Devoirs numériques - Corrigé

C. Lacpatia

Aug 03, 2023

CONTENTS

Voici les corrigés des devoirs sur les capacités numériques

CHAPTER ONE

RÉPONSE À UNE RAMPE DE TENSION

1.1 Position du problème.

On considère un condensateur de capacité C reliée en série à une résistance R. L'ensemble est branché en série à une source idéale de tension $E(t)$ délivrant une rampe de tension:

$$E(t) = \begin{cases} 0, & \text{if } x < 0 \\ E_0 \frac{t}{t_0}, & \text{if } 0 \leq x \leq t_0 \\ E_0, & \text{if } x > t_0 \end{cases}$$

A $t = 0$, le condensateur est complètement déchargé.

1.2 Etude analytique

1.2.1 Généralité

1. La loi des mailles $u_C(t) + Ri(t) = E(t) \Longrightarrow u_C(t) + RC\frac{\mathrm{du_C}}{\mathrm{dt}} = E(t)$
2. $u_C = E_0$ donc $\Delta E = \frac{1}{2}CE_0^2$
3. $u(t) = E_0(1 - \exp(-t/\tau))$ et $i(t) = \frac{E_0}{R}\exp(-t/\tau)$. Il vient:

$$E_g = \int_{t=0}^{t=+\infty} \frac{E_0^2}{R}\exp(-t/\tau)dt = CE_0^2 \Longrightarrow \eta_{creneau} = \frac{1}{2}$$

1.3 Etude numérique

```
from matplotlib.pyplot import *
from numpy import *
```

1.3.1 Fonctions utiles

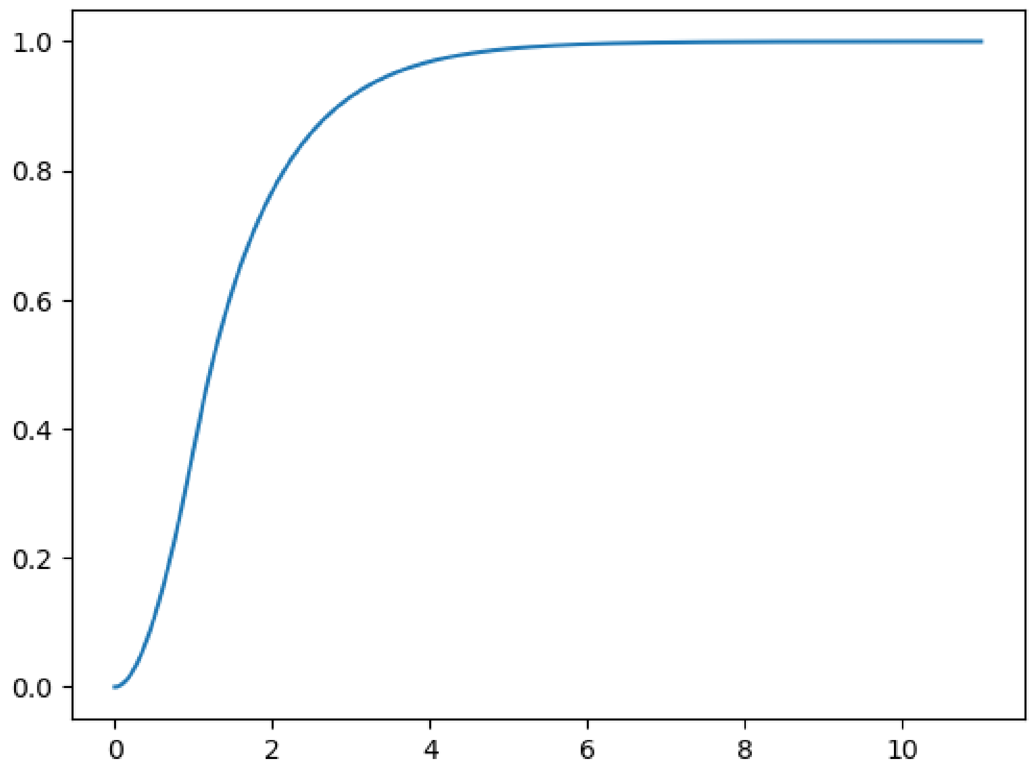
```
def deriv(xk, yk):
  n = len(xk)
  L = [(yk[1] - yk[0])/(xk[1] - xk[0])]  # Dérivée à gauche
  for k in range(1, n - 1):
    L.append((yk[k+1] - yk[k-1])/(xk[k+1] - xk[k-1]))  # Dérivée au centre
  L.append((yk[n-1] - yk[n-2])/(xk[n-1] - xk[n-2]))  # Dérivée à droite
  return array(L)

def integ(xk, yk):
  L = [0]  # Premier terme (integrale nulle)
  n = len(xk)
  for k in range(0, n-1):
    L.append(L[-1] + (yk[k] + yk[k+1]) / 2 * (xk[k+1] - xk[k]))  # Intégration par␣
↪méthode des trapèzes
  return array(L)

def euler(f, y0, t0, tf, pas):
  tk = [t0]
  yk = [y0]
  while tk[-1] < tf:
    yk.append(yk[-1] + pas * f(tk[-1], yk[-1]))
    tk.append(tk[-1] + pas)
  return array(tk), array(yk)
```

