur demande.
- 2) Dans le cas du filtrage par FIR, l'ajout de *calc_power_spectrum()* risque de causer des problèmes avec la méthode *Overlap & Save* (glitches) pour cause de dépassement de l'intervalle de temps maximum permis pour l'exécution de la boucle *while* (par exemple en utilisant *sqrt()* dans la fonction), i.e. dans le cas du FIR, la fonction *calc_power_spectrum()* est à utiliser pour le déverminage seulement.

9 Mots clefs suggérés pour le schéma de concept au tutorat 2

Éléments suggérés du schéma de concept :

- Séquence logique qui relie l'équation aux différences avec la transformée en z et les usages des puissances négatives et positives.
- Séquence logique qui relie la conception d'un filtre IIR en partant d'une représentation algébrique analogique avec la représentation finale du filtre en forme Direct II.

Mots clefs à discuter :

- Équations à différence
- Filtres FIR et IIR
- Réponse impulsionnelle
- Convolution, définition et lien avec les équations à différences
- Fonction de transfert harmonique, calcul à partir de l'équation à différences
- Transformée en z , usage de la définition versus usage de la propriété de décalage
- Usage des puissances de z positives et négatives
- Pôles et zéros, conséquences du positionnement, calcul de la fonction de transfert
- Réponse en fréquence
- Transformée de Fourier à partir de la T_z
- Conditions pour la périodicité dans le domaine temporel ou fréquentiel
- Lien entre la convolution et la réponse impulsionnelle
- Différentes informations et usages des représentations temporelles (h) et fréquentielles (H) d'un filtre
- Calcul la réponse d'un filtre en régime permanent en connaissant sa fonction de transfert harmonique
- Lien entre la transformation de Fourier discrète (TFD) et la FFT
- *Twiddle factor*
- Notation Q , raison d'être, méthode de conversion
- Représentations Directe I et II, biquad, filtres en cascade
- Théorème de convolution
- FFT^{-1} par FFT
- Gauchissement des fréquences
- Transformation bilinéaire