BANQUE PT 2020 : CORRIGE DE LA PARTIE INFORMATIQUE DE L'EPREUVE D'INFORMATIQUE ET MODELISATION DE SYSTEMES PHYSIQUES

ALGORITHME DE CONVOLUTION APPLIQUE A LA REVERBERATION

A/ Produit de convolution : réalisation numérique. (15 %)

A.1. Signaux numériques à traiter.

Q1. Pour n piquets, il y a (n – 1) intervalles : $\boxed{T = (n-1)dt}$.

Q2.

```
T = 10      # durée en s
fe = 100    # fréquence d'échantillonnage en Hz
n = 1 + T * fe          # multiplier deux entiers donne un entier
# ou bien :
# dt = 1 / fe
# n = int(1 + T / dt)   # attention, diviser 2 entiers donne un flottant
tps = []
for k in range(n):
    tps.append(k / fe)

# autre possibilité : par compréhension
T = 10      # durée en s
fe = 100    # fréquence d'échantillonnage en Hz
tps = [k / fe for k in range(1 + T * fe)]

# autre possibilité mais qui n'utilise pas des listes, mais des tableaux Numpy
import numpy as np
T = 10      # durée en s
fe = 100    # fréquence d'échantillonnage en Hz
n = 1 + T * fe          # multiplier deux entiers donne un entier
tps = np.linspace(0, T, n)  # borne supérieure incluse

# autre possibilité mais qui n'utilise pas des listes, mais des tableaux Numpy
import numpy as np
T = 10      # durée en s
fe = 100    # fréquence d'échantillonnage en Hz
dt = 1 / fe
tps = np.arange(0, T + dt, dt)  # borne supérieure exclue
```

Q3.

```
from math import *

def sigh(t):
    h = []
    for i in range(len(t)):
        hi = cos(2 * pi * 12 * t[i]) * exp(- t[i])
        h.append(hi)
    return h

# autre possibilité : par compréhension

def sigh_bis(t):
    return [cos(2 * pi * 12 * Tps) * exp(- Tps) for Tps in t]
```

Q4.

```
def sigx(t):
    a = [1, 0.5, 0.2, 0.1]
    fx = [0.5, 5, 10, 20]
    x = []
    for i in range(len(t)):
        xi = 0
        for k in range(4):
            xi = xi + a[k] * cos(2 * pi * fx[k] * t[i] + k * pi)
        x.append(xi)
    return x
```

Q5.

```
import matplotlib.pyplot as plt
plt.figure()                          # crée une fenêtre de tracé vide
plt.plot(tps, sigx(tps), 'b-')        # tracé de x(t)
plt.plot(tps, sigh(tps), 'g-')        # tracé de h(t)
plt.legend(["x(t)", "h(t)"])          # légende du graphique
plt.xlabel('temps (s)')               # étiquette des abscisses
plt.show()                            # affichage des différentes figures
```

A.2. Convolution directe.

Q6.

```
def convolution(x, h, t):
    y = []
    n = len(t)      # n est le nombre d'instants
    dt = t[1] - t[0]    # le pas est supposé constant, on le calcule ici
    for i in range(n):  # on calcule yi à chaque instant ti
        yi = 0
        for k in range(n - 1):  # on somme les (n-1) rectangles
            yi = yi + x[k] * h[i - k] * dt
        y.append(yi)
    return y
```

Q7. Il y a deux boucles for de longueur n et (n-1) qui sont imbriquées, donc la complexité est de l'ordre de n(n-1), donc n^2, donc quadratique, en $O(n^2)$.
En effet, il y a n valeurs de y_i à calculer, et pour chaque calcul de y_i, il y a (n-1) rectangles à sommer.

B/ Convolution par transformée de Fourier discrète (15 %)

B.1. Transformée de Fourier Discrète (TFD).

