TD 02: Études de convergence

1 Limites de suites et séries

1. Pour $x > 0$ réel on définit la suite $\{u_n\}_{n\in\mathbb{N}}$ par $u_0 = 0$ et la relation de récurrence $u_{n+1} = \dfrac{u_n + x}{u_n + 1}$. On admet que cette suite converge en général vers une limite $\ell(x)$ et on va considérer que la limite est atteinte dès lors que l'écart $|u_{n+1} - u_n|$ devient inférieur à une certaine valeur réelle ε, qui sera la *précision* de la convergence. On pourra faire varier ε entre 10^{-15} et 10^{-5} par exemple.

   (a) Écrire et tester une fonction **Python** d'en-tête `def lim(x, eps)` dont le résultat est formé d'une liste de deux nombres : la valeur (réelle) obtenue pour $\ell(x)$ et la première valeur (entière) de n telle que $|u_{n+1} - u_n| < \varepsilon$, où `eps` est la variable de valeur ε. La fonction se terminera donc par exemple par `return l, n`.

   (b) Tester cette fonction pour $\varepsilon = 10^{-10}$ fixé et x variable avec quelques valeurs choisies entre 1 et 9 par exemple. Que peut-on conjecturer pour la limite $\ell(x)$?

   (c) Tester cette fonction pour x fixé (de votre choix) et $\varepsilon = 10^{-p}$ où p varie de 5 à 15. Montrer que le nombre n d'itérations nécessaires à la convergence est directement proportionnel à p ; commenter.

2. On peut montrer (c'est la formule de Leibniz–Grégory) que $\pi = 4 \times \left[1 - \dfrac{1}{3} + \dfrac{1}{5} - \dfrac{1}{7} + \dfrac{1}{9} - \ldots\right]$. On convient d'arrêter le calcul lorsque le premier terme omis dans cette série alternée comporte au moins p zéros après la virgule. Par exemple $\dfrac{1}{99} = 0{,}010\ldots$ et $\dfrac{1}{101} = 0{,}0099\ldots$ donc le calcul limité à $p = 2$ zéros fournira la valeur approchée $\pi \simeq 4 \times \displaystyle\sum_{k=0}^{49} \frac{(-1)^k}{2k+1}$.

   (a) Écrire et tester une fonction **Python** d'en-tête `def valPi(p)` calculé par cette méthode avec un arrêt tel que défini ci-dessus.

   (b) Chronométrer le temps Δt_p de calcul en fonction de p et proposer une illustration graphique.

   Après `import time as tm` on notera que `t = tm.time()` donne une valeur de l'instant t en secondes (à partir d'une origine conventionnelle).

   Pour des listes (ou tableaux) de valeurs de mêmes dimension `X` et `Y` on obtient une représentation graphique de la courbe $y = f(x)$ au moyen des instructions :

```
import matplotlib.pyplot as pl

pl.figure()
pl.plot(X, Y)
pl.show()
```

   Pour un tracé en échelle semi-logarithmique il suffit de remplacer `pl.plot(X,Y)` par `pl.semilogx(X,Y)` ou `pl.semilogy(X,Y)` selon l'axe choisi. Pour une échelle doublement logarithmique on utilisera `pl.loglog(X,Y)`.

   (c) Compte tenu de la lenteur de la convergence de la formule de Leibniz–Grégory, on utilise aujourd'hui d'autres formules, comme celle de Ramanujan :

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9\,801} \sum_{k=0}^{\infty} u_k \quad \text{avec} \quad u_k = \frac{(4k)!}{(k!)^4} \frac{1\,103 + 26\,390 \times k}{396^{4k}}$$

   où on rappelle que $p!$ désigne la factorielle de p, soit $p! = 1 \times 2 \times \ldots \times p$. Écrire trois fonction **Python** qui calculent successivement $p!$, puis u_k et enfin π par la formule de Ramanujan avec n termes dans la somme. Vérifier qu'avec trois termes la précision dépasse celle du calcul de **Python**.

2 Autres cas de convergence

Pour déterminer un nombre réel aléatoire x dans l'intervalle $[0\,,\,1[$, et après importation de la bibliothèque *ad hoc* selon `import random as rd` il suffit de faire `x = rd.random()`.