1.3.2 Réponse à une rampe particulière

1. $\frac{\mathrm{du}}{\mathrm{dt}} = E_0 \frac{\mathrm{dz}}{\mathrm{dx}} \frac{\mathrm{dx}}{\mathrm{dt}} = \frac{E_0}{\tau} \frac{\mathrm{dz}}{\mathrm{dx}}$ soit l'équation demandée en utilisant l'équation différentielle trouvée précédemment.

```
def f(x, y):
  if x < 1:
    return - z * y + z * x
  else:
    return - z * y + z

def rampe(x):
  if x < 1:
    return x
  else:
    return 1

v0 = 0
z = 1
tf = max(2, 1 + 10 / z)
pas = 1e-3
tk, vk = euler(f, v0, 0, tf, pas)
plot(tk, vk)
```

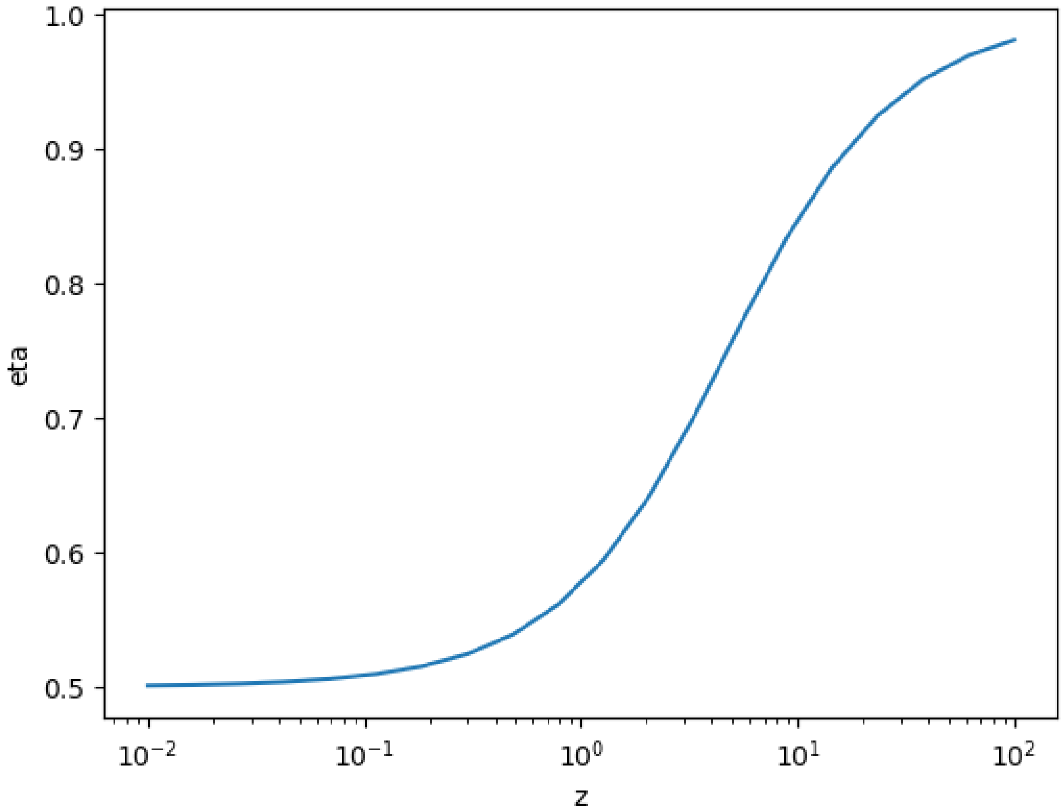
```
[<matplotlib.lines.Line2D at 0x24837377cd0>]
```



