T.P. d'informatique n° 7-1

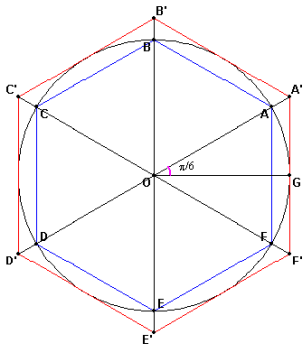
Calcul de π

Ce TP (largement inspiré du livre Python en Prepa édité par l'ISEN) est consacré à l'étude de différentes méthodes pour calculer les décimales du nombre π.

Convergence d'une suite

Méthode d'Archimède

Le premier calcul de π remonte à Archimède de Syracuse (287-212 avant J.C.). Celui-ci reposait sur un encadrement du périmètre du cercle par ceux de polygones réguliers inscrits et circonscrits.



Soit p_n le périmètre d'un polygone régulier à n cotés inscrit dans un cercle de diamètre 1 et p'_n celui d'un polygone régulier à n cotés circonscrit au même cercle. En utilisant l'inégalité $p_n < \pi < p'_n$ pour $n = 6 \times 2^k$, on peut obtenir une approximation de π avec :

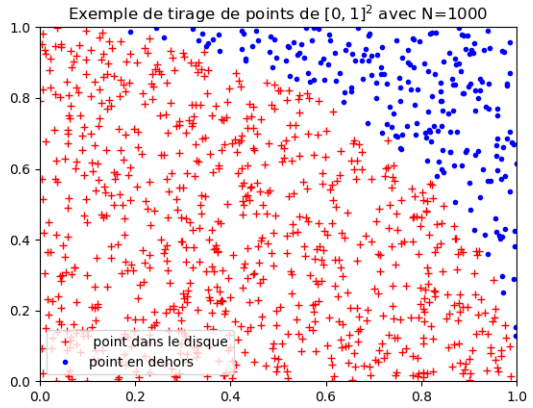
$$p_n = n\,u_n \text{ où } u_n = \sin\frac{\pi}{n},\ u_6 = \frac{1}{2} \text{ et } u_{2n} = \sqrt{\frac{1-\sqrt{1-u_n^2}}{2}} \text{ et}$$

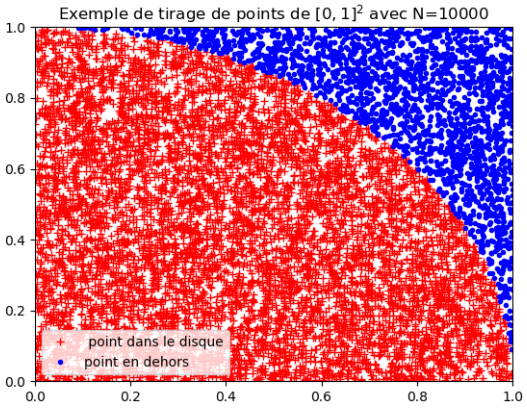
$$p'_n = n\,u'_n \text{ où } u'_n = \tan\frac{\pi}{n},\ u'_6 = \frac{1}{\sqrt{3}} \text{ et } u'_{2n} = \frac{u_n}{\sqrt{1-u_n^2}}$$

1) Programmer le calcul de la suite u_n par une méthode itérative et par méthode récursive. Pour évaluer la vitesse de convergence de cette suite, tracer $\log(\pi - p_n)$ en fonction de n. La valeur de π pourra être prise dans le module numpy ou sympy.

Méthode de Monte-Carlo

La méthode de Monte-Carlo pour approximer π consiste à tirer au hasard des couples de réels $(x, y) \in [0,1]^2$ et de constater s'il appartient (ou non) au quart de disque de rayon 1 de l'ensemble $[0,1]^2$.





La probabilité d'obtenir un point dans le quart de disque est égale au rapport des aires du quart de disque de rayon 1 et du carré de côté 1, autrement dit $\frac{\pi}{4}$. Un théorème de probabilité (*la loi des grands nombres*) stipule que si N est le nombre de points tirés et que A_N le nombre de points tirés appartenant au disque, alors :

$$\frac{A_N}{N} \underset{N\to+\infty}{\longrightarrow} \frac{\pi}{4}$$

2) Tirer au sort un couple (x_i, y_i) de nombres compris entre 0 et 1. Évaluer $r_i = x_i^2 + y_i^2$. Si $r_i \leqslant 1$, alors incrémenter un compteur c. Réitérer le calcul pour un très grand nombre N de couples (x_i, y_i). Tracer $u_n = 4 \times \frac{c}{N}$ en fonction de n.

Convergence d'une série

Série arctan

D'après le DSE de arctan (formule de Gregory), on a $\arctan(x) = \sum_{k=0}^{+\infty} \frac{(-1)^k x^{2k+1}}{2k+1}$ pour tout $x \in [-1, 1]$ [1]. Lorsque l'on calcule la somme partielle d'ordre n, l'erreur commise est de l'ordre de $\frac{|x|^{2n+3}}{2n+3}$.

En particulier pour $x = 1$, on trouve $\frac{\pi}{4} = \sum_{k=0}^{+\infty} \frac{(-1)^k}{2k+1}$ (appelée formule de Leibniz).

3) Programmer le calcul de $S_n = 4 \times \sum_{k=0}^{n} \frac{(-1)^k}{2k+1}$. Pour évaluer la vitesse de convergence de cette suite, tracer $\log(|\pi - S_n|)$ en fonction de n.