3. Pour déterminer l'unique solution de l'équation $x = \cos(x)$, nous admettrons qu'il suffit de procéder ainsi :
   — choisir une valeur initiale arbitraire de x dans $[0\,,\,1[$ et une précision ε choisie ;
   — calculer $y = \cos(x)$, et recommencer à partir de la valeur précédente tant que $|y - x| > \varepsilon$;

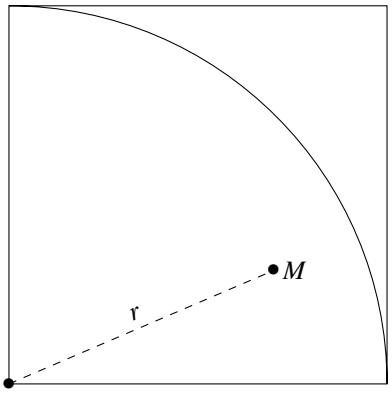
— lorsque x et $\cos(x)$ sont assez voisins, on a ainsi trouvé une approximation x de la solution exacte $x_0 = 0{,}739085133\ldots$.

(a) Écrire et tester une fonction **Python** d'en-tête `def x0(eps)` qui réalise cette détermination avec la précision `eps` et rend comme résultat l'estimation proposée pour x_0.

(b) Modifier la fonction pour qu'elle rende comme résultat le nombre N_c de calculs du cosinus qui ont été nécessaires à la résolution.

(c) Améliorer l'étude pour qu'elle ne dépende pas de la valeur initialement tirée au sort en faisant la moyenne d'un grand nombre d'estimations.

4. La méthode du comte de Buffon pour le calcul (statistique) de π consiste à tirer au sort deux coordonnées x et y toutes les deux dans l'intervalle $[0\,,\,1[$. On détermine ainsi les coordonnes d'un point M.



Si la distance $r = \sqrt{x^2+y^2}$ de M à l'origine vérifie $r \leqslant 1$, alors M est dans un quart de cercle de rayon $R = 1$, donc de surface $S = \dfrac{1}{4}\pi R^2 = \dfrac{\pi}{4}$. Que ce soit le cas ou non, les points ainsi tirés au sort sont tous dans un carré de côté $c = 1$ donc de surface $S' = c^2 = 1$. Finalement, la proportion de points ainsi tirés au sort qui vérifient $r \leqslant 1$ vaut $S/S' = \dfrac{\pi}{4}$. On peut donc calculer une estimation de π en :

— tirant au sort N couples de coordonnées (x,y) ;
— comptant parmi ceux-ci les N_c qui vérifient $x^2+y^2 \leqslant 1$;
— affirmant que, pour N assez grand, $\pi \simeq 4\dfrac{N_c}{N}$.

(a) Écrire et tester une fonction **Python** d'en-tête `def PiBuffon(N)` qui réalise les $2N$ tirages au sort et donne pour résultat une estimation de π.

(b) Écrire et tester une fonction **Python** d'en-tête `def PiErreur(N, p)` qui réalise p simulations indépendantes par la méthode de Buffon à N tirages, et donne comme résultat un majorant de l'écart $|\pi - \pi_{\text{simulé}}|$.

3 Solutions : limites de suites et séries

1. (a) Les deux variables u et v gèrent les valeurs de u_n et u_{n+1}.

```
def lim(x, eps):
    u = 0
    n = 0
    v = (u + x)/(u + 1)
    while abs(u - v) > eps:
        n += 1
        u, v = v, (v + x)/(v + 1)
    return u, n
```

   (b) Avec les lignes de test :

```
for x in range(11):
    l, n = lim(x, 1E-10)
    print(x, l)
```

   on se rend compte que $\ell(x) = \sqrt{x}$. On peut bien sûr aussi le démontrer.

