Pi

February 15, 2019

1 La millionième décimale de π

Marc Lorenzi

15 janvier 2019

Il fit la cuve en métal fondu. Elle faisait 5 mètres d'un bord à l'autre et était entièrement ronde. Elle faisait 2 mètres et demi de haut et on pouvait mesurer sa circonférence avec un cordon de 15 mètres (Ancien Testament, 1 Rois, 7:23)

Exemple pour faire un champ circulaire de 9 khet. Combien est-ce en surface ? Enlève lui $\frac{1}{9}$, c'est à dire 1. Le reste est 8. Fais la multiplication : 8×8, *cela devient* 64. *Sa quantité c'est, en surface, 64 setat (Papyrus Rhind, problème 50)*

```
In [1]: from sympy import *
        import matplotlib.pyplot as plt
        import math
        import gc
```

1.1 0. Remarques

Nous allons dans ce notebook effectuer des calculs intensifs sur des nombres ayant un grand nombre de décimales. Lors de l'installation du module Python `sympy`, le module `mpmath` a été aussi installé. Ce module permet de calculer en virgule flottante à des précisions quelconques. Une remarque s'impose cependant.

`mpmath` utilise par défaut les entiers longs de Python pour faire ses calculs. Il n'est donc pas extrêmement efficace. Pour ceux d'entre-vous qui voudraient calculer très vite, il convient d'installer `gmpy`. Cette installation se fait en deux temps :

1. Installer la bibliothèque C `gmp` (*GNU MultiPrecision Library*). Plusieurs possibilités :
   - Votre ordinateur tourne sous Windows. Je ne sais pas comment il faut faire.
   - Votre ordinateur tourne sous Linux. Votre gestionnaire de packages installera `gmp` sans difficulté.
   - Votre ordinateur tourne sous MacOS. Vous devez installer un gestionnaire de packages. Personnellement j'utilise MacPorts. Une fois MacPorts installé, ouvrir un terminal et exécuter `sudo port install gmp`.
2. Installer `gmpy`. Ça c'est facile. Ouvrir un terminal et exécuter `pip3 install gmpy`.

Une fois `gmpy` installé, le module `mpmath` s'en aperçoit et utilise la bibliothèque `gmp` pour les calculs en virgule flottante. Le gain en vitesse est énorme, en gros un facteur 100. Conclusion :

Si vous n'installez pas `gmpy`, d'ici la fin de ce notebook vous pourrez calculer π avec 100000 décimales en 3 secondes et avec 1 million de décimales en 3 minutes. Si vous installez `gmpy` vous pourrez calculer en 3 secondes 1 million de décimales de π et en 40 secondes 10 millions de décimales.

1.2 1. Les valeurs antiques de π

Les premières valeurs approchées historiquement attestées du nombre π se trouvent sur des tablettes babyloniennes. Différentes valeurs y sont présentes, en particulier $3 + \frac{1}{8} \simeq 3.125$.

Les Égyptiens, utilisaient une valeur approchée de π intéressante. Le papyrus Rhind, qui date du XVIème siècle avant notre ère, en fournit la preuve (problèmes 41, 50, etc.). Lors de calculs de surfaces ou de volumes, nos amis du Nil utilisaient la valeur $4\left(\frac{8}{9}\right)^2 \simeq 3.16$.

On trouve également dans la Bible un passage où le nombre π apparaît de façon implicite. Il s'agit du premier livre des Rois, 7:23, parlant de la construction du palais de Salomon. On y trouve la valeur approchée $\pi \simeq \frac{15}{5} = 3$. Cela ne veut pas dire que les Hébreux avaient une approximation de π moins précise que les Égyptiens, mais tout simplement que la valeur qu'ils utilisaient n'était pas d'une importance capitale dans le passage en question de l'ancien testament !

