Corrigé

Transformée de Fourier discrète

Durant ce TP on utilisera les modules suivants :

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.io.wavfile as wave
```

Transformée de Fourier discrète

Question 1. Si pour représenter les signaux on utilise des tableaux numpy, il faut bien penser à déclarer que ceux-ci sont destinés à contenir des nombres complexes au moment de leur création.

```
def dft(s):
    n = len(s)
    S = np.zeros(n, dtype=complex)
    for k in range(n):
        for l in range(n):
            S[k] += s[l] * np.exp(-2j * np.pi * k * l / n)
    return S
```

```
def idft(S):
    n = len(S)
    s = np.zeros(n, dtype=complex)
    for l in range(n):
        for k in range(n):
            s[l] += S[k] * np.exp(2j * np.pi * k * l / n)
    return s / n
```

Ces deux fonctions ont une complexité en $O(n^2)$.

Échantillonnage et théorème de Shannon

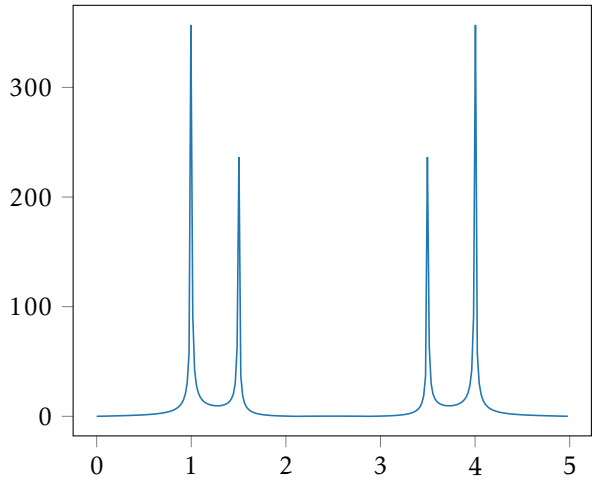
On définit la fonction ϕ :

```
def phi(x):
    return 3 * np.sin(2 * np.pi * x) + 2 * np.cos(3 * np.pi * x)

phi = np.vectorize(phi)
```

La dernière ligne permet de « vectorialiser » cette fonction, ce qui en Python signifie qu'il est désormais possible de l'appliquer à un tableau numpy (c'est-à-dire qu'on applique ϕ à chaque élément du tableau).

Question 2. On réalise le script suivant :

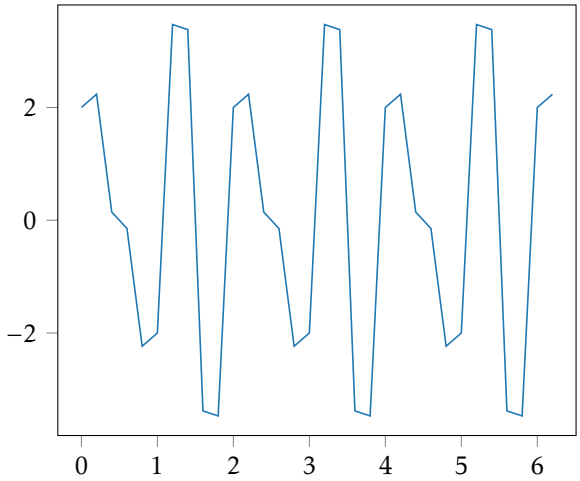
```
fe = 5
n = 256
t = np.arange(0, n / fe, 1 / fe)
s = phi(t)  # possible car la fonction phi a été vectorialisée

f = np.arange(0, fe, fe / n)
S = dft(s)