Q8. Il y a des pics au niveau des fréquences contenues dans le signal x(t) (0,5 Hz, 5 Hz, 10 Hz, 20 Hz). Et l'amplitude de ces pics décroit quand la fréquence augmente, tout comme l'amplitude des coefficients a_k (1 / 0,5 / 0,2 / 0,1). En ordonnée, on constate toutefois qu'on n'a pas les amplitudes a_k. Il semble que ça ne correspond même pas tout à fait aux amplitudes à une constante multiplicative près. Par exemple, pour 0,5 Hz, 500 / 1 = 500, et pour 10 Hz, 80 / 0,2 = 400 ≠ 500. Le rapport des pics du module de la transformée de Fourier X(f) et des amplitudes des différentes composantes du signal x(t) n'est donc pas tout à fait constant, l'algorithme utilisé n'est donc pas parfait. A ce détail près, le graphe correspond au spectre de fréquence du signal x(t), c'est-à-dire l'amplitude des différentes fréquences contenues dans le signal, en fonction de la fréquence. A noter également qu'il y a repliement du spectre, ce qui fait apparaitre des fréquences négatives opposées aux fréquences contenues dans le signal.

Q9. *L'expression (9) est bizarre… On intègre entre 0 et n.dt, ce qui fait (n+1) points. Alors que jusqu'à présent, il y avait n points et l'instant final était indicé (n-1)…*
Il y a également une maladresse : la variable d'intégration est t ce qui donne un dt dans l'intégrale et un dt dans les bornes. On aurait pu noter τ la variable temporelle, comme dans l'expression (4), et alors, on aurait eu : $X(f)=\int_0^{(n-1).dt} x(\tau).\exp(-j.2.\pi.f.\tau).d\tau$

Partons toutefois de l'expression proposée (avec (n+1) points : $X(f)=\int_0^{n.dt} x(t).\exp(-j.2.\pi.f.t).dt$

Pour la fréquence f_k, on a : $X_k=\int_0^{n.dt} x(t).\exp(-j.2.\pi.f_k.t).dt$

On somme la surface des rectangles, de largeur $dt=t_{l+1}-t_l$ et de hauteur $x(t_l).\exp(-j.2.\pi.f_k.t_l)$.
L'intégrale allant de 0 à n.dt, il y a donc cette fois (n+1) piquets et donc n intervalles, donc n rectangles de largeur dt à considérer : la somme va donc de l = 0 à l = n-1.
$X_k=\sum_{l=0}^{n-1} x_l.\exp(-j.2.\pi.f_k.t_l).dt$.
De plus, $f_k=k.f_e/n$, $t_l=l.dt=l/f_e$, $\omega_n=\exp(-j.2.\pi/n)$
Donc $\exp(-j.2.\pi.f_k.t_l)=\exp(-j.2.\pi.(k.f_e/n).(l/f_e))=\exp(-j.2.\pi.k.l/n)=(\exp(-j.2.\pi/n))^{k.l}=\omega_n^{k.l}$
Donc $\boxed{X_k=\sum_{l=0}^{n-1} x_l.\omega_n^{k.l}.dt}$

Q10.

```
from cmath import *     # pour pouvoir utiliser l'exponentielle complexe

def Fourier_directe(x):
    n = len(x)
    omega_n = exp(- 2j * pi / n)
    X = []
    for k in range(n):
        Xk = 0
        for l in range(n):
            Xk = Xk + x[l] * omega_n**(k*l)
        X.append(Xk)
    return X
```

Q11. Il y a deux boucles for de longueur n qui sont imbriquées, donc la complexité est <u>quadratique</u>, en $O(n^2)$.
En effet, il y a n valeurs de X_k à calculer, et pour chaque calcul de X_k, il y a n valeurs à sommer.

Q12. Les définitions sont très ressemblantes, mis à part le signe dans l'exponentielle.
Donc en posant $\omega_n=\exp(+j.2.\pi/n)$, on a : $x_k=\sum_{l=0}^{n-1} X_l.\omega_n^{k.l}.df$

```
def Fourier_directe_inverse(X):
    n = len(X)
    omega_n = exp(2j * pi / n)
    x = []
    for k in range(n):
        xk = 0
        for l in range(n):
            xk = xk + X[l] * omega_n**(k*l)
        x.append(xk)
    return x
```

B.2. Convolution avec la TFD.

