Devoirs numériques - Corrigé

C. Lacpatia

Jan 18, 2024

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

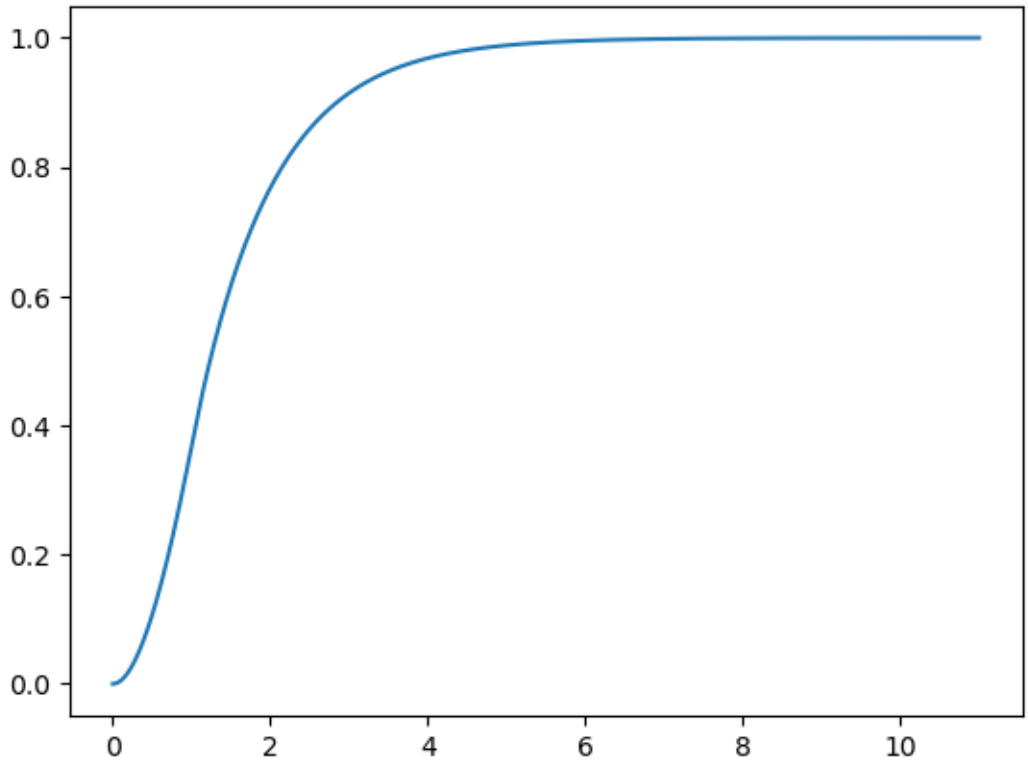
```
def deriv(xk, yk):
  n = len(xk)
  L = [(yk[1] - yk[0])/(xk[1] - xk[0])]  # Dérivée à gauche
  for k in range(1, n - 1):
    L.append((yk[k+1] - yk[k-1])/(xk[k+1] - xk[k-1]))  # Dérivée au centre
  L.append((yk[n-1] - yk[n-2])/(xk[n-1] - xk[n-2]))  # Dérivée à droite
  return array(L)

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)
```

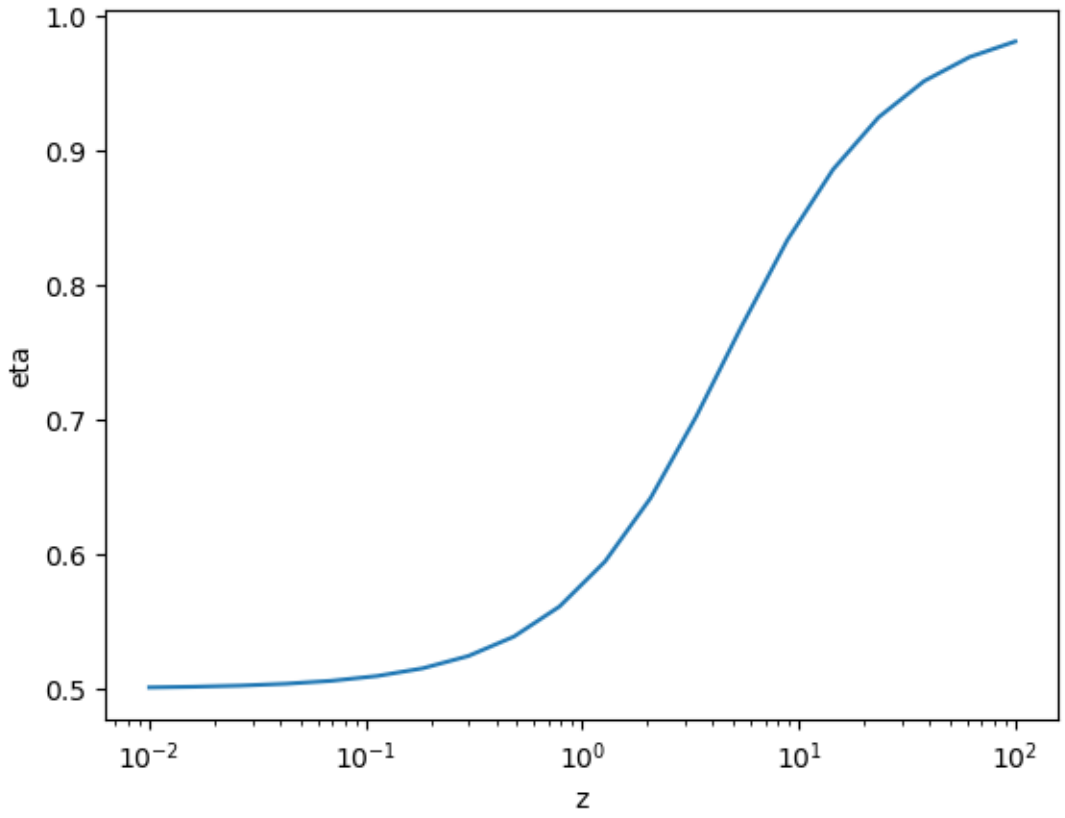
```
[<matplotlib.lines.Line2D at 0x1c42c7d49d0>]
```



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 0x1c42c844750>]
```


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

```
"""Importation des bibliothèques"""
import numpy as np
import matplotlib.pyplot as plt
```

2.1 Etude générale

1. C'est un filtre **passe-bas** qui vaut 1 à basse fréquence et 0 à haute fréquence.
2.

```
def butterworth(f:float, fc:float, n:int) -> complex:
    P = 1  # Initialisation du produit
    for k in range(1, n+1):  # k va de 1 à n
      P = P * 1/(1j * f / fc - np.exp(1j * np.pi / (2 * n) * (2*k+n-1)))
    return P
```

3.

