I Capacité numérique

Simuler, à l'aide d'un langage de programmation, l'action d'un filtre sur un signal périodique dont le spectre est fourni. Mettre en évidence l'influence des caractéristiques du filtre sur l'opération de filtrage.

II Modules

Le module `signal` de `scipy` (documentation) donne accès entre autres :

- à des fonctions représentant des signaux usuels (créneau avec `square`, triangle et dents de scie avec `sawtooth`)
- à des fonctions définissant des fonctions de transfert de filtres par les coefficients de leurs fractions rationnelles (fonction `freqs`)

Le module `fft` de `scipy` (documentation) permet entre autres de réaliser les transformées de Fourier et transformées de Fourier inverse (en utilisant l'algorithme de Fast Fourier Transform) pour :

- à partir d'un signal f(t) numérisé, calculer une approximation numérique de son spectre de Fourier discret
- à partir d'un spectre de Fourier discret, calculer la valeur d'une fonction correspondante en un ensemble d'instants.

Dans ces deux cas, l'ensemble des instants où est évaluée la fonction et l'ensemble des fréquences où est évaluée sa transformée de Fourier sont tous les deux discrets.

Le module `ma` de `numpy` (documentation) offre ici la facilité de «masquer» certains éléments d'un tableau qu'on souhaite éliminer : ici les parties du spectre de poids trop faible.

```
%matplotlib inline
```

La ligne précédente ne doit apparaître que dans les notebooks `Jupyter`, pas dans un fichier python.

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

Out[86] :

III Étude d'un filtre du premier ordre

III.1 Définition

On définit un filtre passe-bas du premier ordre par son gain en bande passante et sa fréquence de coupure.

```
pi = np.pi
# Définition du filtre
f_c = 2e2                # fréquence de coupure (en Hz)
omega_c = 2.0*pi*f_c     # pulsation de coupure (en rad/s)
H_0 = 2                  # gain en bande passante
# Coefficients du dénominateur rangés par degrés décroissants
a = np.array( [1/omega_c,1.0] )
# Coefficients du numérateur
b = np.array( [H_0] )
H_coeffs = (b,a)
```

Out[87] :

III.2 Diagramme de Bode

On fait calculer le module et l'argument de la fonction de transfert pour un ensemble de fréquences déterminé, choisi en échelle logarithmique : ici 400 fréquences équiréparties en échelle log entre 10^1 et 10^5 Hz. La fonction `signal.freqs` renvoie :

- `w` la liste des pulsations utilisées
- `H` le tableau des valeurs (complexes) de la fonction de transfert pour ces pulsations.

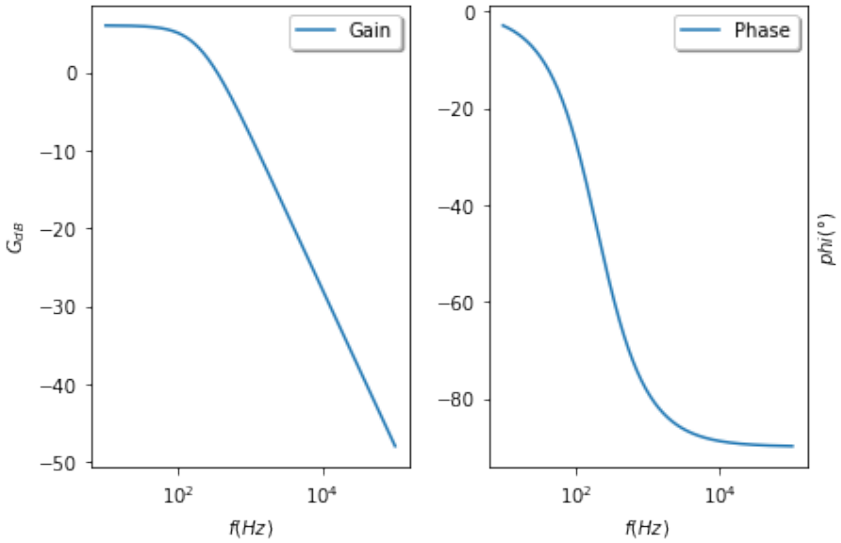
```
# Plage de fréquence à étudier
frequences = np.logspace(1 , 5 , 400)
[ w, H ] = signal.freqs(b,a,2*pi*frequences)
```