   (c) Avec les lignes de test :

```
for p in range(5, 16):
    l, n = lim(2, 10**(-p))
    print(p, n)
```

   on se rend compte que $n \simeq 1{,}4 \times p$: augmenter la précision (en fait ici, le nombre de chiffres significatifs) impose d'augmenter de la même maière le nombre d'itérations, donc le temps de calcul. Ce quotient 1,4 dépend de x ; il est en fait en général un peu supérieur à $\sqrt{x}$.

2. (a) On peut proposer :

```
def valPi(p):
    g = 1
    k = 0
    s = 0
    inverr = 10**p
    while (2*k+1) < inverr:
        s += g/(2*k+1)
        g = -g
        k += 1
    return 4*s
```

   Attention l'exécution ralentit très vite si p augmente ; ne tentes pas de dépasser $p = 8$ par exemple !

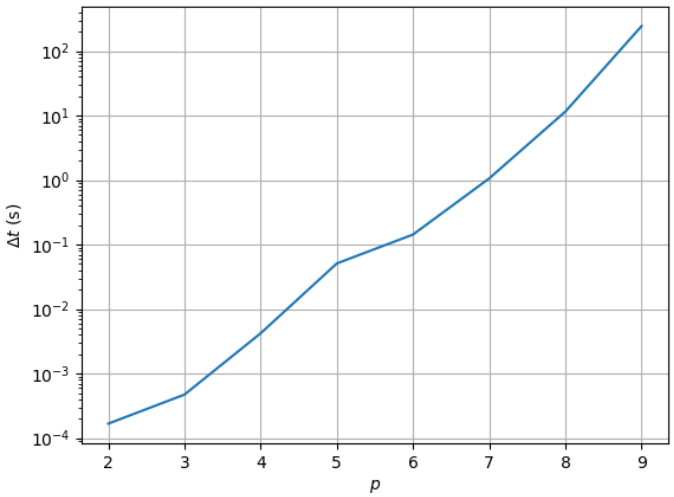
   (b) On commence par faire le calcul du temps d'exécution pour p variant de 2 à 9 (durée de calcul de l'ordre de plusieurs minutes dans ce dernier cas) :

```
import time as tm

LP = list(range(2,10))
LT = []

for p in LP:
    t = tm.time()
    print(p, valPi(p))
    t = tm.time() - t
    LT.append(t)
```

   et le tracé en échelle semi-logarithmique montre une croissance très rapide du temps de calcul : la méthode est très lente.



(c) Il n'y a qu'à coder :

```
def fact(p):
    r = 1
    for k in range(1,p+1):
        r *= k
    return r

def u(k):
    N = fact(4*k)
    N *= (1103+26390*k)
    D = fact(k)
    D *= 396**k
    D **= 4
    return N/D

def Ramanujan(n):
    s = 0
    for k in range(n):
        s += u(k)
    s *= 2**1.5
    s /= 9801
    return 1/s
```

On vérifie que dès la valeur `Ramanujan(3)` on retrouve toutes les décimales de `math.pi`, soit $\pi \simeq 3{,}141592653589793$. En effet $u_2 \sim 10^{-13}$ et $u_3 \sim 10^{-20}$.

4 Solutions : autres cas de convergence

3. (a) De façon évidente :

```
def x0(eps):
    x = rd.random()
    y = cos(x)
    while abs(y-x) > eps:
        x, y = y, cos(y)
    return x
```

(b) De même :

```
def x0(eps):
    r = 1
    x = rd.random()
    y = cos(x)
    while abs(y-x) > eps:
        x, y = y, cos(y)
        r += 1
    return r
```

(c) On peut ainsi constater qu'améliorer la précision d'un facteur 10 exige environ cinq itérations supplémentaires :

```
def x0moy(eps, ntir):
    k = 0
    for i in range(ntir):
        k += x0(eps)
    return k/ntir
```

4. (a) Il n'y a qu'à écrire :

```
import random as rd

def PiBuffon(N):
    p = 0
    for k in range(N):
        x = rd.random()
        y = rd.random()
        r2 = x*x + y*y
        if r2 <= 1:
            p += 1
    return 4*p/N
```

(b) De même :

```
from math import pi

def PiErreur(N, p):
    r = 0
    for k in range(p):
        r += PiBuffon(N)
    r = r/p
    return abs(pi - r)
```