```
fc = 2000 # Fréquence de coupure
n = 2

fs = np.logspace(1, 5, 1000)  # Fréquences de calcul
Hs = butterworth(fs, fc, n)  # Valeur de H aux fréquences précédentes
Gs = np.abs(Hs)  # Gain réel
Gdbs = 20 * np.log10(Gs)  # Gain en décibel
phis = np.angle(Hs)  # Phase

fgain, axg = plt.subplots()  # Tracé du diagramme de Bode en gain
axg.set_xlabel('f(Hz)')
axg.set_ylabel('Gdb(dB)')
axg.semilogx(fs, Gdbs)
axg.grid()  # Optionnel :  tracé de la grille

fphase, axph = plt.subplots()  # Tracé du diagramme de Bode en phase
axph.set_xlabel('f(Hz)')
axph.set_ylabel('Phase(rad)')
axph.semilogx(fs, phis)
axph.grid()  # Optionnel :  tracé de la grille

plt.show()
```

4. On observe que les valeurs de gain en décibel autour de 1.5kHz ($G_{dB} \approx -2dB$) et 2.5kHz ($G_{dB} \approx -10dB$) ne sont pas acceptables (valeurs attendues $-0.9dB$ et $-20dB$). Ce filtre n'est donc pas acceptable

2.1.1 Choix de n

On définit

$$\eta = \frac{G(f_1)^{-2} - 1}{G(f_2)^{-2} - 1}$$

1.

$$\begin{aligned}\eta &= \frac{\left(\frac{1}{\sqrt{1+x_1^{2n}}}\right)^{-2} - 1}{\left(\frac{1}{\sqrt{1+x_2^{2n}}}\right)^{-2} - 1} \\ &= \frac{x_1^{2n}}{x_2^{2n}} \\ &= \left(\frac{f_1}{f_2}\right)^{2n}\end{aligned}$$

2. De $G(f_1) > 0.9 = G_{1m}$ et $G(f_2) < 0.1 = G_{2m}$, il vient

$$\begin{aligned}\eta &= \frac{G(f_1)^{-2} - 1}{G(f_2)^{-2} - 1} \\ &< \frac{G_{1m}^{-2} - 1}{G_{2m}^{-2} - 1} = 2.3 \times 10^{-3}\end{aligned}$$

3.

$$\begin{aligned}\eta &< \eta_C \\ \left(\frac{f_1}{f_2}\right)^{2n} &< \eta_C \\ n &> \frac{1}{2}\frac{\ln \eta_C}{\ln \frac{f_1}{f_2}} = 5.9\end{aligned}$$

4.

```
f1 = 1500
f2 = 2500

def eta(n:int)->float:
    fc = 1000
    return (np.abs(butterworth(f1, fc, n)) ** (-2) - 1) / (np.abs(butterworth(f2, fc,␣
 ↪n)) ** (-2) - 1)

etaC = (0.9**-2 - 1) / (0.1**-2 - 1)  # Valeur limite de eta
print(etaC)
```

(continues on next page)

(continued from previous page)

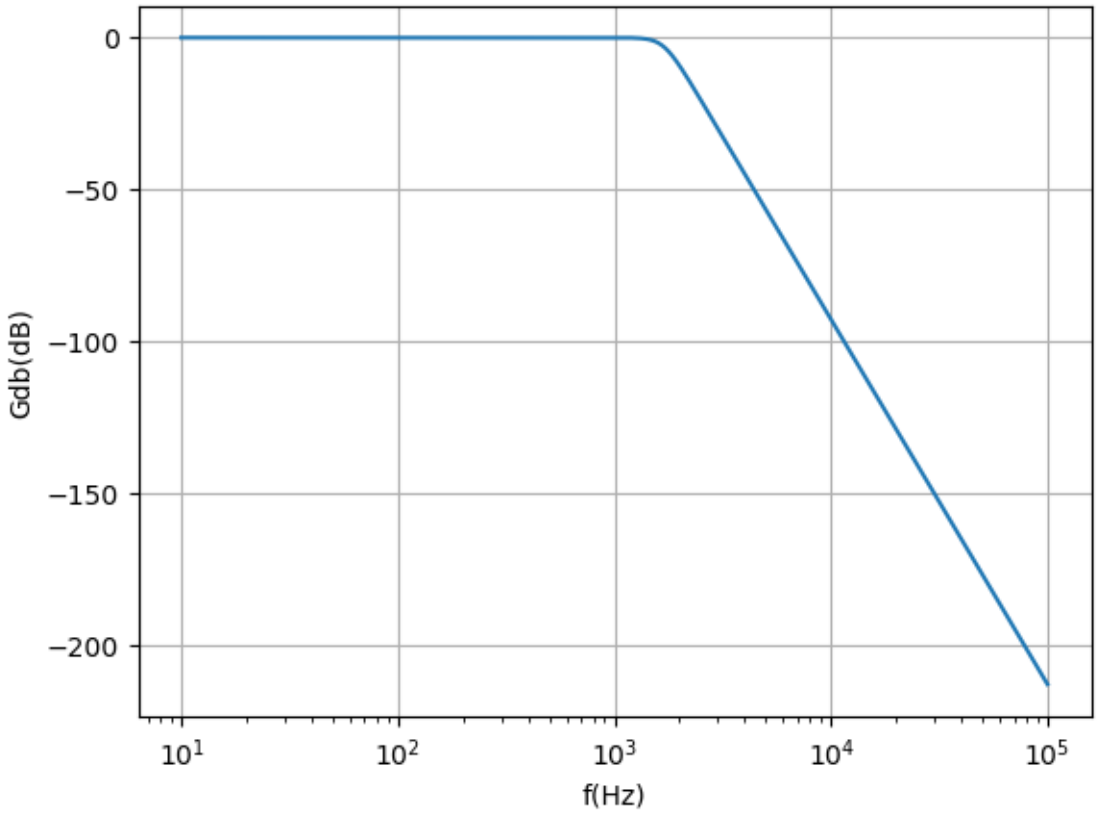
```
n = 1 # On part d'un ordre 1
while eta(n) > etaC: # Test pour déterminer l'ordre minimale
    n = n + 1
print(n)
print(np.log(etaC) / np.log(f1 / f2) / 2)  # Pour comparer à la valeur analytique
```

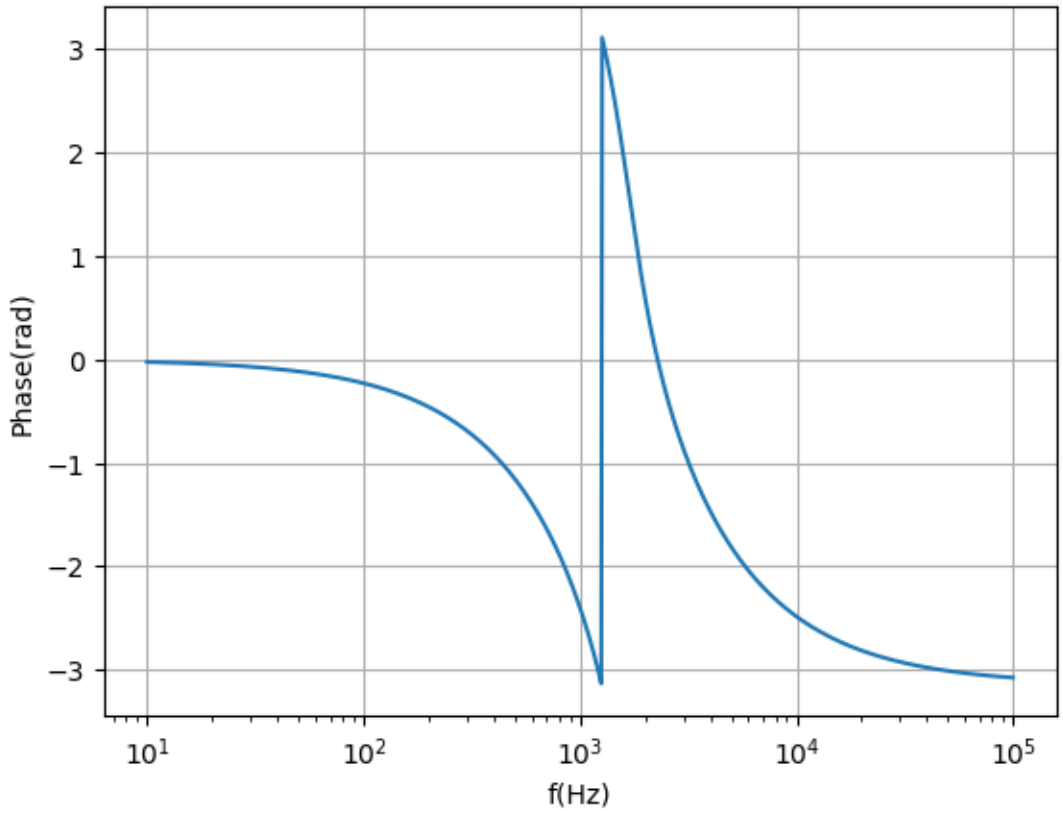
2.1.2 Choix de f_c

1.