Out[88] :

On peut ensuite tracer le diagramme de Bode correspondant.

```
figBode1,(axBodeGain1,axBodePhase1) = plt.subplots(1,2) #pour avoir deux figures côte à
↪  côte
figBode1.tight_layout()
# gain en décibel, np.absolute calcule le module
G = 20.0 * np.log10(np.absolute(H))
# phase en degrés, np.angle calcule l'argument en radian, converti par rad2deg
phase = np.rad2deg(np.angle(H))
axBodeGain1.semilogx()
axBodePhase1.semilogx()
axBodeGain1.plot(frequences,G,label='Gain')
axBodeGain1.set_xlabel(r"$f(Hz)$")
axBodeGain1.set_ylabel(r"$G_{dB}$" )
axBodePhase1.plot(frequences,phase,label='Phase')
axBodePhase1.set_xlabel(r"$f(Hz)$")
axBodePhase1.set_ylabel(r"$phi(°)$")
axBodePhase1.yaxis.set_label_position("right")
axBodePhase1.legend(loc='best',shadow=True)
axBodeGain1.legend(loc='best',shadow=True)
figBode1.show()
```

Out[89] :



IV Simulation du filtrage

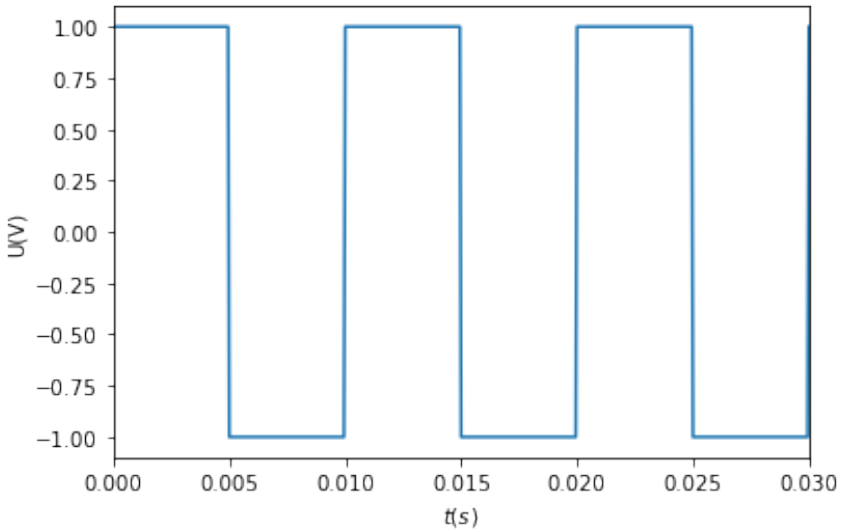
IV.1 Échantillonage de la fonction

On calcule les valeurs de la fonction sur laquelle sera appliquée le filtre sur un échantillon d'instants. La durée entre deux instants `Delta_t` doit être suffisamment courte pour que deux fonctions différentes n'aient pas le même échantillonage. On utilisera le «critère de Shannon» selon lequel `Delta_t = 1/(2 fmax)` avec `fmax` la fréquence maximale du spectre du signal.