Si l'on peut se prêter à sourire de ces approximations, il faut toutefois souligner que les besoins de précision des Égyptiens pour la construction de greniers à blé n'étaient peut-être pas aussi importants que les nôtres. Une approximation à 2×10^{-2} près était sans aucun doute suffisante pour prévoir les capacités de stockage des récoltes. Par ailleurs, il n'est, ni chez les Hébreux ni chez les Égyptiens, fait mention du nombre π en tant que tel. Nous trouvons dans les textes antiques des "recettes" pour calculer l'aire d'un disque, sans que le coefficient multiplicateur y soit clairement identifié comme un nombre remarquable.

D'excellents livres sur l'histoire de π existent, et ce n'est pas le but de ce notebook. Nous allons dans ce qui suit étudier quelques algorithmes liés au calcul de valeurs approchées "modernes" de π.

```
In [2]: pi_babylone = 3 + 1 / 8
        pi_babylone

Out[2]: 3.125

In [3]: pi_egypte = 4 * (8 / 9) ** 2
        pi_egypte

Out[3]: 3.1604938271604937
```

Passons maintenant à un algorithme qui permet de calculer quelques centaines, voire milliers de décimales de π.

1.3 2. La formule de Machin

Euh, c'est quoi ce truc ? John Machin est un mathématicien anglais qui a calculé en 1706 les 100 premières décimales du nombre π. Nous allons voir comment, et calculer 1000 décimales de π avec sa formule.

1.3.1 2.1 Approximons la fonction arc tangente

Je m'excuse à l'avance des quelques préliminaires mathématiques qui vont suivre. Si vous ne voyez pas le rapport entre arctan et π, soyez patients.

Soit n un entier non nul. Soit $x \in]0,1]$.On a

$$\int_0^x \sum_{k=0}^{n-1} (-1)^k t^{2k} dt = \sum_{k=0}^{n-1} (-1)^k \frac{x^{2k+1}}{2k+1}$$

Par ailleurs, pour tout $t \in [0,x]$,

$$\sum_{k=0}^{n-1} (-1)^k t^{2k} = \frac{1-(-1)^n t^{2n}}{1+t^2}$$

Ainsi,

$$\sum_{k=0}^{n-1} (-1)^k \frac{x^{2k+1}}{2k+1} = \int_0^x \frac{dt}{1+t^2} - (-1)^n \int_0^x \frac{t^{2n} dt}{1+t^2} = \arctan x - (-1)^n \int_0^x \frac{t^{2n} dt}{1+t^2}$$

L'intégrale du membre de droite est facile à majorer, en remarquant que le dénominateur de la fraction est au moins égal à 1 :

$$\left| \int_0^x \frac{t^{2n} dt}{1+t^2} \right| \leq \int_0^x t^{2n} dt = \frac{x^{2n+1}}{2n+1}$$

On en déduit la
Proposition :

$$\sum_{k=0}^{n-1} (-1)^k \frac{x^{2k+1}}{2k+1} = \arctan x + \varepsilon_n$$

où

$$|\varepsilon_n| \leq \frac{x^{2n+1}}{2n+1}$$

1.3.2 2.2 Qu'obtient-on pour $x = 1$?

En prenant $x = 1$ dans la proposition ci-dessus, on obtient

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

où

$$|\varepsilon_n| \leq \frac{1}{2n+1}$$

Pour obtenir 10 décimales de π il faudrait (suffirait, pour être exact) donc additionner n termes, où n vérifie $\frac{4}{2n+1} \leq 10^{-10}$, c'est à dire environs 20 milliards de termes. Contentons-nous d'une approximation à 10^{-5} près en ajoutant 200000 termes.

```
In [6]: def approx_mauvaise(n):
            s = 0
            for k in range(n):
                s = s + (-1) ** k / (2 * k + 1)
            return 4 * s

In [9]: approx_mauvaise(200000)

Out[9]: 3.1415876535897618

In [10]: approx_mauvaise(200001)

Out[10]: 3.141597653564762
```

1.3.3 2.3 Passons à des choses plus sérieuses

Dorénavant, prenons $0 < x < 1$.