```
n = 6
prec = 0.1  # Précision demandée
fmin = f1  # On démarre avec l'intervalle [f1;f2]
fmax = f2
while (fmax - fmin) > prec:
    fc = (fmax + fmin) / 2
    """On calcule G(f1) pour trois fréquences de coupure fmin, fmax, fc"""
    Ga = np.abs(butterworth(f1, fmin, n)) - 0.9
    Gb = np.abs(butterworth(f1, fmax, n)) - 0.9
    Gc = np.abs(butterworth(f1, fc, n)) - 0.9
    if Gc == 0:  # Le gain vaut 0.9 en fc
        fmax, fmin = fc, fc
    elif Ga * Gc < 0:  # Le gain vaut 0.9 entre fmin et fc
        fmax = fc
    else:  # Le gain vaut 0.9 entre fc et fmax
        fmin = fc
fc = (fmax + fmin) / 2  # On choisit le milieu du dernier intervalle.
print(fc)
print(np.abs(butterworth(f1, fc, n)))  # Pour vérifier que G(g1) >= 0.9
print(np.abs(butterworth(f2, fc, n)))  # Pour vérifier que G(g1) <= 0.1
```

A titre d'information, voici les diagrammes de Bode qu'on obtient:


0
−50
−100
−150
−200
Gdb(dB)
10^1
10^2
10^3
10^4
10^5
f(Hz)


3
2
1
0
−1
−2
−3
Phase(rad)
10^1
10^2
10^3
10^4
10^5
f(Hz)

2.2 Réponse du filtre

2.2.1 Représentation fréquentielle d'un signal

1.

```
sinus1 = [[500, 2, 0]]  # Sinus seul
somme1 = [[500, 2, 0], [5000, 2, np.pi/2]]  # Somme de deux sinus
```

2.

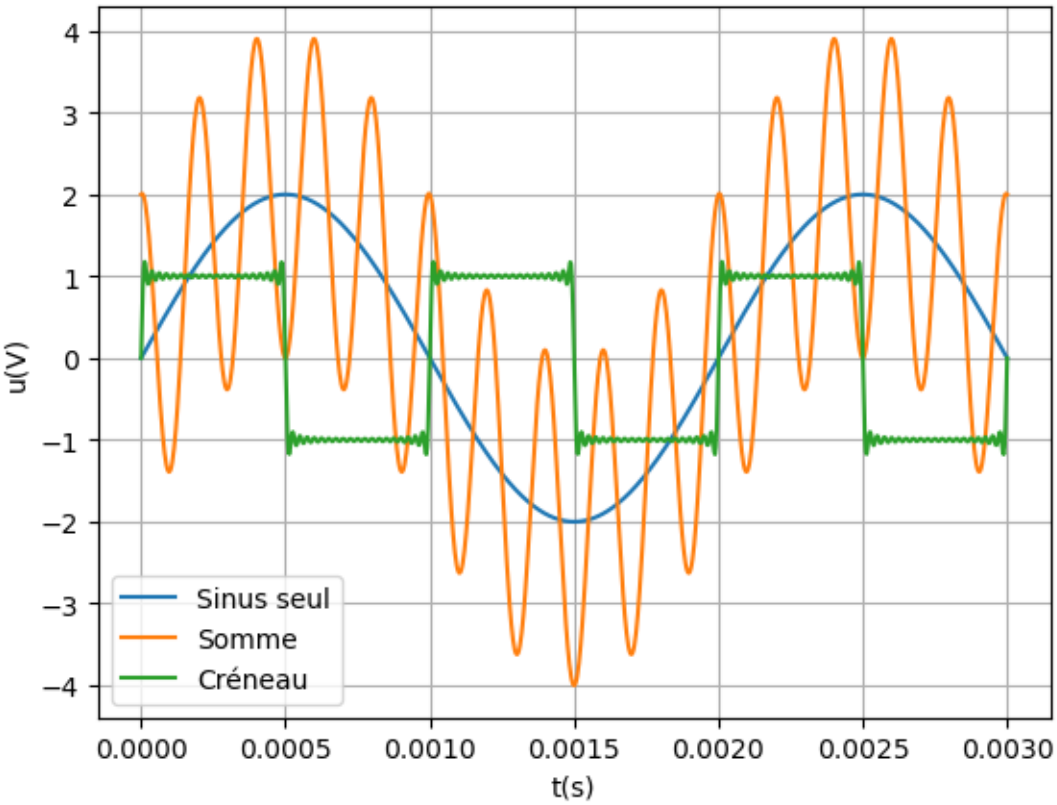
```
def creneau(N:int) -> list[list[float]]:
    spec = []  # Initialisation du spectre
    f = 1000
    for k in range(0,N):
        spec.append([(2 * k +1) * f, 4 / (np.pi * (2 * k +1)), 0])  # Ajout d'une␣
↪composante
    return spec
```

3.