1.3.3 Etude énergétique

```
zs = logspace(-2, 2, 20)

def eta(z):
  pas = 1e-3
  v0 = 0
  tf = max(2, 1 + 10 / z)
  tk, vk = euler(f, v0, 0, tf, pas)
  rampek = array([rampe(t) for t in tk])
  ik = deriv(tk, vk)
  Ek = integ(tk, ik * rampek)
  return 1 / (2 * Ek[-1])

etas = []
for z in zs:
  etas.append(eta(z))

f, ax = subplots()
ax.set_xscale('log')
ax.set_xlabel('z')
ax.set_ylabel('eta')
ax.plot(zs, etas)
```

```
[<matplotlib.lines.Line2D at 0x248374075e0>]
```


1.0
0.9
0.8
0.7
0.6
0.5
eta
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
z

CHAPTER

TWO

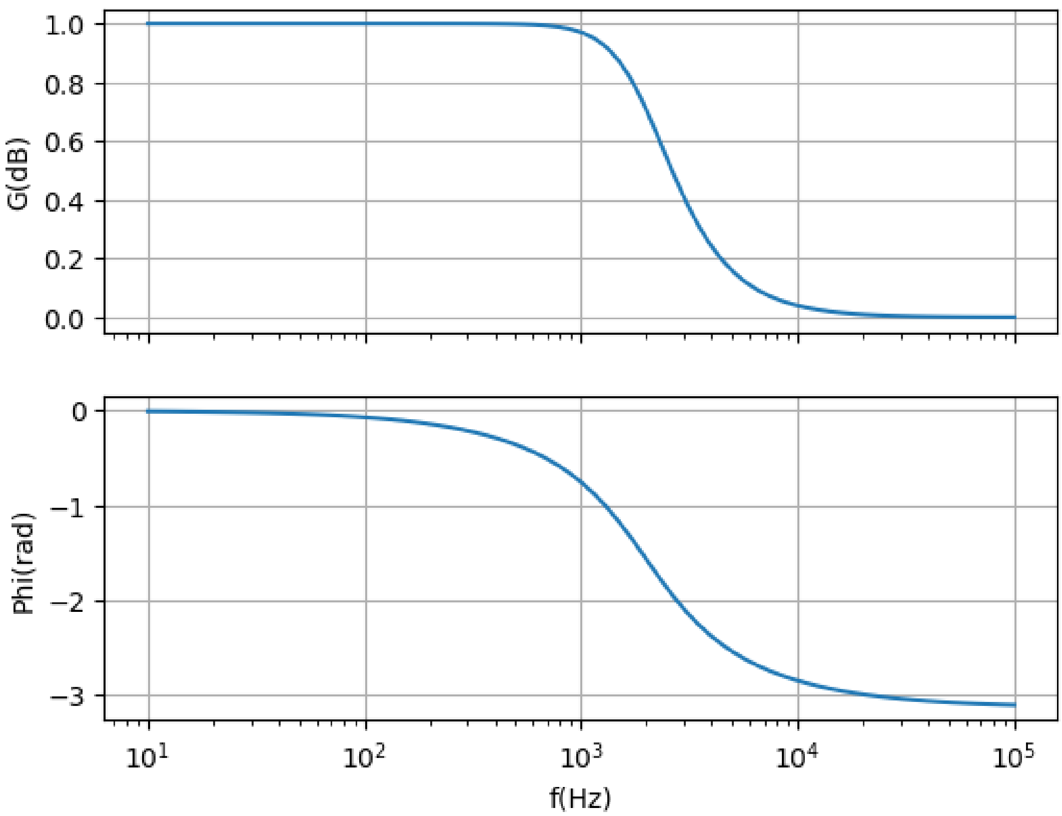
ETUDE D'UN FILTRAGE LINÉAIRE

2.1 Etude générale

1. Il s'agit d'un filtre passe-bas

```
from numpy import *
import matplotlib.pyplot as plt
```

```
def butterworth(f:float, fc:float, n:int) -> complex:
    """Filtre de butterworth - Fonction de transfert complexe"""
    H = 1
    for k in range(n):
        pk = exp((2 * k + 1 + n)* pi * 1j / (2*n))