On utilise ici une fonction créneau de fréquence `fCre`. Pour que son échantillonnage reproduise fidèlement ses harmoniques jusqu'au rang `9`, on doit donc échantillonner avec `Delta_t = 1/(18 fCre)`. On rajoute un facteur `10` pour augmenter la précision.

```
HarmoniqueMax = 9
fCre = 100 #Hz
Delta_t = 1/(20*HarmoniqueMax*fCre)
echantillon_t = np.arange(0,20/fCre,Delta_t) # pour disposer de 20 périodes pour
↪  l'échantillonage
EntreeCreneau = signal.square(echantillon_t*2*pi*fCre) # signal.square a pour période
↪  2 pi
figCreneau,axCreneau = plt.subplots()
axCreneau.set_xlim(0,3/fCre)
axCreneau.plot(echantillon_t,EntreeCreneau)
axCreneau.set_xlabel(r"$t(s)$")
axCreneau.set_ylabel(r"U(V)" )
figCreneau.show()
```

Out[90] :



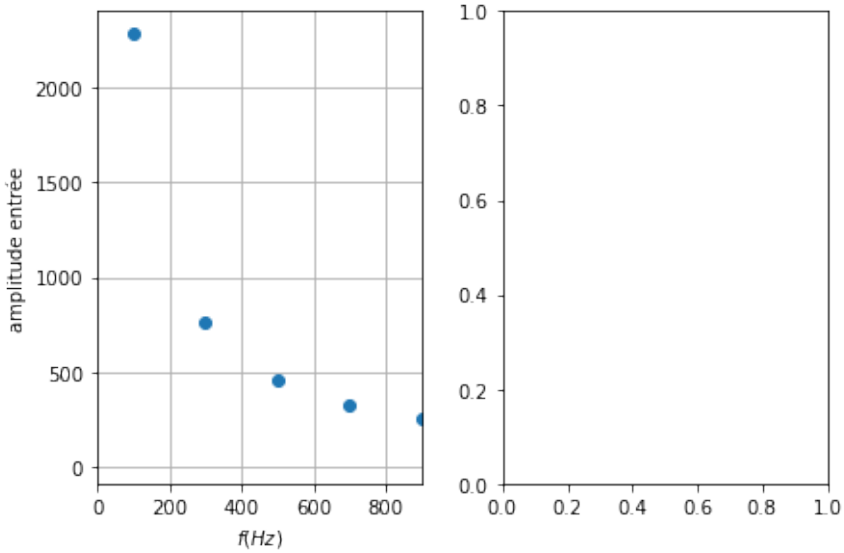
IV.2 Transformée de Fourier rapide

On calcule les fréquences `FrequencesFFTCreneau` adaptées à l'échantillonage avec `fftfreq`. On applique la transformée de Fourier rapide au signal échantillonné avec `fft` pour obtenir `FFTEntreeCreneau`, un tableau des amplitudes complexes des composantes de Fourier de `EntreeCreneau`. Le masque `mask` permet de ne garder à l'affichage que les composantes de Fourier dont l'amplitude est supérieure à un seuil.

On trace ensuite les modules de ces amplitudes complexes.

```
FrequencesFFTCreneau =  fftpack.fftfreq(len(EntreeCreneau) , Delta_t)
FFTEntreeCreneau = fftpack.fft(EntreeCreneau)
CutoffGdB=40
FFTEntreeCreneauGdB=20*np.log10(np.absolute(FFTEntreeCreneau))
mask=ma.masked_less(FFTEntreeCreneauGdB,np.max(FFTEntreeCreneauGdB)-CutoffGdB).mask
FrequencesFFTCreneauMasked=FrequencesFFTCreneau[~mask]
FFTEntreeCreneauMasked=FFTEntreeCreneau[~mask]
FFTEntreeCreneauGdBMasked=FFTEntreeCreneauGdB[~mask]
figFFTCreneau,(axFFTEntreeCreneau,axFFTSortieCreneau) = plt.subplots(1,2) #pour avoir
↪  deux figures côte à côte
figFFTCreneau.tight_layout()
axFFTEntreeCreneau.plot(FrequencesFFTCreneauMasked,
↪  np.absolute(FFTEntreeCreneauMasked), 'o', label='Entree')
axFFTEntreeCreneau.set_xlabel(r"$f(Hz)$")
axFFTEntreeCreneau.set_ylabel(r"amplitude entrée" )
axFFTEntreeCreneau.set_xlim([0, HarmoniqueMax*fCre])
axFFTEntreeCreneau.grid(which='both')
```

Out[91] :