```
def freq2temp(repf:list[list[float]], t:np.ndarray) -> np.ndarray:
    u = np.zeros(len(t))  # Initialisation des valeurs de u pour la somme
    for composante in repf:  # On parcourt chaque composante de u
        u = u + composante[1] * np.sin(composante[0] * 2 * np.pi * t + composante[2])␣
↪ # Ajout d'une composante - On utilise la vectorialisation
    return u
```

4.

```
t = np.linspace(0, 3e-3, 1000)
y1 = freq2temp(sinus1, t)
y2 = freq2temp(somme1, t)
y3 = freq2temp(creneau(20), t)

f, ax = plt.subplots()
ax.set_xlabel('t(s)')
ax.set_ylabel('u(V)')
ax.plot(t, y1, label="Sinus seul")
ax.plot(t, y2, label="Somme")
ax.plot(t, y3, label="Créneau")
ax.grid()
ax.legend()  # Affichage de la légende
plt.show()
```



2.2.2 Réponse du filtre.

1. Le conditionnement du filtre permet de considérer que les composante dans la bande passante ne sont pas modifiées alors que les composantes hors de la bande passante sont supprimées. Ainsi:

- Pour le sinus seul de fréquence 500Hz, on attend un signal de sortie identique à l'entrée **mais déphasé comme le montre le diagramme de Bode en phase** ($\phi \approx 1.5$rad **soit une quadrature de phase).**
- Pour la somme de sinusoïdes, la composante à 5kHz devraient être coupées et on trouve donc un signal identique au cas précédent.
- Pour le créneau, seul le fondamental à 1kHz est dans la bande passante, les autres harmoniques (la première est à 3kHz) sont coupées. On attend donc un sinusoïde quasi pur de fréquence 1kHz et d'amplitude $4A/\pi$. **Il sera encore plus déphasé car la phase du filtre à 1kHz est de l'ordre de** -2.4rad**.**

2.

```
def reponse(filtre: callable, repf:list[list[float]]) -> list[list[float]]:
    reps = []  # Initialisation du spectre en sortie
    for composante in repf:
        h = filtre(composante[0], fc, n)  # Calcul de H pour la composante étudiée
        amplitude = composante[1] * np.abs(h)  # Calcul de l'amplitude de la␣
↪composante en sortie
        phase = composante[2] + np.angle(h)  # Calcul de la phase à l'origine de la␣
↪composante en sortie
        reps.append([composante[0], amplitude, phase])
    return reps
```

3. Pour un signal dont la représentation fréquentielle est stockée dans la variable `sig1`, écrire une série d'instructions qui trace sur un même graphique le signal d'entrée et la réponse au filtre de Butterworth précédemment dimensionné (1000 points entre 0ms et 3ms).

```
sig1 = sinus1  # Ligne inutile à l'écrit mais qui évite à Python de planter car sig1 n
↪'est pas définie.

reps = reponse(butterworth, sig1)
t = np.linspace(0, 3e-3, 1000)
entree_temp = freq2temp(sig1, t)  # Passage au temporel pour l'entrée
sortie_temp = freq2temp(reps, t)  # Passage au temporel pour l'entrée

f, ax = plt.subplots()
ax.set_xlabel('t(s)')
ax.set_ylabel('u(V)')
ax.plot(t, entree_temp, label="Entrée")
ax.plot(t, sortie_temp, label="Sortie")
ax.grid()  # Facultatif
ax.legend()
plt.show()
```

4. [Ordinateur] Obtenir la réponse à chacun des trois signaux précédant et comparer aux prévisions approchées faites précédemment.

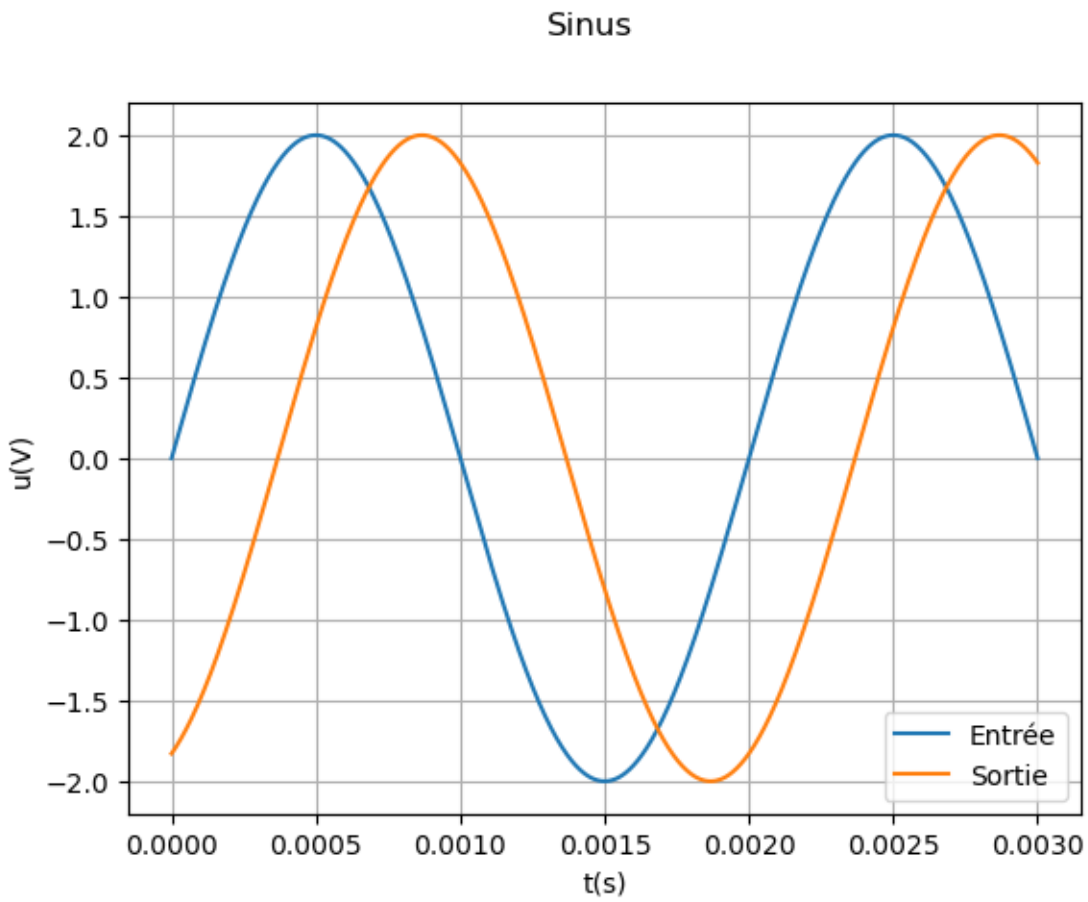
```
"""Cas du sinus"""
sig1 = sinus1