Soit $p \in \mathbb{N}$. Combien suffit-il d'additionner de termes dans la formule vue plus haut pour obtenir une approximation de arctan x à 10^{-p} près ? Il nous suffit évidemment de choisir n tel que $\frac{x^{2n+1}}{2n+1} \leq 10^{-p}$, ce qui s'écrit encore

$$(2n+1)\log\frac{1}{x} + \log(2n+1) \geq p$$

où log désigne le logarithme en base 10.

Cette inégalité ne peut pas être résolue de façon exacte. Une majoration plus crue nous dit qu'il suffit de choisir n tel que

$$(2n+1)\log\frac{1}{x} \geq p$$

ou encore

$$n \geq \frac{1}{2}\left(\frac{p}{\log\frac{1}{x}} - 1\right)$$

Par exemple, pour $p = 1000$ et $x = \frac{1}{5}$, cela donne

```
In [11]: (1000 / math.log(5, 10) - 1) / 2

Out[11]: 714.8382790366966
```

Le facteur $2n+1$ est-il si important que cela ? Écrivons une fonction qui renvoie le plus petit entier n tel que $(2n+1)\log\frac{1}{x} + \log(2n+1) \geq p$.

```
In [12]: def erreur(x, p):
             n = 1
             while (2 * n + 1) * math.log(1 / x, 10) + math.log(2 * n + 1, 10) <= p + 1: n = n +
             return n

In [13]: erreur(1 / 5, 1000)
```

```
Out[13]: 714
```

Moralité : le facteur $2n+1$ n'est pas important du tout :-).

Bon, au travail ! La fonction `approx_atan` prend en paramètres un flottant x et un entier p. Elle renvoie une approximation de $\arctan x$ à 10^{-p} près.

```
In [25]: def approx_atan(x, p):
             x = N(x, p + 10)
             n = erreur(x, p)
             s = 0
             for k in range(n):
                 s = s + (-1) ** k * x ** (2 * k + 1) / (2 * k + 1)
             return s
```

Testons $\arctan \frac{1}{5}$ avec 1000 chiffres après la virgule.

```
In [26]: x15 = S(1) / 5
```

```
In [27]: approx_atan(x15, 1000)
```

```
Out[27]: 0.19739555984988075837004976519479029344758510378785210151768894024103396997824378573260
```

Et vérifions avec `sympy`!

```
In [28]: atan(x15).evalf(1000) - approx_atan(x15, 1000)
```

```
Out[28]: -1.43921165047991985943461800361191388290705777252891935301980596579059054758245676739
```

Tout va bien. Passons à la formule de Machin.

Proposition (Machin) :

$$\frac{\pi}{4} = 4\arctan\frac{1}{5} - \arctan\frac{1}{239}$$

Démonstration : Voici tout d'abord un exercice laissé au lecteur : pour $a, b \in \mathbb{R}_+$ tels que $ab < 1$, on a

$$\arctan a + \arctan b = \arctan\frac{a+b}{1-ab}$$

On en déduit que $2\arctan\frac{1}{5} = \arctan\frac{\frac{2}{5}}{1-\frac{2}{5}\frac{2}{5}} = \arctan\frac{5}{12}$.

De là, $4\arctan\frac{1}{5} = 2\arctan\frac{5}{12} = \arctan\frac{\frac{10}{12}}{1-\frac{25}{144}}$.

Au secours, Python !

```
In [29]: Rational(10, 12) / (1 - Rational(25, 144))
```

```
Out[29]: 120/119
```

Bon, super. $4\arctan\frac{1}{5} = \arctan\frac{120}{119}$. Encore un petit effort !

$\arctan\frac{120}{119} - \arctan\frac{1}{239} = \arctan\frac{\frac{120}{119}-\frac{1}{239}}{1+\frac{120}{119}\frac{1}{239}}$.