On évalue ensuite les valeurs du gain complexe `HFFTCreneau` calculé pour ces fréquences puis on multiplie une à une les amplitudes de `FFTEntreeCreneau` par la valeur du gain `HFFTCreneau` pour obtenir la transformée de Fourier discrète du signal de sortie `FFTSortieCreneau`.

```
HFFTCreneau = signal.freqs(b,a,2*pi*FrequencesFFTCreneau)[1]
FFTSortieCreneau = HFFTCreneau * FFTEntreeCreneau
FFTSortieCreneauMasked = FFTSortieCreneau[~mask]
axFFTSortieCreneau.plot(FrequencesFFTCreneauMasked,
↪  np.absolute(FFTSortieCreneauMasked), '+', label='Sortie')
axFFTSortieCreneau.set_xlabel(r"$f(Hz)$")
axFFTSortieCreneau.set_ylabel(r"amplitude sortie" )
axFFTSortieCreneau.set_xlim([0, HarmoniqueMax*fCre])
axFFTSortieCreneau.grid(which='both')
axFFTSortieCreneau.yaxis.set_label_position("right")
axFFTSortieCreneau.legend(loc='best',shadow=True)
axFFTEntreeCreneau.legend(loc='best',shadow=True)
figFFTCreneau.show()
```

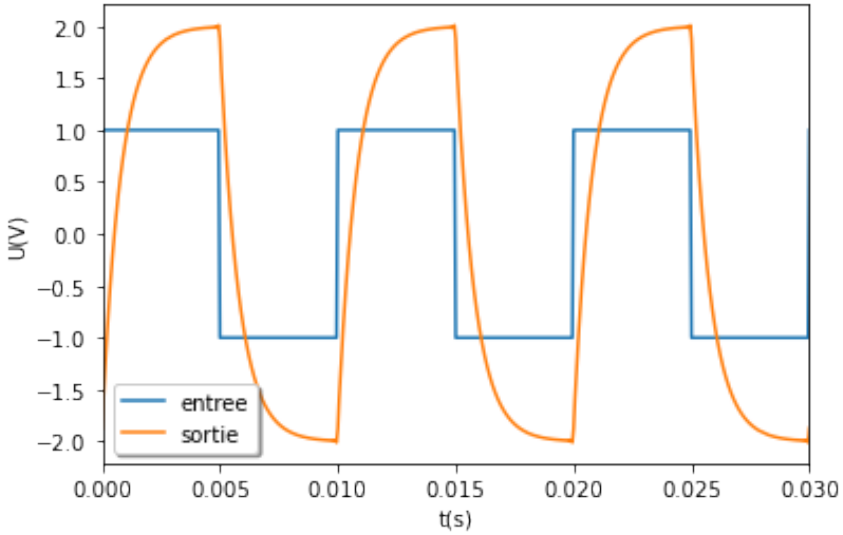
Out[92] :

Il suffit ensuite d'utiliser la transformée de Fourier Inverse `ifft` pour retrouver la fonction en sortie de filtre, qu'on superpose à la fonction en entrée pour les comparer.

L'algorithme d'aller-et-retour entre `fft` et `ifft` peut faire apparaître des parties imaginaires dans le signal de sortie. On vérifie qu'elles sont négligeables avec `max(np.imag)` et on ne trace que la partie réelle

```
SortieCreneau = fftpack.ifft(FFTSortieCreneau)
figCreneauES,axCreneauES = plt.subplots()
axCreneauES.plot(echantillon_t,EntreeCreneau,label='entree')
print(f'max des parties imaginaires {max(np.imag(SortieCreneau))}')
axCreneauES.plot(echantillon_t, np.real(SortieCreneau), label='sortie')
axCreneauES.set_xlim([0, 3/fCre])
axCreneauES.set_xlabel(r't(s)')
axCreneauES.set_ylabel(r'U(V)')
axCreneauES.legend(loc='best',shadow=True)
figCreneauES.show()
```

Out[93] :



V Questions du DM05

V.1 II4a

V.2 II4c

V.3 III4b