reps = reponse(butterworth, sig1)
print(reps)
t = np.linspace(0, 3e-3, 1000)
entree_temp = freq2temp(sig1, t)  # Passage au temporel pour l'entrée
sortie_temp = freq2temp(reps, t)  # Passage au temporel pour l'entrée

f, ax = plt.subplots()
f.suptitle("Sinus")
ax.set_xlabel('t(s)')
ax.set_ylabel('u(V)')
ax.plot(t, entree_temp, label="Entrée")
ax.plot(t, sortie_temp, label="Sortie")
ax.grid()  # Facultatif
ax.legend()
plt.show()

"""Cas de la somme"""
sig1 = somme1

reps = reponse(butterworth, sig1)
print(reps)
t = np.linspace(0, 3e-3, 1000)
entree_temp = freq2temp(sig1, t)  # Passage au temporel pour l'entrée
sortie_temp = freq2temp(reps, t)  # Passage au temporel pour l'entrée

f, ax = plt.subplots()
f.suptitle("Somme")
ax.set_xlabel('t(s)')
ax.set_ylabel('u(V)')
ax.plot(t, entree_temp, label="Entrée")
```

(continues on next page)

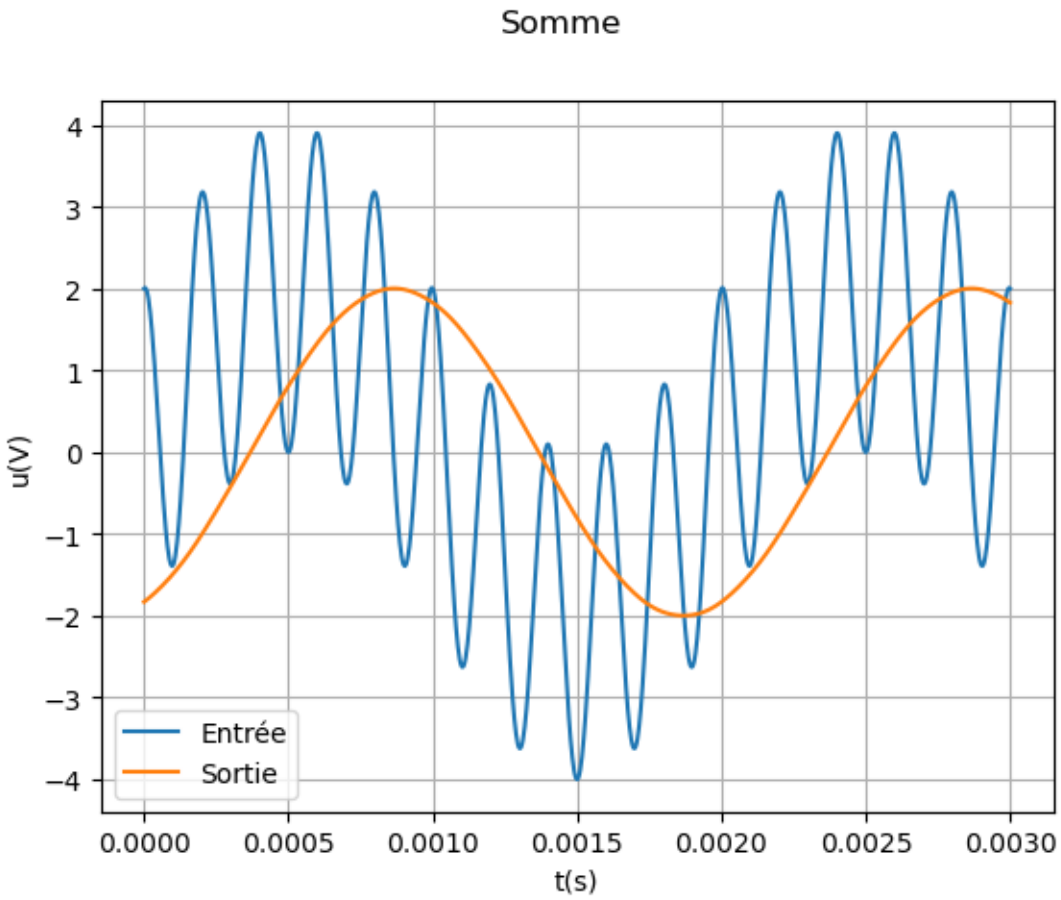
(continued from previous page)

```
ax.plot(t, sortie_temp, label="Sortie")
ax.grid()  # Facultatif
ax.legend()
plt.show()

"""Cas du créneau"""
sig1 = creneau(20)

reps = reponse(butterworth, sig1)
print(reps)
t = np.linspace(0, 3e-3, 1000)
entree_temp = freq2temp(sig1, t)  # Passage au temporel pour l'entrée
sortie_temp = freq2temp(reps, t)  # Passage au temporel pour l'entrée