4) Cette série convergeant assez lentement (car $x = 1$), on peut - pour accélérer la convergence - utiliser des expressions $\arctan x$ pour des valeurs de x inférieures à 1. Par exemple, la formule de Machin (John) $\frac{\pi}{4} = 4\arctan\frac{1}{4} - \arctan\frac{1}{239}$ a historiquement était très utilisée (car l'une des premières à avoir été découverte) .
Une formule plus efficace (Pourquoi ?) est la formule de Gauss : $\pi = 48\arctan\frac{1}{18} + 32\arctan\frac{1}{57} - 20\arctan\frac{1}{239}$.
Programmer le calcul approché de π par la formule de Gauss (voire de Machin) et évaluer la vitesse de convergence de cette (ou ces) méthode(s).

Série de Ramanujan

Au début du vingtième siècle, S. Ramanujan, mathématicien autodidacte indien, proposa la série :
$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{+\infty} \frac{(4k)!}{(k!)^4} \frac{1103 + 26390\ k}{396^{4k}}$

5) Programmer le calcul de $S'_n = \frac{9801}{2\sqrt{2}} \frac{1}{\sum_{k=0}^{n} \frac{(4k)!}{(k!)^4} \frac{1103 + 26390\ k}{396^{4k}}}$.
Cette suite converge très vite : vérifier qu'au bout de 2 termes, on a déjà π avec 8 décimales. Mais les termes supplémentaires ne permettent de gagner qu'une précision relativement faible.

6) Évaluer l'évolution de la précision $P_n = \pi - S'_n$ en traçant $\log|P_{n+1} - P_n|$ en fonction de n

Solution

```
import math
import decimal #bibliothÃ¨que pour pouvoir faire des calculs prÃ©cis
import sympy
import matplotlib.pyplot as plt
import numpy as np

D = decimal.Decimal
```

1. Remarquez que pour $x = 1$ ou -1, cela provient de la continuité de arctan et de celle de la série de fonctions car il y a convergence uniforme grâce au CSSA

```
decimal.getcontext().prec = 500
c = decimal.Context(prec=500)
PI=sympy.pi.evalf(500) #vraie valeur de pi Ã  500 dÃ©cimales
DPI=D(str(PI)) #pi au format decimal.Decimal


#mÃ©thode par rÃ©currence
def U(n):
    if n==6 :
        return(1/2)
    else:
        return(math.sqrt((1-math.sqrt(1-U(n//2)**2))/2))
def Archimede(n):
    return(n*U(n))
def Erreur(n):
    return(PI-Archimede(n))

nn=range(10)
liste =[]
for i in nn:
    liste+=[math.log10(Erreur(6*2**i))]
plt.plot(nn,liste,'.')
plt.show()

#mÃ©thode itÃ©rative
k=6
u=1/2
nn=range(10)
liste=[]
for _ in nn:
    k=2*k
    u=math.sqrt((1-math.sqrt(1-u**2))/2)
    liste+=[math.log10(PI-k*u)]
plt.plot(nn,liste,'.')
plt.show()



#Monte-Carlo
TirageMax = 10000
inclus = 0
nn=range(1,TirageMax)
liste=[]
for tirage in nn:
    if np.random.rand()**2+np.random.rand()**2 <=1:
        inclus+=1
    liste+=[4*inclus/tirage]
plt.plot(nn,liste)
plt.show()

ListeErreur=np.add(liste,-PI)
plt.plot(nn,ListeErreur)
plt.show()



#Arctan
def SerieArcTan(x,n):
    sum=D(0)
    for k in range(0,n,1):
        sum=sum+(D('-1')**k*x**(2*k+1))/(2*k+1)
    return(sum)

def Erreur(n):
    return abs(PI-4*SerieArcTan(1,n))
```

```
73
74liste=[]
75nn = range(1,500,1)
76for i in nn:
77    liste+=[math.log10(Erreur(i))]
78plt.plot(list(nn),liste)
79plt.show()
80
81
82def Gauss(n): #indispensable de calculer prÃ©cisÃ©ment les fractions
83    return (48*SerieArcTan(D(1)/D(18),n)+32*SerieArcTan(D(1)/D(57),n)-20*SerieArcTan
        (D(1)/D(239),n))
84def Erreur(n):
85    return abs(DPI-Gauss(n))
86
87liste=[]
88nn = range(1,100,1)
89for i in nn:
90    liste+=[math.log10(Erreur(i))]
91plt.plot(list(nn),liste,'.')
92plt.show()
93
94def _factorial(n, m):
95    if (n > m):
96        return _factorial(m, n)
97    elif m == 0:
98        return 1
99    elif n == m:
100        return n
101    else:
102        return _factorial(n, (n+m)//2) * _factorial((n+m)//2 + 1, m)
103
104def factorial(n): #procÃ©dure qui calcule les factorielles de grands nombres en
    entier (et non en float)
105    return _factorial(n, 0)
106
107def S(n):
108    sum=D(0)
109    for k in range(n):
110        num=factorial(4*k)*(1103 + D(26390)*k) #avoir au moins un terme au format D
            pour que num soit D
111        den=factorial(k)**4*D(393)**(4*k) #idem
112        sum=sum+num/den #S au format decimal
113    return(sum)
114def Ramanujan(n):
115    return(D(9801)/(2*D(2).sqrt())*S(n)) #format decimal
116def Erreur(n):
117    return abs(DPI-Ramanujan(n))
118
119nn = range(1,50,1)
120ListeErreur=list(map(Erreur,nn))
121ListePrecision=[0]
122for i in range(48):
123    ListePrecision+=[c.log10(ListeErreur[i+1]-ListeErreur[i])]
124plt.plot(list(nn),np.log10(ListeErreur),'.')
125plt.show()
126plt.plot(list(nn),ListePrecision,'.')
127plt.show()
```