        H = H * 1 / (1j * f / fc - pk)
    return H

fc = 2e3  # Fréquence de coupure
n = 2 # Ordre
f, ax = plt.subplots(2, 1, sharex='col')
ax[0].set_ylabel('G(dB)')
ax[1].set_ylabel('Phi(rad)')
ax[1].set_xlabel('f(Hz)')
fs = logspace(1, 5, 100)
Hs = butterworth(fs, fc, n)
ax[0].semilogx(fs, abs(Hs))
ax[1].semilogx(fs, angle(Hs))
ax[0].grid()
ax[1].grid()
plt.show()
```



2.1.1 Choix de n

1. Il vient $\eta = \left(\frac{f_1}{f_2}\right)^{2n}$
2. La première expression impose $\eta < \frac{0.9^{-2}-1}{0.1^{-2}-1} = 2.4 \times 10^{-3}$
3. Il vient $n > -\frac{1}{2}\ln \eta_C \ln(f_2/f_1) = 5.9$

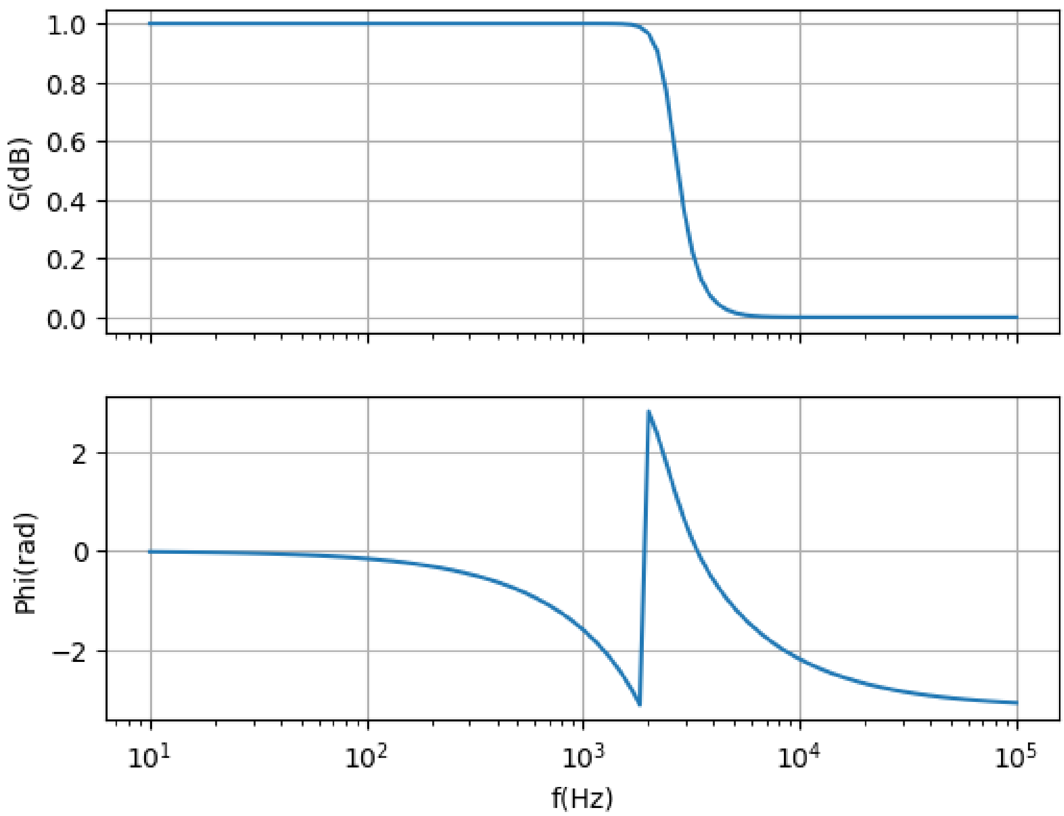
```
f1 = 1500
f2 = 2500
fc = 1000


def eta(n:float) -> float:
    """Calcul de eta"""
    return (abs(butterworth(f1, fc, n)) ** -2 - 1 ) / (abs(butterworth(f2, fc, n)) **
 ↪-2 - 1 )

etac = (0.9 ** -2 - 1) / (0.1**-2 - 1)  # Valeur critique de eta

n = 1
while eta(n) > etac:
    n =n + 1
print(n)
```

```
6
```