f, ax = plt.subplots()
f.suptitle("Créneau")
ax.set_xlabel('t(s)')
ax.set_ylabel('u(V)')
ax.plot(t, entree_temp, label="Entrée")
ax.plot(t, sortie_temp, label="Sortie")
ax.grid()  # Facultatif
ax.legend()
plt.show()
```

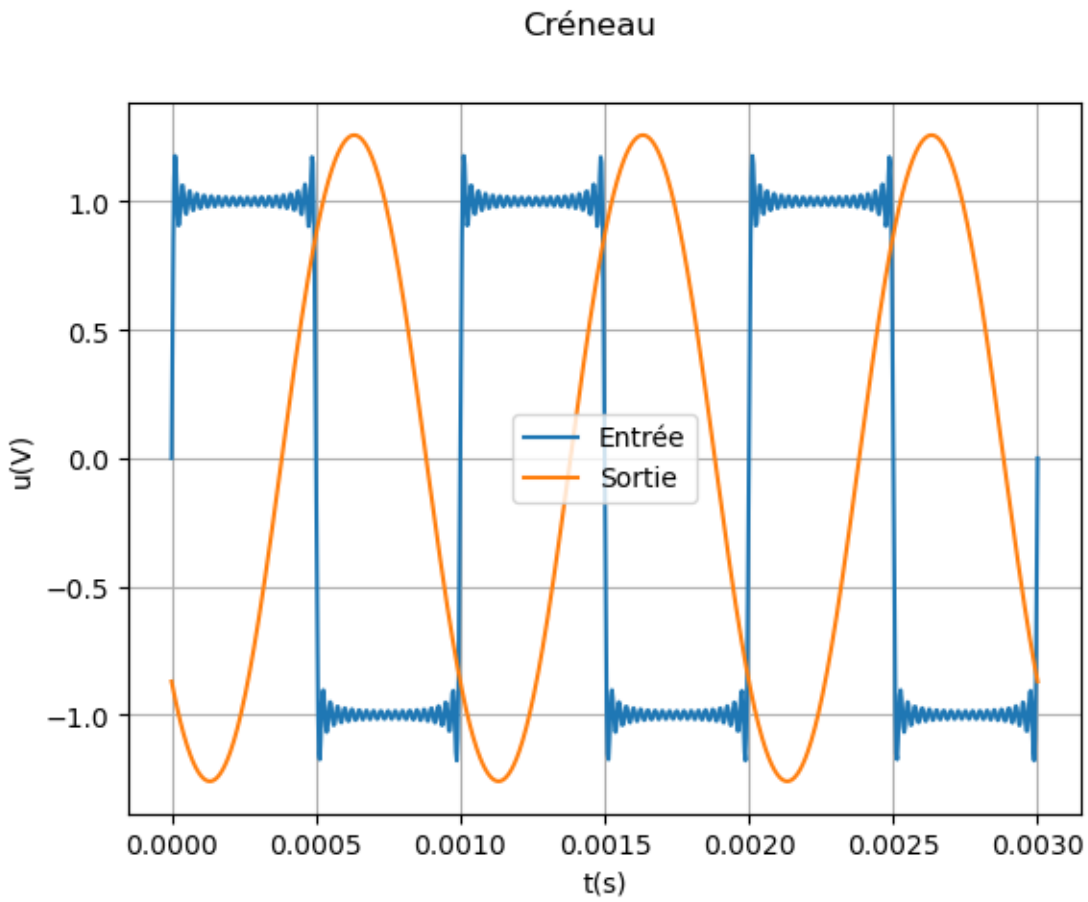
```
[[500, 1.9999995586217596, -1.153961258724054]]
```




```
[[500, 1.9999995586217596, -1.153961258724054], [5000, 0.003010398381814965, -0.
↪243514756696375 35]]
```




```
[[1000, 1.2720901670641669, -2.400535062344238], [3000, 0.013685161817802528, -0.
↪861053290702103], [5000, 0.00038329582651335566, -1.814311083491272], [7000, 3.
↪636120976090362e-05, -2.2004735460835425], [9000, 6.26075986257474e-06, -2.
↪4117481964311933], [11000, 1.5366530301306317e-06, -2.5453181250475194], [13000,␣
↪4.772227583977342e-07, -2.637472989356491], [15000, 1.752612074602213e-07, -2.
↪70491637385404], [17000, 7.297635426249076e-08, -2.7564237452350637], [19000, 3.
↪350035476226061e-08, -2.797051535843489], [21000, 1.662606789549357e-08, -2.
↪829920030009844], [23000, 8.794876810747181e-09, -2.8570596887125745], [25000, 4.
↪906192137168885e-09, -2.879848995116354], [27000, 2.8627159893486543e-09, -2.
↪899256813809156], [29000, 1.7359556574234064e-09, -2.915984078099162], [31000, 1.
↪0884105831473884e-09, -2.9305504505685667], [33000, 7.026305579634404e-10, -2.
↪943349381232317], [35000, 4.654234942393066e-10, -2.9546842417815076], [37000, 3.
↪1543693512464164e-10, -2.9647927149783317], [39000, 2.1820885157905781e-10, -2.
↪973863665036163]]
```



La page ci-présente existe en version notebook téléchargeable grâce au bouton (choisir le format `.ipynb`). On rappelle qu'l faut ensuite l'enregistrer dans un répertoire adéquat sur votre ordinateur (`capa_num` par exemple dans votre répertoire personnel) puis lancer Jupyter Notebook depuis Anaconda pour accéder au notebook, le modifier et exécutez les cellules de code adéquates.

CHAPTER

THREE

SYSTÈME CONSERVATIF : EFFETS NON LINÉAIRES.

But : résoudre numériquement une équation différentielle du deuxième ordre non-linéaire et faire apparaître l'effet des termes non-linéaires.

3.1 Position du problème

Rappel : Le système HCl est supposé isolé. On suppose que l'interaction intramoléculaire est modélisée par l'énergie potentielle :

$$E_p = \frac{C}{r^n} - \alpha \frac{e^2}{4\pi\epsilon_0 r}$$

Les constantes précédentes prennent alors les valeurs suivantes en unités arbitraires (UA = Unité arbitraire):

$C = 1.4 \times 10^{-1}(UA); \alpha = 0.40; n = 12; \epsilon_0 = 3.45 \times 10^{-1}(UA); e = 1(UA); m_H = 1(UA)$

3.1.1 Etude de l'énergie potentielle.

Exercice 1 :

- Définition de la fonction `Ep(r:float)->float` puis de la fonction force par dérivation numérique.
- Tracé de la fonction
- Recherche par dichotomie du 0 de `force(r, pas)` pour déterminer la position d'équilibre.

```
"""
N'oubliez pas les bibliothèques scientifiques
et pensez à commenter votre code.
"""
from numpy import *
from matplotlib.pyplot import *

C = 1.4e-1
alpha = 0.4
n = 12
epsilon0 = 3.45e-1
e = 1
m = 1

def eP(r:float)->float:
    return C / r ** n - alpha * e ** 2 / (4 * epsilon0 * pi * r)
```

(continues on next page)

(continued from previous page)

```
def force(r:float, pas:float)->float:
    """On utilise le fait que F = -dEp/dr
    On calcule une dérivée au centre.
    """
    return -(eP(r + pas) - eP(r - pas)) / (2 * pas)

"""Tracé graphique de Ep et F"""
rs = linspace(1, 10, 1000)  # Abscisse pour le tracé
ePs = eP(rs)  # On utilise la vectorialisation
pas = 1e-6
forces = force(rs, pas)  # On utilise la vectorialisation

f, ax = subplots(2, 1, sharex='col')  # On pourra séparer les deux graphiques.
f.suptitle("Energie potentielle et force d'interaction pour HCl")
ax[0].set_ylabel("Ep(U.A.)")
ax[1].set_xlim(1,10)
ax[1].set_xlabel("r(A)")
ax[1].set_ylabel("F(U.A.)")
ax[0].plot(rs, ePs)
ax[1].plot(rs, forces)
ax[0].grid()
ax[1].grid()
show()
```