Q13. On suppose que les 2 listes (x et h) sont de même longueur.

```
def produit(x, h):
    n = len(x)
    y = []
    for k in range(n):
        y.append(x[k] * h[k])
    return y

# autre possibilité : par compréhension

def produit_bis(x, h):
    n = len(x)
    return [x[k] * h[k] for k in range(n)]
```

Q14.

```
X = Fourier_directe(x)
H = Fourier_directe(h)
Y = produit(X, H)
y = Fourier_directe_inverse(Y)
```

Q15.

L'appel de la fonction « Fourier_directe » est de complexité quadratique, tout comme pour la fonction « Fourier_directe_inverse ».
L'appel de la fonction « produit » est de complexité linéaire (O(n)) (car une seule boucle for de longueur n).
Les différents appels à ces fonctions étant exécutés l'un après l'autre, la complexité de la détermination d'un produit de convolution de deux signaux en utilisant la transformée de Fourier discrète est quadratique.
A la question 7, on a vu que la complexité du produit de convolution direct est également quadratique.
Il n'y a donc aucun gain en termes de complexité. La complexité est même sans doute moins bonne (succession d'appels de complexité quadratique). Et c'est effectivement le cas (cf Figure 7).

C/ Convolution par transformée de Fourier rapide (FFT). (10 %)

Q16. La fonction « Fourier_rapide » se termine pour tout entier N > 0.

- Pour N = 1, la condition d'arrêt retourne y.
- Pour N > 1, les deux appels récursifs se font avec N = N//2. N est donc diminué strictement car : $1 \leq N//2 < N$.

Et donc N va finir par devenir égal à 1, et l'algorithme se terminera grâce à la condition d'arrêt.

Q17.

```
def nextpow2(n):
    p = 0
    while 2**p <= n:
        p = p + 1
    return 2**(p-1)

# autre possibilité : version récursive

def nextpow2_bis(n):
    if n == 1:
        return 1
    else:
        return 2 * nextpow2_bis(n//2)
```

Q18. Si $N = 2^p$, un appel à la fonction « Fourier_rapide » comporte une boucle for de $N/2 = 2^{p-1}$ itérations avec 3 multiplications par W à l'intérieur, soit $3 \times 2^{p-1}$ multiplications intrinsèques à un appel.
Si on compte les appels récursifs, l'appel avec $N_0 = 2^p$ donne deux sous-appels récursifs (avec $N_1 = 2^{p-1}$) et $3 \times 2^{p-1}$ multiplications intrinsèques à l'appel (comme vu précédemment).
Le nombre total de multiplications est donc la somme sur chaque « niveau » d'appel : 2^k appels sur le niveau k (de 0 à p-1 (car pour N = 1, on tombe sur la condition d'arrêt, qui n'a pas de multiplication)) avec $N_k = 2^{p-k}$ et chaque appel coûte intrinsèquement $3 \times 2^{p-1-k}$ multiplications.
D'où le total : $\sum_{k=0}^{p-1} 2^k \times 3 \times 2^{p-1-k} = 3 \times 2^{p-1} \sum_{k=0}^{p-1} 1 = 3 \times p \times 2^{p-1}$ multiplications.
Or $N = 2^p$, donc $p = \log(N)/\log(2)$.
Le nombre total de multiplications est donc $\boxed{\frac{3}{2.\log(2)}.N.\log(N)}$.
La complexité est donc en $\boxed{O(N.\log(N))}$.