2.1.2 Choix de f_c

```python
prec = 0.1
fca = f1
fcb = f2

n = 6

def test_b(fc:float):
    # Fonction dont on cherche la racine par dichotomie
    return abs(butterworth(f1, fc, n) - 0.9)

while abs(fcb - fca) > prec:  # Méthode de dichotomie
    if test_b(fca) == 0:
        fcb = fca
    elif test_b(fcb) == 0:
        fca = fcb
    else:
        fcc = (fcb + fca ) / 2
        if test_b(fcc) == 0:
            fca = fcc
            fcb = fcc
        elif test_b(fcc) * test_b(fca) < 0:
            fcb = fcc
        else:
            fca = fcc

fcc = (fca + fcb) / 2
print(fcc, abs(butterworth(f1, fcc, n)))

f, ax = plt.subplots(2, 1, sharex='col')
ax[0].set_ylabel('G(dB)')
ax[1].set_ylabel('Phi(rad)')
ax[1].set_xlabel('f(Hz)')
fs = logspace(1, 5, 100)
Hs = butterworth(fs, fcc, n)
ax[0].semilogx(fs, abs(Hs))
ax[1].semilogx(fs, angle(Hs))
ax[0].grid()
ax[1].grid()
plt.show()
```

```
2499.969482421875 0.998913223582161
```



2.2 Réponse du filtre

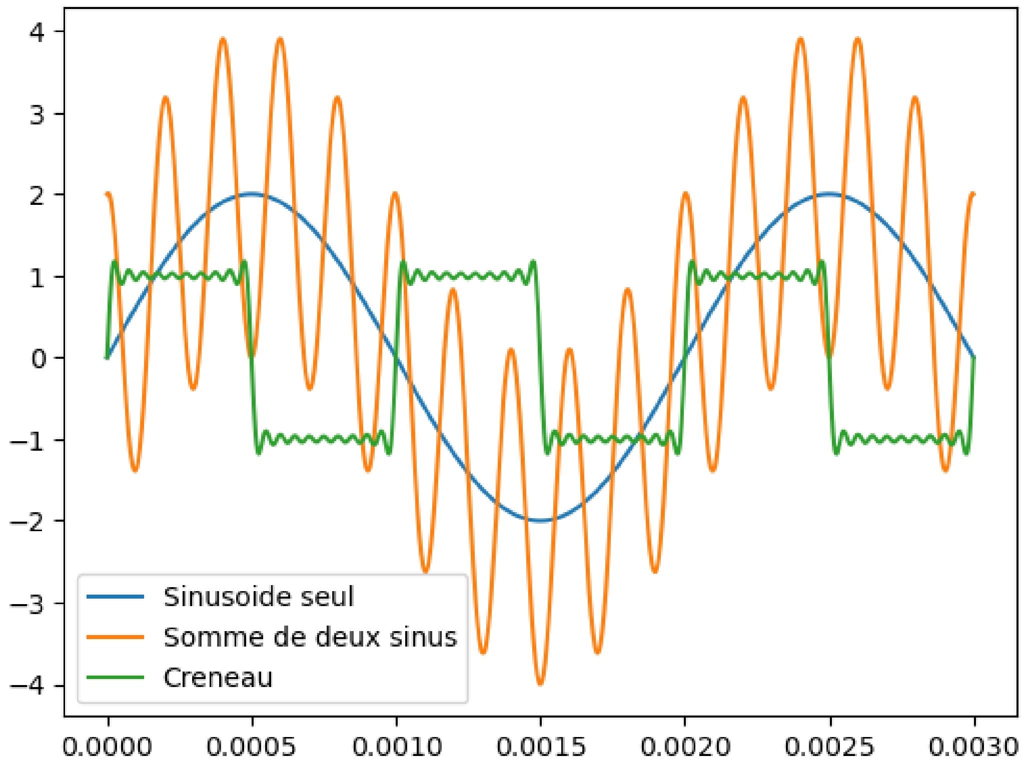
```
sig1 = [[500, 2, 0]]  # Représentation du premier signal
sig2 = [[500, 2, 0], [5000, 2, pi/2]]  # Représentation du second signal

def creneau(N:int) -> list[list[float]]:
    L = []
    f0 = 1000
    for k in range(N):
        L.append([(2 * k + 1) * 1000, 4 / (pi * (2 * k + 1)), 0])
    return L


sig3 = creneau(10)  # 10 premières composantes du créneau

def freq2temp(repf:list[list[float]], t:ndarray) -> ndarray:
    """Fonction qui renvoie la représentation temporelle
    à partir de la représentation fréquentielle."""
    u = zeros(len(t))
    for sinus in repf:
        u += sinus[1] * sin(2 * pi * sinus[0] * t + sinus[2])
    return u

t = linspace(0, 3e-3, 1000)
```