```
"""Détermination de la position d'équilibre"""
r1 = 1
r2 = 2 # Le tracé graphique montre que rE est compris en 1 et 2 Angstrom.
prec = 1e-4
while (r2 - r1) > prec:
    rc = (r2 + r1) / 2 # Milieu
    if force(rc, pas)== 0:  # Le milieu est la solution
        r2 = rc
        r1 = rc  # On réduit l'intervalle à rc
    elif force(rc, pas)*force(r1, pas) < 0:  # Solution entre r1 et rc
        r2 = rc
    else:
        r1 = rc

print(rc)
```

On trouve donc que $r_E = 1.3019A$.

3.2 Schéma d'Euler pour un équation d'ordre 2

Exercice 2:

3.3 Application à l'étude de la vibration

3.3.1 Effets de non linéarité

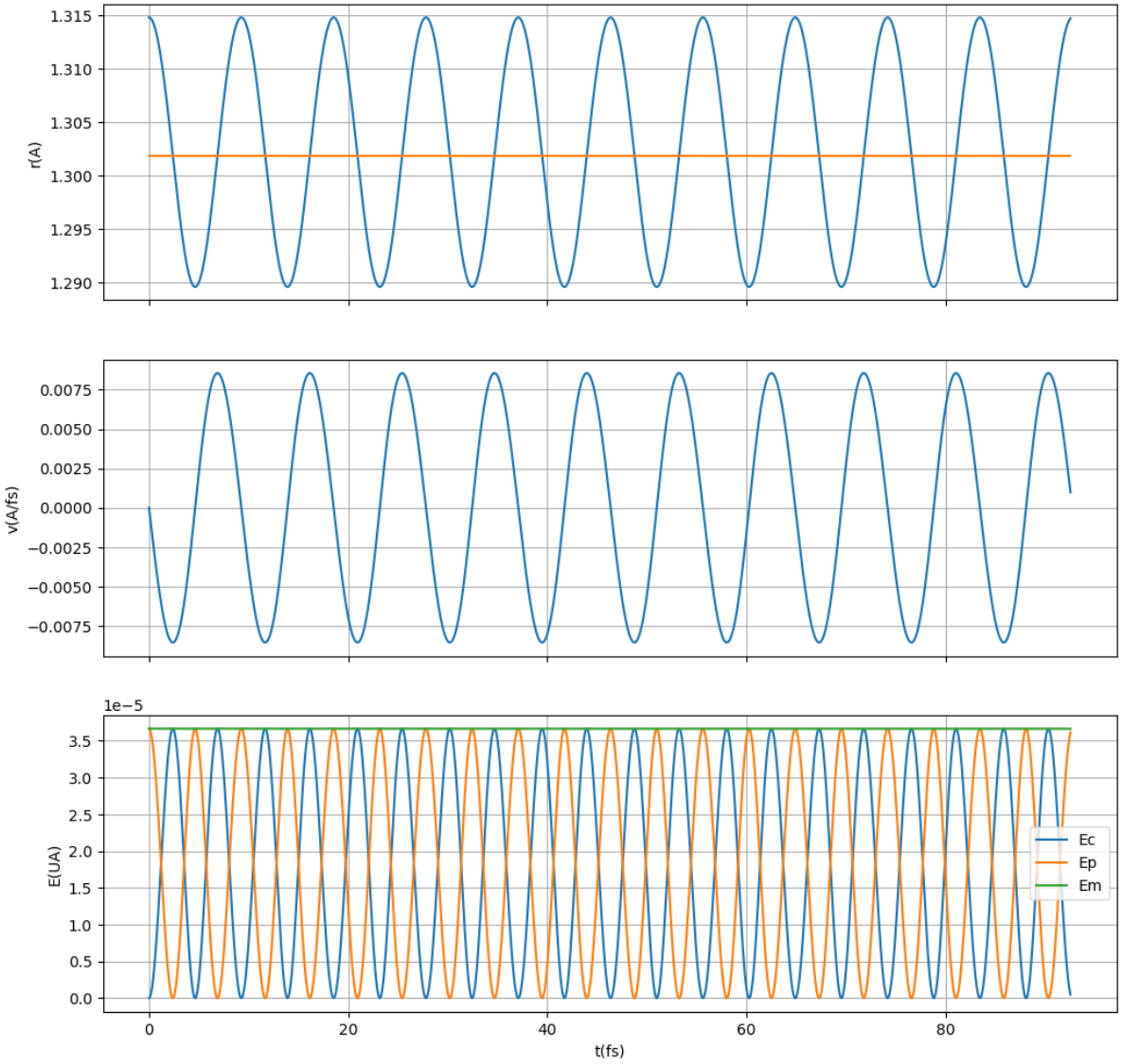
Exercice 3:
On choisit de réaliser l'intégration sur un intervalle de temps comptant plusieurs fois (10 fois) la période propre des petites oscillations (pour tenir compte de la non linéarité qui peut augmenter la période réelle des oscillations aux grandes amplitudes). La période propre s'obtient par $T_0 = 2\pi\sqrt{\frac{m}{\frac{\mathrm{d}^2 E_p}{\mathrm{d}t^2}(r_E)}}$

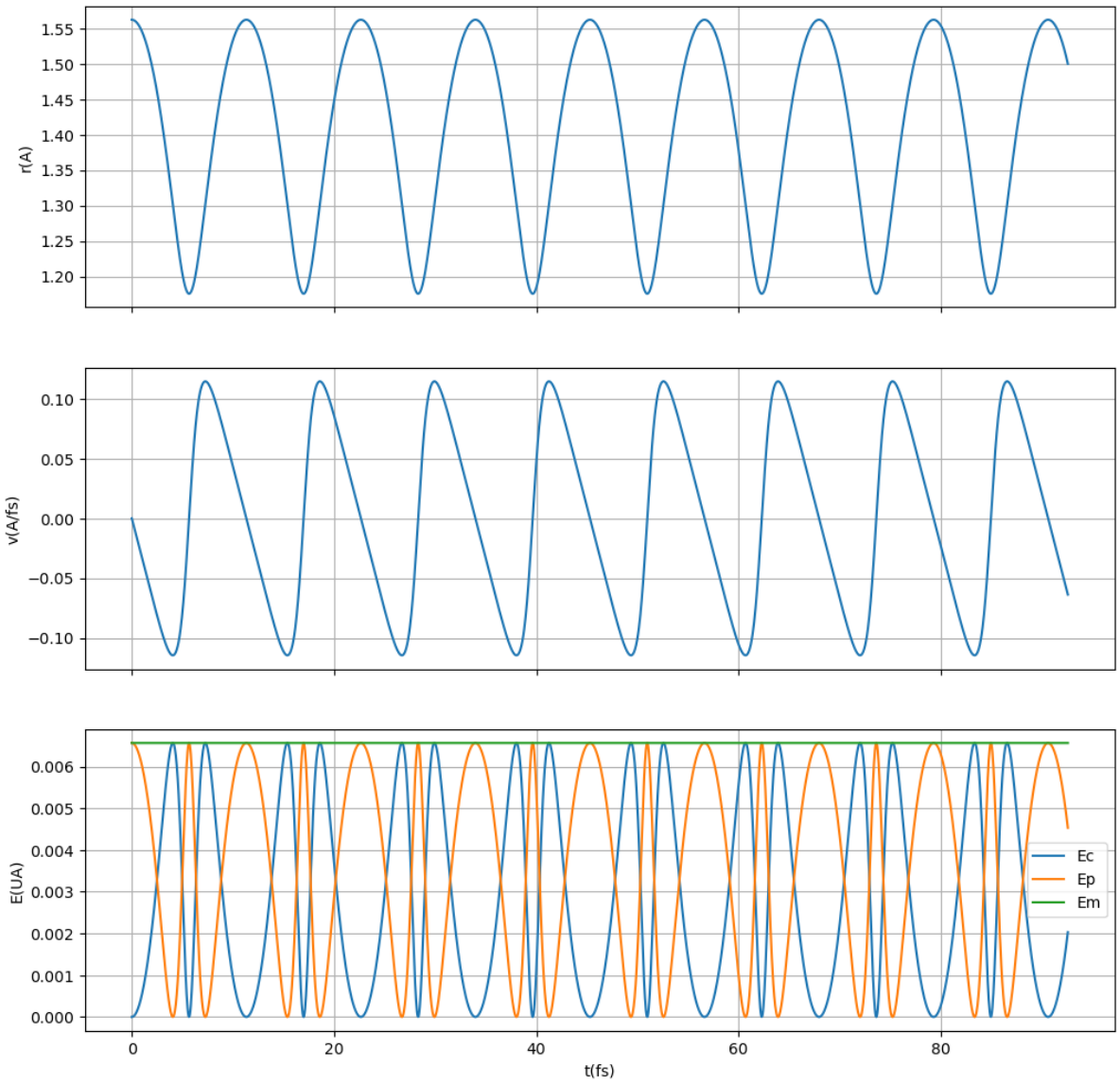
```
"""Ne pas oublier d'importer la fonction odeint"""
def eC(v: float) ->float:
    return 1 / 2 * m * v ** 2