plt.plot(f, abs(S))
plt.show()
```



Reconstitution d'un signal périodique

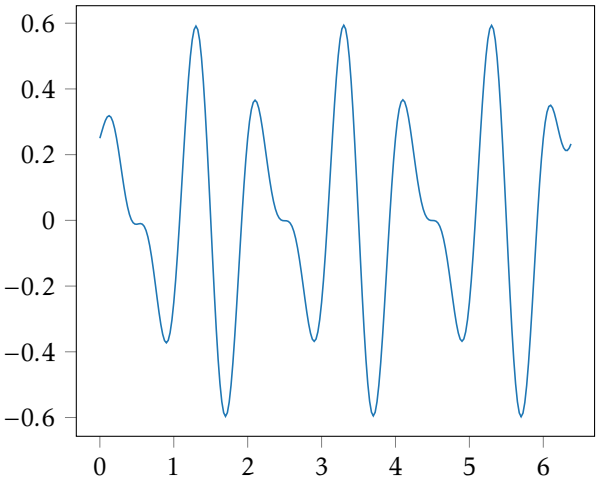
Question 3. Commençons par représenter le signal ϕ échantillonné à la fréquence $f_e = 5$, avec 32 échantillons.

```
fe = 5
n = 32
t = np.arange(0, n / fe, 1 / fe)
s = phi(t)
plt.plot(t, s)
plt.show()
```



Calculons maintenant la transformée de Fourier discrète de s, puis réalisons les opérations suggérées :

```
S = dft(s)
S2 = np.zeros(8 * n, dtype=complex)
S2[:n//2] = S[:n//2]
S2[-n//2:] = S[-n//2:]

s2 = idft(S2)
t2 = np.arange(0, n / fe, 1 / (8 * fe))
plt.plot(t2, s2)
plt.show()
```



Transformée de Fourier rapide

Question 4. On a, en séparant les termes d'indices pairs des termes d'indices impairs :

$$S_k = \sum_{0\leqslant 2\ell\leqslant 2n-1} s_{2\ell}\, e^{-i\pi k(2\ell)/n} + \sum_{0\leqslant 2\ell+1\leqslant 2n-1} s_{2\ell+1}\, e^{-i\pi k(2\ell+1)/n} = \sum_{\ell=0}^{n-1} s_{2\ell}\, e^{-2i\pi k\ell/n} + e^{-ik\pi/n} \sum_{\ell 0}^{n-1} s_{2\ell+1}\, e^{-2i\pi k\ell/n} = S_k^p + e^{-i\pi k/n}\, S_k^i.$$

On calcule de même $S_{k+n} = S_k^p + e^{-i\pi(k+n)/n} S_k^i = S_k^p - e^{-i\pi k/n} S_k^i$.

Ces formules conduisent à la version récursive :