(continues on next page)

(continued from previous page)

```
sigt1 = freq2temp(sig1, t)
sigt2 = freq2temp(sig2, t)
sigt3 = freq2temp(sig3, t)

plt.plot(t, sigt1, label="Sinusoide seul")
plt.plot(t, sigt2, label="Somme de deux sinus")
plt.plot(t, sigt3, label="Creneau")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x23ae79fe1c0>
```



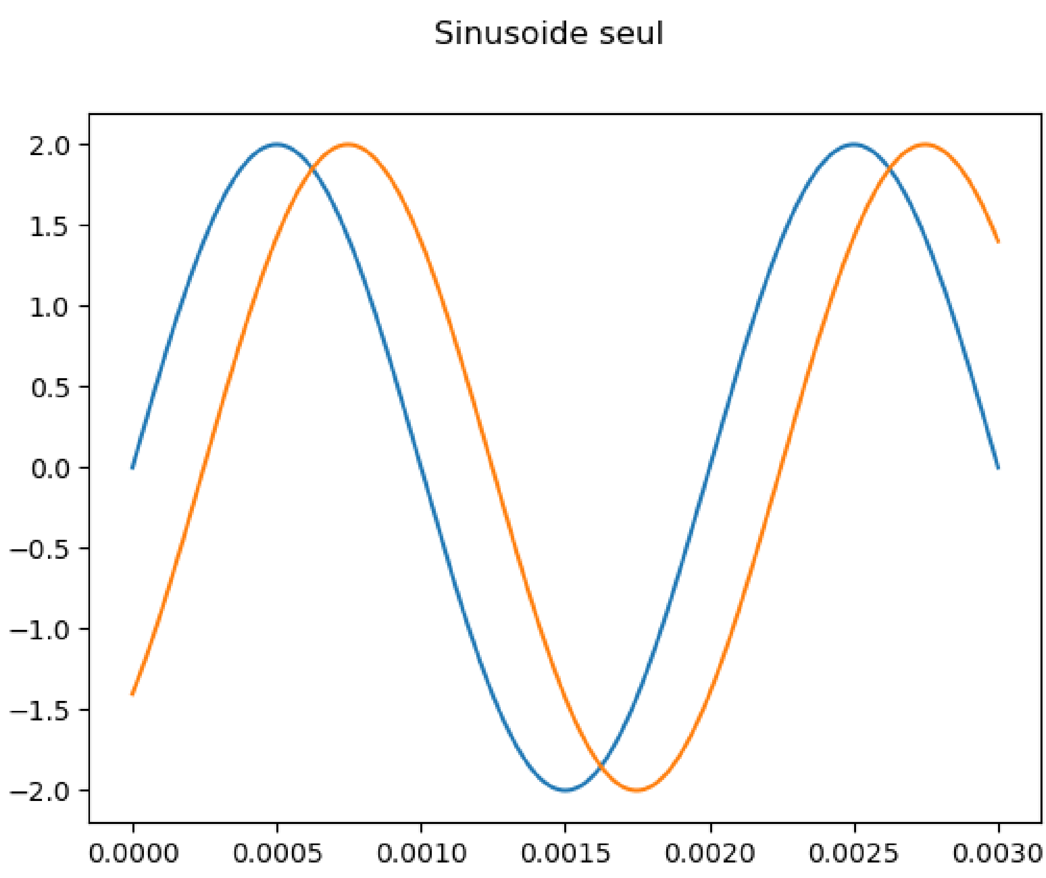

```
fc = fcc
n = 6

def reponse(filtre:callable, repf:list[list[float]]) -> list[list[float]]:
    """Calcul de la réponse pour une représentation fréquentielle donnée"""
    s = []
    for sinus in repf:
        sinussf = filtre(sinus[0], fc, n)  # Calcul de l'amplitude complexe pour␣
↪frequence
        """Le module et l'argument donne les caractéristiques de la composante pour␣
↪la sortie"""
        s.append([sinus[0], abs(sinussf) * sinus[1], angle(sinussf) + sinus[2]])
    return s

sigs1 = freq2temp(reponse(butterworth, sig1), t)
sigs2 = freq2temp(reponse(butterworth, sig2), t)
```