# Période propre aux petits mouvement
k = -(force(rc+pas, pas) - force(rc-pas, pas)) / (2 * pas)
w0 = sqrt(k/m)
t0 = 2 * pi / w0
tf = 10 * t0  # Choix du temps final

tks = arange(0, tf, tf / 2000)

from scipy.integrate import odeint
r0 = 1.01 * rc  # Point de départ - Faibles oscillations
y0 = array([r0, 0])  # Conditions initiales
yks = odeint(f, y0, tks, tfirst=True)
Ecks = eC(yks[:,1])
Epks = eP(yks[:,0])-eP(rc)  # Soustraction non demandée mais elle permet de␣
 ↪visualiser les variations des grandeurs
Emks = Epks + Ecks

f1, ax1 = subplots(3, 1, sharex='all', figsize=(12,12))
ax1[0].set_ylabel("r(A)")
ax1[1].set_ylabel("v(A/fs)")
ax1[2].set_ylabel("E(UA)")
ax1[2].set_xlabel("t(fs)")
ax1[0].plot(tks, yks[:,0])
ax1[0].plot([tks[0], tks[-1]], [rc, rc])
ax1[1].plot(tks, yks[:,1])
ax1[2].plot(tks, Ecks, label='Ec')
ax1[2].plot(tks, Epks, label='Ep')
ax1[2].plot(tks, Emks, label='Em')
ax1[2].legend()
ax1[0].grid()
ax1[1].grid()
ax1[2].grid()
show()
```

(continues on next page)

(continued from previous page)

```
r0 = 1.2 * rc  # Point de départ - Faibles oscillations
y0 = array([r0, 0])  # Conditions initiales
yks = odeint(f, y0, tks, tfirst=True)
Ecks = eC(yks[:,1])
Epks = eP(yks[:,0])-eP(rc)  # Soustraction non demandée mais elle permet de␣
↪visualiser les variations des grandeurs
Emks = Epks + Ecks

f1, ax1 = subplots(3, 1, sharex='all', figsize=(12,12))
ax1[0].set_ylabel("r(A)")
ax1[1].set_ylabel("v(A/fs)")
ax1[2].set_ylabel("E(UA)")
ax1[2].set_xlabel("t(fs)")
ax1[0].plot(tks, yks[:,0])
ax1[1].plot(tks, yks[:,1])
ax1[2].plot(tks, Ecks, label='Ec')
ax1[2].plot(tks, Epks, label='Ep')
ax1[2].plot(tks, Emks, label='Em')
ax1[2].legend()
ax1[0].grid()
ax1[1].grid()
ax1[2].grid()
show()
```


r(A)
1.315
1.310
1.305
1.300
1.295
1.290
v(A/fs)
0.0075
0.0050
0.0025
0.0000
−0.0025
−0.0050
−0.0075
1e−5
E(UA)
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
20
40
60
80
t(fs)
Ec
Ep
Em



3.4 Trajectoire de diffusion

Exercice 5:
Il faut que l'énergie mécanique soit supérieur à Em. On peut soit partir de r_E avec une vitesse suffisante ou prendre r assez petit pour que $E_p(r) > 0$. On choisit la deuxième solution. On remarque sur le tracé graphique que $E_p(r = 1A) > 0$, on prend donc cette valeur.

```
r0 = 1  # Point de départ - Faibles oscillations
y0 = array([r0, 0])  # Conditions initiales
yks = odeint(f, y0, tks, tfirst=True)
Ecks = eC(yks[:,1])
Epks = eP(yks[:,0])-eP(rc)  # Soustraction non demandée mais elle permet de␣
↪visualiser les variations des grandeurs
Emks = Epks + Ecks
```

(continues on next page)

(continued from previous page)

```
f1, ax1 = subplots(3, 1, sharex='all', figsize=(12,12))
ax1[0].set_ylabel("r(A)")
ax1[1].set_ylabel("v(A/fs)")
ax1[2].set_ylabel("E(UA)")
ax1[2].set_xlabel("t(fs)")
ax1[0].plot(tks, yks[:,0])
ax1[1].plot(tks, yks[:,1])
ax1[2].plot(tks, Ecks, label='Ec')
ax1[2].plot(tks, Epks, label='Ep')
ax1[2].plot(tks, Emks, label='Em')
ax1[2].legend()
ax1[0].grid()
ax1[1].grid()
ax1[2].grid()
show()
```