Corrigé

Calcul approché d'une intégrale

Exercice 1. Calcul approché d'une intégrale par les méthodes de Newton-Côtes

Rappelons que les quatre méthodes étudiées en cours appliquent les formules suivantes :

$$\mathrm{R}_n(f)=\frac{b-a}{n}\sum_{k=0}^{n-1}f(x_k) \qquad \mathrm{M}_n(f)=\frac{b-a}{n}\sum_{k=0}^{n-1}f\Big(\frac{x_k+x_{k+1}}{2}\Big) \qquad \mathrm{T}_n(f)=\frac{b-a}{n}\sum_{k=0}^{n-1}\Big(\frac{f(x_k)+f(x_{k+1})}{2}\Big)$$

$$\mathrm{S}_n(f)=\frac{b-a}{6n}\sum_{k=0}^{n-1}\Big(f(x_k)+4f\Big(\frac{x_k+x_{k+1}}{2}\Big)+f(x_{k+1})\Big) \qquad \text{en ayant posé } x_k=a+k\frac{b-a}{n}.$$

On peut diminuer le nombre d'opérations effectuées en utilisant les formules suivantes pour les deux dernières méthodes :

$$\mathrm{T}_n(f)=\frac{b-a}{n}\Big(\sum_{k=1}^{n-1}f(x_k)+\frac{f(a)+f(b)}{2}\Big) \qquad \text{et} \qquad \mathrm{S}_n(f)=\frac{b-a}{n}\Big(\frac{1}{3}\sum_{k=1}^{n-1}f(x_k)+\frac{2}{3}\sum_{k=0}^{n-1}f\Big(\frac{x_k+x_{k+1}}{2}\Big)+\frac{f(a)+f(b)}{6}\Big).$$

```
def rectangle(f, a, b, n):
    h = (b - a) / n
    x, s = a, 0
    for k in range(n):
        s += f(x)
        x += h
    return h * s
```

```
def trapeze(f, a, b, n):
    h = (b - a) / n
    x, s = a + h, (f(a) + f(b)) / 2
    for k in range(n-1):
        s += f(x)
        x += h
    return h * s
```

```
def milieu(f, a, b, n):
    h = (b - a) / n
    x, s = a + h / 2, 0
    for k in range(n):
        s += f(x)
        x += h
    return h * s
```

```
def simpson(f, a, b, n):
    h = (b - a) / n
    x, s1 = a + h, 0
    for k in range(n-1):
        s1 += f(x)
        x += h
    x, s2 = a + h / 2, 0
    for k in range(n):
        s2 += f(x)
        x += h
    return h * (s1 / 3 + 2 * s2 / 3 + (f(a) + f(b)) / 6)
```

Commençons par définir la fonction $f : t \mapsto \mathrm{e}^{-t^2}$ et calculons son intégrale à l'aide de la fonction quad :

```
>>> def f(x): return np.exp(-x * x)
>>> v = quad(f, 0, 1)[0]
```

(la fonction quad retourne un couple (I, ε) où I est une valeur approchée de l'intégrale et ε une majoration de l'erreur commise).
On définit ensuite la fonction :

```
def precision(methode, n):
    return abs(methode(f, 0, 1, n) - v)
```

Pour employer une recherche dichotomique il faut connaître une valeur qui réalise la précision demandée ; c'est le rôle du paramètre `depart`. Cette valeur sera déterminée par essais successifs.
On définit alors :

```
def rang_min(methode, epsilon, depart):
    if precision(methode, depart) > epsilon:
        return None
    i, j = 0, depart
    while i + 1 < j:
        k = (i + j) // 2
        if precision(methode, k) <= epsilon:
            j = k
        else:
            i = k
    return j
```

On maintient lors de la recherche dichotomique l'invariant suivant : le rang minimal n_0 vérifie $i < n_0 \leqslant j$. La condition d'arrêt a lieu lorsque $j = i + 1$: dans ce cas on a nécessairement $n_0 = j$.