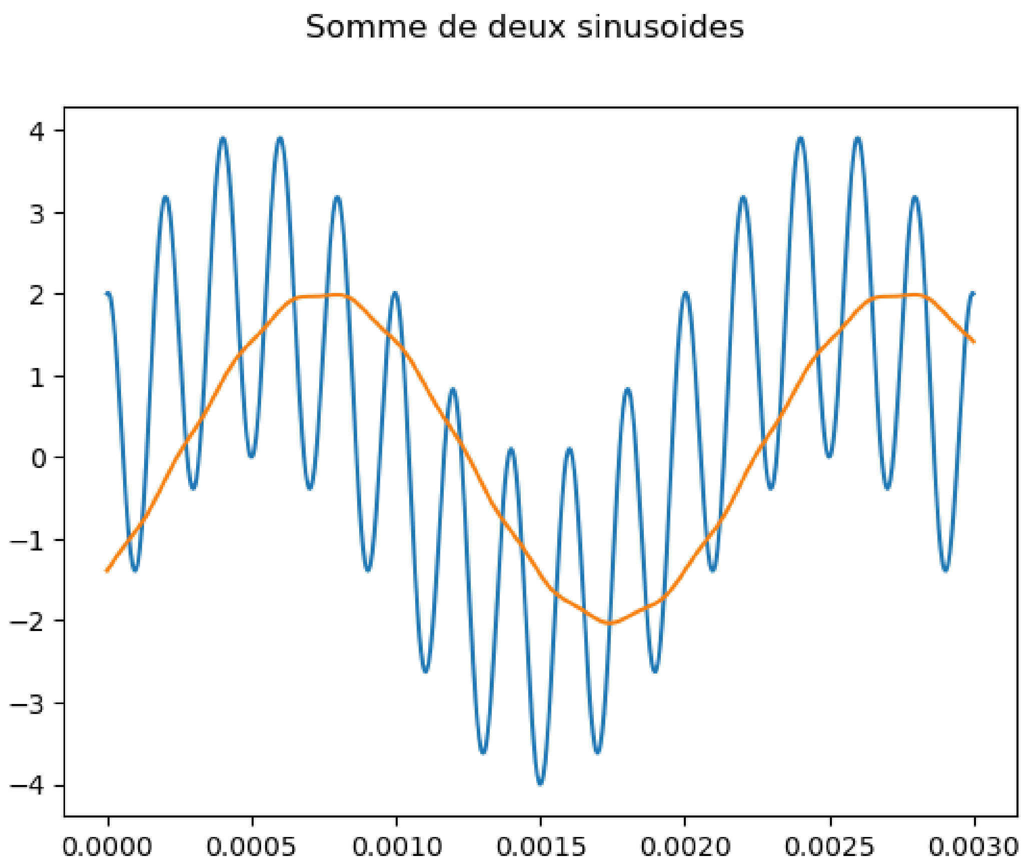
(continues on next page)