```
def fft(s):
    n = len(s)
    if n == 1:
        return s
    Sp, Si = fft(s[0::2]), fft(s[1::2])
    S = np.zeros(n, dtype=complex)
    for k in range(n // 2):
        S[k] = Sp[k] + np.exp(-2j * k * np.pi / n) * Si[k]
        S[n // 2 + k] = Sp[k] - np.exp(-2j * k * np.pi / n) * Si[k]
    return S
```

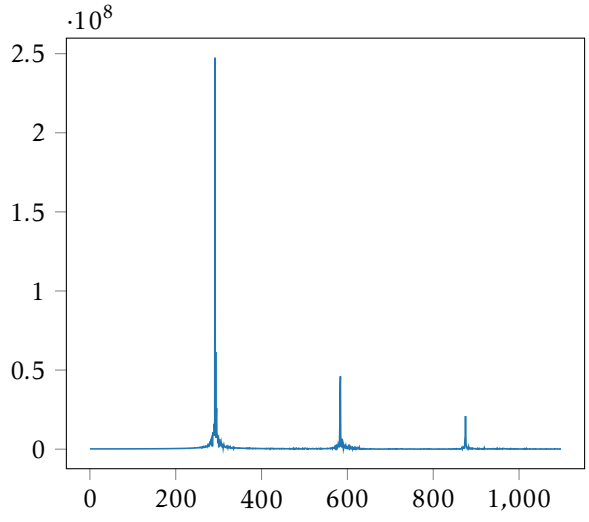
Lz calcul de `s[0::2]` et `s[1::2]` se réalise en temps linéaire $O(n)$; les deux appels récursifs ont une complexité en $O(n/2)$ et la calcul du tableau `S` a de nouveau une complexité en $O(n)$ donc la complexité $C(n)$ de cette fonction vérifie la relation : $C(n) = 2C(n/2) + O(n)$.
Si on pose $n = 2^p$ et $u_p = C(2^p)$ il existe une constante K telle que $u_p \leqslant 2u_{p-1} + K2^p$.
Alors $\frac{u_p}{2^p} - \frac{u_{p-1}}{2^{p-1}} \leqslant K$ et par télescopage $\frac{u_p}{2^p} - \frac{u_0}{2^0} \leqslant Kp$ soit $u_p \leqslant (Kp + u_0)2^p$ ou encore $C(n) \leqslant (K\log n + u_0)n$, ce qui prouve que $C(n) = O(n\log n)$.

Question 5. On a ici : $s_\ell = \frac{1}{2n}\sum_{k=0}^{2n-1} S_k e^{i\pi k\ell/n} = \frac{1}{2n}\left(\sum_{k=0}^{n-1} S_{2k} e^{2i\pi k\ell/n} + e^{i\pi\ell/n}\sum_{k=0}^{n-1} S_{2k+1} e^{2i\pi k\ell/n}\right) = \frac{1}{2}\left(s_\ell^p + e^{i\pi k\ell/n} s_\ell^i\right)$ et de même $s_{n+\ell} = \frac{1}{2}\left(s_\ell^p + e^{i\pi k(n+\ell)/n} s_\ell^i\right) = \frac{1}{2}\left(s_\ell^p - e^{i\pi k\ell/n} s_\ell^i\right)$. On en déduit la fonction :

```
def ifft(S):
    n = len(S)
    if n == 1:
        return S
    sp, si = ifft(S[0::2]), ifft(S[1::2])
    s = np.zeros(n, dtype=complex)
    for l in range(n // 2):
        s[l] = sp[l] + np.exp(2j * l * np.pi / n) * si[l]
        s[n // 2 + l] = sp[l] - np.exp(2j * l * np.pi / n) * si[l]
    return s / 2
```

Question 6. On réalise le script suivant :

```
rate, data = wave.read('flute.wav')

s = data[:2**16]
f = np.arange(0, rate, rate / len(s))
S = fft(s)

plt.plot(f[:1500], abs(S[:1500]))
plt.show()
```



La fréquence fondamentale se trouve entre 200 et 400 Hz ; pour la déterminer avec précision on recherche la fréquence correspondant à la valeur maximale du tableau `abs(S)` :

```
m = 0
for i in range(1500):
    if abs(S[i]) > m:
        m, fmax = abs(S[i]), f[i]
print(fmax)
```

On obtient une fréquence de 291,5 Hz, ce qui correspond à un ré.

Multiplication rapide de polynômes

Question 7. On réalise la fonction suivante :

```
def produit(P, Q):
    n = 1
    while n < len(P) or n < len(Q):
        n *= 2
    P1 = np.zeros(2 * n, dtype=complex)
    Q1 = np.zeros(2 * n, dtype=complex)
    P1[:len(P)] = P
    Q1[:len(Q)] = Q
    return ifft(fft(P1) * fft(Q1))
```

Le script qui suit choisit aléatoirement deux polynômes de degré 512 dont les coefficients sont des entiers de l'intervalle $[\![-10, 10]\!]$. Le produit de ces deux polynômes est calculé de deux façons différentes, puis on détermine l'erreur maximale commise sur les coefficients, pour constater qu'elle est effectivement très petite.

```
from numpy.polynomial import Polynomial
import numpy.random as rd

p = Polynomial(rd.randint(-10, 11, 512))
q = Polynomial(rd.randint(-10, 11, 512))

r1 = p * q
r2 = Polynomial(produit(p.coef, q.coef))

print(max(r1 - r2))
```

```
(6.8212102633e-13-3.08322120939e-13j)
```