Autre rédaction possible :
En notant C(n) le nombre de multiplications réalisées lors de l'exécution de Fourier_rapide(x, n), on a, avec $n = 2^m$, une boucle for de $n/2 = 2^{m-1}$ itérations avec 3 multiplications par W à l'intérieur, soit $3 \times 2^{m-1}$ multiplications intrinsèques à l'appel. Et il y a également 2 sous-appels récursifs avec comme argument 2^{m-1}.
On a donc $C(2^m) = 3 \times 2^{m-1} + 2 \times C(2^{m-1})$, et donc : $\frac{C(2^m)}{2^m} = \frac{3}{2} + \frac{C(2^{m-1})}{2^{m-1}}$.
On reconnait une suite arithmétique de raison $\frac{3}{2}$, ce qui donne $\frac{C(2^m)}{2^m} = \frac{C(2^0)}{2^0} + \frac{3}{2}m = \frac{3}{2}m$, car C(1) = 0 (condition d'arrêt).
On en déduit : $C(2^m) = \frac{3}{2} m\, 2^m$, or $n = 2^m$, donc $m = \log(n)/\log(2)$.
Donc $C(n) = \frac{3}{2\log(2)} n \log(n)$.
Pour n = N, le nombre total de multiplications est donc $\boxed{\frac{3}{2.\log(2)}.N.\log(N)}$.
La complexité est donc en $\boxed{O(N.\log(N))}$.

Q19. Pour la convolution directe et pour la convolution TFD, si on multiplie N par 10 (1 décade), on multiplie le temps d'exécution par 100 (2 décades). Il y a une pente égale à 2 en échelle logarithmique. La complexité est donc en $O(N^2)$, comme nous avons pu l'établir dans les questions 7 et 15.
Pour la convolution FFT, la pente de la courbe est plus faible (et moins régulière, ce qui laisse penser que la complexité n'est pas une puissance de N). Si on multiplie N par 10, on multiplie le temps d'exécution par 12 environ dans la gamme de N proposée. La complexité est donc d'environ $N^{1,2}$ pour les N fournis, ce qui est cohérent avec une complexité plus faible que N^2 (en N.log(N)).
Donc la convolution FFT est plus intéressante car sa complexité temporelle est plus faible, comme nous l'avions vu à la question précédente.

D/ Application aux fichiers musicaux au format WAVE. (20 %)

D.1. Caractéristiques des fichiers audios au format WAVE.

Q20. Le critère de Nyquist-Shannon dit que pour échantillonner correctement un signal, il faut que la fréquence d'échantillonnage f_{ech} soit supérieure à 2 fois la fréquence maximale f_{max} contenue dans le signal : $f_{ech} > 2\, f_{max}$. Ainsi, il y aura au moins 2 points de mesures par période pour toutes les fréquences.
La condition de Nyquist-Shannon est bien vérifiée ici (44,1 kHz > 2 × 20 kHz).

Q21. Un des avantages du complément à 2 par rapport au complément à 1 est qu'on gagne un nombre négatif (-2^{n-1}) car « 0 » n'est codé que d'une seule manière, contrairement au cas du complément à 1.
Deuxième avantage par rapport au complément à 1 : on peut sommer directement deux nombres écrits en représentation binaire et ignorer la retenue (dans le cas du complément à 1, il faut encore ajouter la retenue au résultat).

Q22. Si on travaille avec des entiers non signés (code binaire naturel, sans complément à 2 donc), on peut représenter 2^{16} entiers sur 16 bits. Ces entiers vont du plus petit (0) au plus grand $(2^{16} – 1) = 65531$.
L'énoncé rappelle que pour la représentation binaire en complément à 2, le plus grand entier positif est $(2^{n-1} – 1) = (2^{15} – 1) = 32767$, et le plus petit entier négatif est $-2^{n-1} = -2^{15} = -32768$.

Q23. $-a = -32768 = -2^{15}$ est codé par le code binaire naturel de son complément à 2 : $2^{16} – a = 2^{16} – 2^{15} = 2^{15}$.
Or $(2^{15})_{10} = (1000\ 0000\ 0000\ 0000)_2 = (80\ 00)_{16}$ big endian = $(00\ 80)_{16}$ little endian
Donc la commande retourne :

```
b'\x00\x80'
```

Q24. $(2a\ 00)_{16}$ little endian = $(00\ 2a)_{16}$ big endian = $(0000\ 0000\ 0010\ 1010)_2 = 2^1 + 2^3 + 2^5 = 2 + 8 + 32 = 42$.
La commande retournera donc : 42.