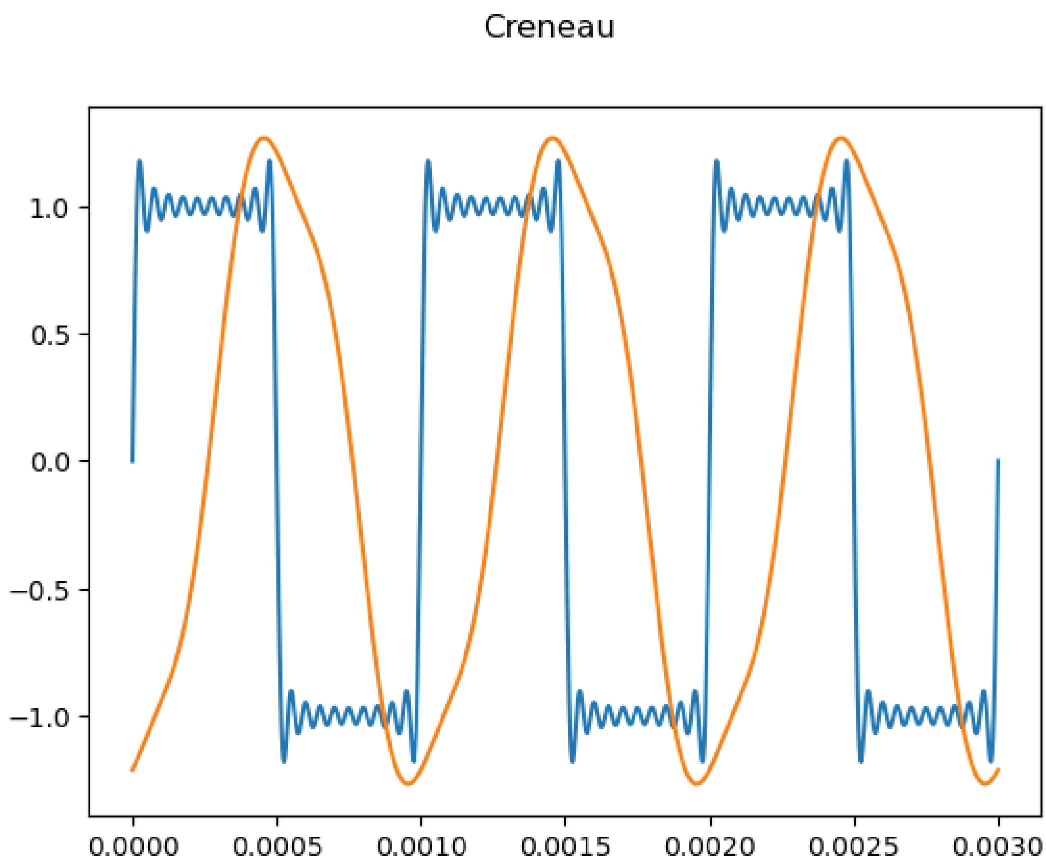
(continued from previous page)

```
sigs3 = freq2temp(reponse(butterworth, sig3), t)

f1, ax1 = plt.subplots()
f1.suptitle("Sinusoide seul")
ax1.plot(t, sigt1)
ax1.plot(t, sigs1)

f2, ax2 = plt.subplots()
f2.suptitle("Somme de deux sinusoides")
ax2.plot(t, sigt2)
ax2.plot(t, sigs2)

f3, ax3 = plt.subplots()
f3.suptitle("Creneau")
ax3.plot(t, sigt3)
ax3.plot(t, sigs3)
```

```
[<matplotlib.lines.Line2D at 0x23ae9b7b820>]
```




Somme de deux sinusoides
4
3
2
1
0
−1
−2
−3
−4
0.0000
0.0005
0.0010
0.0015
0.0020
0.0025
0.0030



On remarque que si le but est d'isoler la fréquence la plus basse, cela ne marche pas complètement car elles sont très proche. Il faudrait un filtre encore plus sélectif ou plus sûrement utiliser un filtre passe-bande accordé à la fréquence du signal.