Encore une fois, Python est notre ami.

```
In [30]: x = Rational(120, 119)
         y = Rational(1, 239)
         (x - y) / (1  + x * y)
```

```
Out[30]: 1
```

Bref, $4 \arctan \frac{1}{5} - \arctan \frac{1}{239} = \arctan 1 = \frac{\pi}{4}$.

Cette formule nous suggère une méthode pour calculer une approximation de π. Combien doit-on additionner de termes dans notre formule fétiche ? Majorons grossièrement (pas trop quand même) l'erreur ε_n commise en additionnant n termes pour approcher $\arctan \frac{1}{5}$ et $\arctan \frac{1}{239}$, puis en multipliant par 4 :

$$\varepsilon_n \leq \frac{4}{2n+1} \left(\frac{4}{5^{2n+1}} + \frac{1}{239^{2n+1}} \right) \leq \frac{4}{2n+1} \frac{1}{5^{2n}}$$

car $239 > 5$ et $4 + 1 = 5$. Pour obtenir une approximation de π à 10^{-p} près, il suffit donc de choisir n tel que

$$\frac{4}{2n+1} \frac{1}{5^{2n}} \leq 10^{-p}$$

ou encore

$$5^{2n} \geq \frac{4 \times 10^p}{2n+1}$$

Comme nous avons compris que le facteur $2n + 1$ ne change pas la face du monde, il suffit donc de choisir n tel que

$$5^{2n} \geq 4 \times 10^p$$

ou encore

$$n \geq \frac{p + \log 4}{2 \log 5}$$

Voulons-nous 1000 décimales de π ?

```
In [31]: (1000 + math.log(4, 10)) / (2 * math.log(5, 10))
```

```
Out[31]: 715.76895559477
```

Voici le Machin.

```
In [33]: def machin(p):
             x15 = N(Rational(1, 5), p + 10)
             x1239 = N(Rational(1, 239), p + 10)
             return 4 * (4 * approx_atan(x15, p) - approx_atan(x1239, p))
```

```
In [34]: machin(1000)
```

```
Out[34]: 3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803
```

Vérifions.

```
In [35]: pi.evalf(1000) - machin(1000)

Out[35]: 8.91886263073848153730998425265489630256243953388702640395558936827633227920579333860870
```

On peut lancer `machin(2000)` ou `machin(3000)` mais n'espérons pas obtenir un million de chiffres de π par cette méthode !

Remarque : John Machin n'avait pas d'ordinateur. Il a donc calculé 100 décimales de π **à la main**.

Le record actuel du nombre de décimales calculées de π est 22 trillions ... Comment atteindre une telle précision en un temps raisonnable ? Nous allons examiner deux algorithmes permettant de calculer extrêmement rapidement des valeurs approchées de π.

L'intérêt d'avoir deux algorithmes totalement distincts est de pouvoir **COMPARER** les résultats renvoyés, et donc de **CONFIRMER** la valeur obtenue. Nous allons commencer par l'algorithme de Salamin et Brent.

1.4 3. L'algorithme de Salamin-Brent

On va construire dans ce qui suit deux suites récurrentes. Une certaine quantité définie à partir de ces suites converge très rapidement vers π. Nous allons voir précisément ce qu'il faut entendre par "rapidement".

1.4.1 3.1 Suites récurrentes

On définit les suites (a_n) et (b_n) par récurrence :

- Tout d'abord, $a_0 = 1$, $b_0 = \frac{1}{\sqrt{2}}$.
- Pour tout $n \geq 0$, $a_{n+1} = \frac{a_n + b_n}{2}$, $b_{n+1} = \sqrt{a_n b_n}$.