Voici les valeurs qu'on obtient pour une précision de 10^{-4} :

```
>>> rang_min(rectangle, 1e-4, 10000)
3161
>>> rang_min(trapeze, 1e-4, 100)
25
>>> rang_min(milieu, 1e-4, 100)
18
>>> rang_min(simpson, 1e-4, 10)
2
```

Pour une précision de 10^{-8} :

```
>>> rang_min(trapeze, 1e-8, 10000)
2477
>>> rang_min(milieu, 1e-8, 10000)
1751
>>> rang_min(simpson, 1e-8, 100)
16
```

Et pour une précision de 10^{-12} :

```
>>> rang_min(simpson, 1e-12, 1000)
151
```

On sait que les erreurs des méthodes des trapèzes et du point milieu sont *grosso-modo* proportionnelles à $\frac{1}{n^2}$ donc pour obtenir une précision de 10^{-12} il faudrait pousser environ jusqu'au rang 247 700 avec la méthode des trapèzes et jusqu'au rang 176 000 avec la méthode du point milieu.
L'expérience reste réalisable pour une précision de 10^{-10} et fournit des résultats conformes :

```
>>> rang_min(trapeze, 1e-10, 25000)
24767
>>> rang_min(milieu, 1e-10, 20000)
17512
```

En revanche il est plus difficile d'atteindre la précision 10^{-12} car pour de telles valeurs de n les incertitudes liées aux erreurs d'arrondi peuvent difficilement être négligées.

La méthode des rectangles a quant à elle une erreur proportionnelle à $\frac{1}{n}$ donc pour obtenir une précision de 10^{-8} il faudrait pousser environ jusqu'au rang 31 600 000 et pour une précision de 10^{-12} jusqu'au rang 316 000 000 000. Autant dire que c'est irréalisable en pratique.

Exercice 2. Méthode de Romberg

a) On sait que $\mathrm{T}(h) = h\Big(\sum_{k=0}^{n-1} f(a+kh) + \frac{f(b)-f(a)}{2}\Big)$ et $\mathrm{T}(h/2) = \frac{h}{2}\Big(\sum_{k=0}^{2n-1} f(a+kh/2) + \frac{f(b)-f(a)}{2}\Big)$. En séparant les termes d'indices pairs et inpairs dans $\mathrm{T}(h/2)$ on obtient :

$$\mathrm{T}(h/2) = \frac{h}{2}\Big(\sum_{k=0}^{n-1} f(a+kh) + \sum_{k=0}^{n-1} f\Big(a+(k+1/2)h\Big) + \frac{f(b)-f(a)}{2}\Big)$$

D'où : $\dfrac{4\mathrm{T}(h/2)-\mathrm{T}(h)}{3} = \dfrac{h}{3}\displaystyle\sum_{k=0}^{n-1} f(a+kh) + \dfrac{2h}{3}\sum_{k=0}^{n-1} f\Big(a+(k+1/2)h\Big) + h\dfrac{f(b)-f(a)}{2}$

$$= \frac{h}{6}\sum_{k=0}^{n-1}\Big(f(a+kh)+4f\Big(a+(k+1/2)h\Big)+f\Big(a+(k+1)h\Big)\Big)$$

On reconnait dans cette dernière expression la formule de Simpson composite, méthode dont l'erreur est en effet un $\mathrm{O}(h^4)$.

b) En posant $h = \dfrac{b-a}{2^{q-1}}$ les quantités $\mathrm{R}_{2,q-1}$ et $\mathrm{R}_{2,q}$ possèdent des développements limités de la forme :

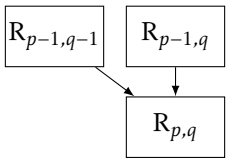
$$\mathrm{R}_{2,q-1} = \mathrm{I} + c_1 h^6 + \mathrm{O}(h^8) \quad \text{et} \quad \mathrm{R}_{2,q} = \mathrm{I} + c_1\frac{h^6}{64} + \mathrm{O}(h^8) \quad \text{donc} \quad \mathrm{R}_{3,q} = \frac{64\mathrm{R}_{2,q} - \mathrm{R}_{2,q-1}}{63} \text{ vérifie : } \mathrm{R}_{3,q} = \mathrm{I} + \mathrm{O}(h^8).$$

