Activité 02 - Approximation de π

Equipe Pédagogique LU1IN0*1

Consignes : Cette activité se compose d'une première partie guidée, suivie de suggestions. Il est conseillé de traiter en entier la partie guidée avant de choisir une ou plusieurs suggestions à explorer.

L'objectif de cette activité est de calculer des valeurs approchées de la constante π de différentes manières.

1 Partie Guidée : Série alternée de Leibniz

La *série alternée de Leibniz pour* π est donnée par l'équation :

$$\frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

En calculant des sommes d'indice de plus en plus élevé, on obtient des approximations de π de plus en plus précises.

Question 1. Ecrire une fonction `terme_leibniz` qui prend en entrée un entier `n` et qui renvoie le terme d'indice `n` de la somme.

```
assert terme_leibniz(0) == 1
assert terme_leibniz(1) == -1 / 3
assert terme_leibniz(10) == 1 / 21
```

Question 2. Ecrire une fonction `somme_leibniz` qui prend en entrée un entier `n` et qui renvoie la somme des `n` premiers termes.

```
assert somme_leibniz(0) == 1
assert somme_leibniz(1) == 1 - 1 / 3
assert somme_leibniz(4) == 1 - 1 / 3 + 1 / 5 - 1 / 7 + 1 / 9
```

Question 3. Ecrire une fonction `approx_leibniz` qui prend en entrée un entier `n` et calcule une approximation de π en utilisant la somme des `n` premiers termes.

```
=== Evaluation de : 'approx_leibniz(10)' ===
3.232315809405594
============================================

=== Evaluation de : 'approx_leibniz(100)' ===
3.1514934010709914
=============================================
```

Question 4. Il est hors de question de tester une fonction renvoyant une constante à virgule flottante avec un test de la forme :

```
assert approx_leibniz(10) == 3.232315809405594
```

En effet, ces égalités sont sujettes aux imprécisions du calcul avec des `float` (en outre, on ne peut "deviner" la valeur de droite qu'en exécutant la fonction). Il est plus raisonnable, dans ce cas, d'écrire des tests avec des **encadrements** de la forme :

```
assert abs(f(v) - res) < 10 ** k
```

où `f(v)` est l'appel de fonction à tester, `res` le résultat formel attendu (ici on utilisera `math.pi`) et `k` la puissance de 10 de la précision attendue (*une précision à* `k` *chiffres apres la virgule*)

Donner un jeu de test pour la fonction `approx_leibniz`. Essayer de trouver le rang (pas forcément de manière précise) à partir duquel l'approximation est précise vis-à-vis de `math.pi` à k chiffres après la virgule, pour des k choisis de manière pertinente.

2 Suggestion : Formule des frères Chudovsky

La formule des frères Chudovsky pour π est donnée par l'équation suivante :

$$\pi = C \left(\sum_{q=0}^{\infty} \frac{M_q . L_q}{X_q} \right)^{-1}$$

avec :

$$M_q = \frac{(6q)!}{(3q)!(q!)^3} \qquad L_q = 545140134q + 13591409 \qquad X_q = (-262537412640768000)^q$$

$$C = 426880.\sqrt{10005}$$

cf. (en anglais) `https://en.wikipedia.org/wiki/Chudnovsky_algorithm`

Implémentation. En écrivant d'abord des fonctions pour chacun des termes M_q, L_q, X_q, écrire une fonction `approx_chudovsky` qui donne l'approximation de π au rang n donné par cette formule. Tester la fonction comme dans la section précédente.

```
=== Evaluation de : 'approx_chudnovsky(1)' ===
3.141592653589793
==============================================
```

3 Suggestion : Optimisation de Chudovsky

Remarque : Il est nécessaire d'avoir traité la Section 2 (Formule des frères Chudovsky) avant de commencer cette section.

Recalculer les termes M_q, L_q et X_q à chaque étape de la boucle qui calcule la somme est superflu, car ces termes peuvent être déduits des termes M_{q-1} , L_{q-1}, X_{q-1} (on pourra s'aider du Web[1]). En les maintenant dans des variables, on peut, à chaque tour de la boucle qui calcule la somme, calculer directement ces termes en fonction de leur valeur au tour de boucle précédent.

Ecrire une fonction `approx_chudnovsky_optim` qui calcule une approximation de π selon la formule des frères Chudnovsky, mais qui implémente l'optimisation décrite ci-dessus.

```
=== Evaluation de : 'approx_chudnovsky_optim(1)' ===
3.141592653589793
==============================================
```

4 Suggestion : Aléatoire

Si on trace un disque de centre $(\frac{1}{2}, \frac{1}{2})$ et de rayon $\frac{1}{2}$ et qu'on prend un point au hasard (uniformément) dans le carré unité (celui dont la diagonale va de $(0, 0)$ à $(0, 1)$) la probabilité que le point se trouve dans le disque est $\frac{\pi}{4}$ (l'aire du carré unité est 1 et l'aire du disque est $\pi.(\frac{1}{2})^2 = \frac{\pi}{4}$).

[1] mais attention, l'expression de M_q sur la page de Wikipédia, semble au moment de la rédaction de ce sujet, fausse, il faut remplacer q par $q + 1$ pour traiter le cas ou q vaut 1

On peut utiliser cette remarque pour approximer π en tirant au hasard des abscisses et des ordonnées entre 0 et 1 et en vérifiant si le point correpondant se trouve dans le disque en question (ce qui se fait facilement en calculant la distance entre le point obtenu et le point $(\frac{1}{2}, \frac{1}{2})$).

Si on tire un grand nombre N de points et qu'on calcule le rapport entre le nombre de ceux qui sont dans le disque et N, on obtient, selon la Loi des Grands Nombres, une approximation de π.

En déduire une fonction `approx_aleatoire` qui utilise cette technique pour calculer une approximation de π.

Rappel: La primitive `random.random() -> float` du module `random` utilise un générateur pseudo-aléatoire pour renvoyer un nombre compris entre 0 et 1.

5 Suggestion : Recherche documentaire

On pourra chercher sur le Web des approximations de constantes ou de fonctions qui utilisent un algorithme itératif (typiquement, une somme de série, ou une fonction récursive) et les implémenter.

6 Remarque

Cette activité mobilise de nombreuses opérations arithmétiques de constantes à virgules flottantes (`float`). *Python* (et, par conséquent, *MrPython*), ne procède pas, par défaut, à un traitement sûr de ces données, ce qui conduit à des erreurs (essayer d'évaluer `10 / 3` ou `0.1 + 0.2`, par exemple). Lorsque l'on veut programmer des opérations (plus) sûres sur ces nombres, on utilise un outil (logiciel ou bibliothèque) adapté, par exemple le module *Python* `decimal`.