Les suites (a_n) et (b_n) sont donc formées en calculant des moyennes arithmétiques et des moyennes géométriques. Elles convergent *très rapidement* vers ce que l'on appelle la moyenne **arithmético-géométrique** de 1 et $\frac{1}{\sqrt{2}}$, que nous noterons ci-dessous $AGM(1, \frac{1}{\sqrt{2}})$.

Définissons ensuite $c_n = a_n^2 - b_n^2$ puis

$$\pi_n = \frac{2a_n^2}{1 - 2\sum_{k=0}^{n} 2^k c_k^2}$$

Proposition :

$$0 \leq \pi - \pi_n \leq \frac{\pi^2 2^{n+4} e^{-\pi 2^{n+1}}}{AGM(1, \frac{\sqrt{2}}{2})^2}$$

Démonstration : J'admets cette proposition. La démonstration en est abordable mais un peu longue.

1.4.2 3.2 L'erreur commise

Nous tenons clairement là un algorithme intéressant pour calculer des approximations de π. Le majorant de l'erreur tend très rapidement vers 0, ce qui nous permet d'espérer de très bonnes approximations en très peu d'itérations. Combien d'itérations, au juste ? Regardons cela.

Voici tout d'abord une fonction `agm` qui calcule la moyenne arithmético-géométrique de deux réels a et b à 10^{-10} près. Inutile d'être précis, nous voulons juste majorer $\pi - \pi_n$.

```
In [36]: def agm(a, b):
             u = a
             v = b
             while abs(v - u) > 1e-10:
                 u, v = (u + v) / 2, math.sqrt(u * v)
             return u
```

```
In [37]: agm(1, 1 / math.sqrt(2))
```

```
Out[37]: 0.8472130848351929
```

Combien de termes sont nécessaires pour approcher π à 10^{-p} près ?

```
In [38]: def nb_termes(p):
             n = 0
             z = math.log(math.pi ** 2 / (agm(1, math.sqrt(2) / 2) - 1e-10), 10)
             while math.pi * 2 ** (n + 1) * math.log(math.e, 10) - (n + 4) * math.log(2, 10) < p
                 n = n + 1
             return n
```

100000 chiffres de π ?

```
In [39]: nb_termes(10 ** 5)
```

```
Out[39]: 16
```

16 itérations suffiront. Et 1 million de chiffres ?

```
In [40]: nb_termes(10 ** 6)
```

```
Out[40]: 19
```

Il suffit de faire **TROIS** itérations de plus. Et 22 trillions de décimales ?

```
In [41]: nb_termes(22 * 10 ** 12)
```

```
Out[41]: 43
```

Eh oui, 22 trillions de décimales en 43 itérations ...

1.4.3 3.3 La boucle magique

La fonction `salamin` est évidente. Elle calcule une approximation de π à 10^{-p} près. La fonction prend également un paramètre opitonnel n. Si n vaut `None`, la fonction appelle `nb_termes` pour décider du nombre d'itérations à effectuer. Sinon, elle effectue n itérations.

Le paramètre optionnel `dbg` affiche en particulier le numéro de l'itération en cours. Cela permet de s'assurer que les calculs avancent, et fait patienter celui qui est assis devant son ordinateur.