Plus généralement, en posant $\mathrm{R}_{p,q} = \dfrac{4^p\mathrm{R}_{p-1,q} - \mathrm{R}_{p-1,q-1}}{4^p - 1}$ on obtient par récurrence $\mathrm{R}_{p,q} = \mathrm{I} + \mathrm{O}(h^{2p+2})$.

c) Ceci conduit à la définition suivante :

```
def romberg(f, a, b, q):
    r = [[0 for j in range(q+1)] for i in range(q+1)]
    for j in range(q+1):
        r[0][j] = trapeze(f, a, b, 2**j)
    for i in range(1, q+1):
        for j in range(i, q+1):
            r[i][j] = (4**i * r[i-1][j] - r[i-1][j-1]) / (4**i - 1)
    return r[q][q]
```

On peut n'utiliser qu'un tableau uni-dimensionnel en observant que chacune des lignes ne dépend que de la ligne précédente en suivant le shéma de dépendance suivant :



La valeur de $\mathrm{R}_{p,q}$ peut donc remplacer la valeur de $\mathrm{R}_{p-1,q}$ à condition de remplir le tableau de la droite vers la gauche pour que $\mathrm{R}_{p-1,q-1}$ n'ait pas encore été remplacé par sa nouvelle valeur au rang p.
Ceci donne la version :

```
def romberg(f, a, b, q):
    r = [trapeze(f, a, b, 2**j) for j in range(q+1)]
    for i in range(1, q+1):
        for j in range(q, i-1, -1):
            r[j] = (4**i * r[j] - r[j-1]) / (4**i - 1)
    return r[q]
```

d) Dans la pratique, la valeur $q = 5$ suffit pour obtenir la valeur de l'intégrale $\mathrm{I} = \displaystyle\int_0^1 \mathrm{e}^{-t^2}\,\mathrm{d}t$ à la précision de 10^{-12} :

```
>>> rang_min(romberg, 1e-12, 10)
5
```

Exercice 3. Méthode de Monte-Carlo

On définit la fonction :

```
from numpy.random import random

def montecarlo(f, a, b, M, n):
    u = 0
    for k in range(n):
        x = a + (b - a) * random()
        y = M * random()
        if y < f(x):
            u += 1
    return M * (b - a) * u / n
```

(La fonction `random` renvoie un nombre flottant de l'intervalle $[0,1[$.)
Pour calculer la précision moyenne obtenue pour $n = 10\,000$ on réalise le script suivant :

```
f = lambda x: np.exp(- x * x)
v = quad(f, 0, 1)[0]
p = 0
for k in range(100):
    p += abs(montecarlo(f, 0, 1, 1, 10000) - v)
print("Précision moyenne : {}".format(p / 100))
```

qui nous apprend qu'en moyenne la précision obtenue est de l'ordre de 3.10^{-3}.
Pour adapter cette méthode au cas d'une fonction de signe quelconque, il faut supposer connu un minorant m de f et considérer les deux fonctions de signe constant $f^+ = \max(f,0)$ et $f^- = \min(f,0)$. On a alors :

$$\int_a^b f(t)\,\mathrm{d}t = \int_a^b f^+(t)\,\mathrm{d}t - \int_a^b f^-(t)\,\mathrm{d}t.$$

Concrètement, on commence par choisir au hasard x dans $[a,b]$ et :

- si $f(x) > 0$ on choisit y dans $[0,\mathrm{M}]$ pour calculer la valeur approchée de f^+ ;
- si $f(x) < 0$ on choisit y dans $[m,0]$ pour calculer la valeur approchée de f^-.

```
def montecarlo(f, a, b, m, M, n):
    u1, u2 = 0, 0
    for k in range(n):
        x = a + (b - a) * random()
        if f(x) > 0:
            y = M * random()
            if y < f(x):
                u1 += 1
        else:
            y = m * random()
            if y > f(x):
                u2 += 1
    return (b - a) * (M * u1 + m * u2) / n
```

Le script suivant permet de calculer une valeur approchée de l'intégrale demandée :

```
v = quad(np.log, .5, 2)[0]
u = montecarlo(np.log, .5, 2, np.log(.5), np.log(2), 100000)
print("Précision obtenue : {}".format(abs(v-u)))
```

et nous donne une précision de l'ordre de 5.10^{-4}.