Q25. La figure 10 montre que la concaténation est supportée par les « bytes » (le symbole « + » ne correspond pas à une somme pour les « bytes », mais à une concaténation).

D.2. Convolution de deux fichiers audios au format WAVE.

Q26. et Q27.
Dans l'écriture de la fonction « creawave_2oct », on peut trouver MaxiG qui est la valeur maximale de la valeur absolue des éléments de la liste « voieG ». Ainsi, en divisant chaque élément de la liste (voieG[i]) par MaxiG, et en multipliant par 32767 (plus grand entier représentable sur 2 octets), on norme l'amplitude des signaux, comme demandé (ne pas oublier de convertir en entier « int »).
Pour l'écriture dans le fichier WAVE, on concatène les bytes 2 par 2.

```
def creawave_2oct(voieG,voieD,destination):
    #...
    for i in range(int(duree * frequence)):
        g = int(voieG[i] / MaxiG * 32767)
        gbyte = int.to_bytes(g, 2, byteorder='little', signed=True)
        d = int(voieD[i] / MaxiD * 32767)
        dbyte = int.to_bytes(d, 2, byteorder='little', signed=True)
        wavef.writeframesraw(gbyte+dbyte)   # concaténation des bytes
    #...
```

Q28. *L'énoncé n'est pas clair… Faut-il toujours considérer que len(voie) < N ou pas ?*

Si on est sûr que len(voie) < N, on peut écrire :

```
def addzeros(voie, N):
    L = []  # on crée une nouvelle liste pour ne pas modifier la liste "voie"
    for i in range(len(voie)):
        L.append(voie[i])
    for i in range(N - len(voie)):
        L.append(0)
    return L
```

Sinon, il vaut mieux écrire :

```
def addzeros2(voie, N):
    L = []  # on crée une nouvelle liste pour ne pas modifier la liste "voie"
    if len(voie) < N:
        for i in range(len(voie)):
            L.append(voie[i])
        for i in range(N - len(voie)):
            L.append(0)
    else:
        L = voie
    return L
```

Q29.

```
def convolution_reverb(fichier1, fichier2, fichier3):
    # on extrait les voies gauche et droite de chaque fichier
    voieG1, voieD1 = extrawave_2oct(fichier1)
    voieG2, voieD2 = extrawave_2oct(fichier2)
    # on détermine la longueur de chaque fichier
    N1 = len(voieG1)
    N2 = len(voieG2)
    # on calcule le nombre d'échantillons
    N = nextpow2(N1 + N2 - 1)
    # on redimensionne les listes
    voieG1_longN = addzeros(voieG1, N)
    voieD1_longN = addzeros(voieD1, N)
    voieG2_longN = addzeros(voieG2, N)
    voieD2_longN = addzeros(voieD2, N)
    # on effectue le produit de convolution (cf Q14) :
    # on commence par calculer les transformées de Fourier discrètes
    FrapG1 = Fourier_rapide(voieG1_longN, N)
    FrapG2 = Fourier_rapide(voieG2_longN, N)
    FrapD1 = Fourier_rapide(voieD1_longN, N)
    FrapD2 = Fourier_rapide(voieD2_longN, N)
    # on effectue les produits
    produit_G = produit(FrapG1, FrapG2)
    produit_D = produit(FrapD1, FrapD2)
    # on calcule les transformées de Fourier discrètes inverses
    voieG3 = Fourier_rapide_inverse(produit_G, N)
    voieD3 = Fourier_rapide_inverse(produit_D, N)
    # on crée le fichier wave à partir des données sur les deux voies
    creawave_2oct(voieG3, voieD3, fichier3)
```

Q30. Dans la figure 13, les signaux sont plus étirés, plus allongés, que dans la figure 8, ce qui correspond bien à un effet de réverbération.
Vérification numérique sur le paquet d'ondes central pour la voie gauche :

- Sur le signal de la figure 8 : durée = 1,2 cm / 10,8 cm × 2,530 s = 0,28 s.
- Sur le signal de la figure 13 : durée = 3 à 4 cm / 15,8 cm × 2,972 s = 0,56 à 0,75 s (> 0,28 s, OK).