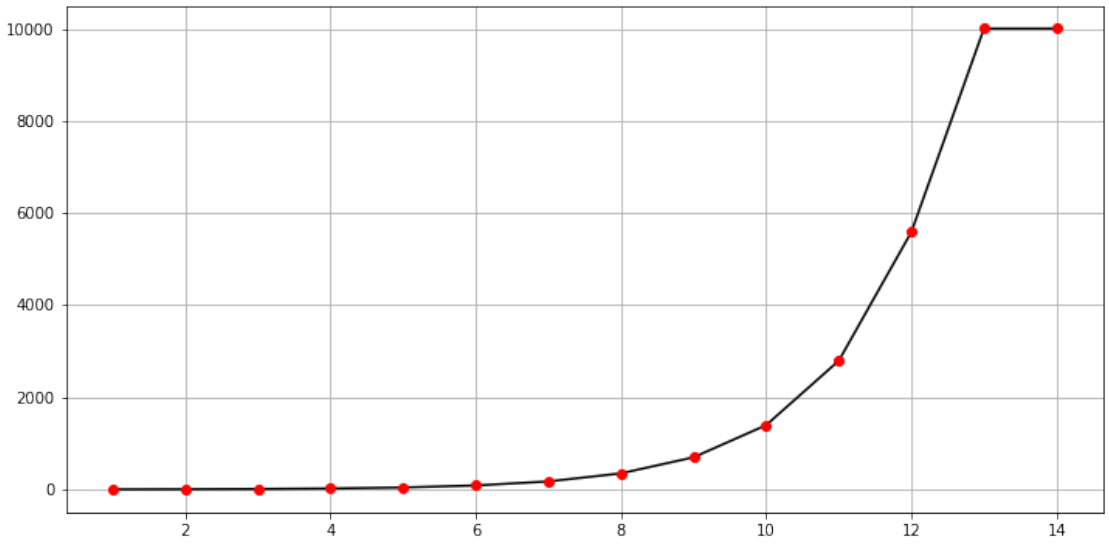
Remarquez le $p + 10$ dans les approximations numériques. On prévoit 10 chiffres significatifs supplémentaires dans les calculs en espérant que cela suffira pour compenser les erreurs d'arrondi.

```
In [42]: def salamin(p, n=None, dbg=False):
             if n == None:
                 n = nb_termes(p) + 1
             if dbg: print('Nombre itérations :', n)
             a = N(1, p + 10)
             b = 1 / N(sqrt(2), p + 10)
             s = N(1, p + 10) / 2
             for k in range(1, n + 1):
                 if dbg: print(k)
                 a, b = (a + b) / 2, sqrt(a * b)
                 c = a * a - b * b
                 s = s - 2 ** k * c
                 #gc.collect()
             return N(2 * a ** 2 / s, p + 1)
```

Voici un petit graphique montrant le nombre de décimales obtenues en fonction du nombre d'itérations. Les calculs sont faits avec 10000 chiffres après la virgule. En gros le nombre de décimales exactes **double** à chaque itération : l'algorithme de Salamin-Brent a une vitesse de convergence **quadratique**.

```
In [48]: plt.rcParams['figure.figsize'] = (12, 6)
```

```
In [60]: s = []
         p = 10000
         Npi = N(pi, p + 10)
         ks = list(range(1, 15))
         for k in ks:
             s.append(salamin(p, k) - Npi)
         s = [-log(abs(x), 10) for x in s]
         plt.plot(ks, s, 'k')
         plt.plot(ks, s, 'or')
         plt.grid()
```



Nous y sommes : calculons π avec 10^5 chiffres après la virgule. Si vous avez installé `gmpy` vous pouvez mettre sans risque 10^6 .

```
In [53]: p = 10 ** 5
         x = salamin(p, None, True)
```

```
Nombre itérations : 17
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
```

Écrivons la valeur obtenue dans un fichier !

```
In [54]: f = open('pi_salamin_' + str(p) + '.txt', 'w')
         f.write(str(x))
         f.close()
```

Bonne lecture :-).

Sur ma machine, j'obtiens 10^6 décimales en environ trois secondes. Pour 10^7 décimales, une quarantaine de secondes suffisent. Et pour 10^8 (cent millions de décimales), une quinzaine de minutes s'imposent. Le fichier obtenu fait alors 100 Méga-Octets et mon éditeur de textes ne peut pas l'ouvrir. En plus ma machine utilise pour le calcul toute la mémoire disponible (4 Giga-Octets). Inutile donc de tenter le milliard de décimales :-).

1.4.4 3.4 On a fini, alors ?

Euh, pas tout à fait. Avez-vous lu le fichier `salamin_100000.txt` ? Avez-vous bien vérifié chaque décimale de π ? N'y a-t-il pas d'erreur ? Nous pourrions évidemment demander à `sympy` une évaluation et comparer avec notre résultat. Mais que faire lorsqu'on tente de battre un record est qu'on est le premier à calculer N décimales de π ? La solution qui s'impose est de refaire le calcul avec un autre algorithme puis de comparer les deux résultats. C'est ce que nous allons faire.

1.5 4. L'algorithme de Borwein

L'algorithme de Borwein et Borwein (Jonathan et Peter) est un algorithme **quartique**, c'est à dire que le nombre de décimales calculées **quadruple** à chaque itération. Il fonctionne comme suit :

On pose $a_0 = 6 - 4\sqrt{2}$ et $y_0 = \sqrt{2} - 1$.

Puis, pour tout entier $k \geq 0$, on pose

$$y_{k+1} = \frac{1 - (1 - y_k^4)^{1/4}}{1 + (1 - y_k^4)^{1/4}}$$

et

$$a_{k+1} = a_k(1 + y_{k+1})^4 - 2^{2k+3} y_{k+1}(1 + y_{k+1} + y_{k+1}^2)$$

Alors $\frac{1}{a_k}$ converge quartiquement vers π.

La fonction `borwein` ci-dessous calcule p décimales de π. N'ayant pas réussi à trouver dans la littérature un majorant explicite de l'erreur, je lui fais effectuer une sortie de boucle lorsque deux valeurs successives a_k et a_{k+1} sont égales à 10^{-p} près. Remarquez, encore une fois, les 10 décimales de "sécurité" pour éviter les erreurs d'arrondi.

```
In [55]: def borwein(p, dbg=False):
             a = N(6 - 4 * sqrt(2), p + 10)
             y = N(sqrt(2) - 1, p + 10)
             k = 0
             while True:
                 if dbg: print(k)
                 z = sqrt(sqrt(1 - y ** 4))
                 y = (1 - z) / (1 + z)
                 a1 = a * (1 + y) ** 4 - 2 ** (2 * k + 3) * y * (1 + y + y * y)
                 if N(a1, p) == N(a, p): break
                 a = a1
                 k = k + 1
                 #gc.collect()
             return N(1 / a, p + 1)
```

```
In [56]: p = 10 ** 5
         x = borwein(p, True)

0
1
2
3
4
5
6
7
8
```

```
In [57]: f = open('pi_borwein_' + str(p) + '.txt', 'w')
         f.write(str(x))
         f.close()
```

Bon, et maintenant ? Nous disposons de deux fichiers, `pi_salamin_100000.txt` et `pi_borwein_100000.txt`. Ces deux fichiers devraient être identiques.

Si votre ordinateur tourne sous Linux ou MacOS, exécutez la cellule ci-dessous.

```
In [58]: !diff -s pi_borwein_100000.txt pi_salamin_100000.txt

Files pi_borwein_100000.txt and pi_salamin_100000.txt are identical
```

Cela nous remplit d'une joie sans bornes :-).

Si vous êtes sous Windows, essayez la commande `FC` (je ne l'ai pas testée).

1.6 5. Peut-on faire mieux ?

Il existe des algorithmes, toujours basés sur des récurrences, dont la convergence est cubique, quartique (algorithme de Borwein), ..., nonique. Nonique, cela veut dire qu'à chaque itération le nombre de décimales exactes est multiplié par 9. Je ne les présenterai pas ici, mais une petite recherche sur Internet vous en donnera une idée.

Apparemment, la formule tendance pour obtenir les derniers records est la formule de Chudnovsky :

$$\frac{1}{\pi} = 12 \sum_{k=0}^{\infty} \frac{(-1)^k (6k)!(13591409 + 545140134k)}{(3k)!(k!)^3 640320^{3k+3/2}}$$

Chaque terme de la formule permet d'obtenir 14 chiffres corrects supplémentaires. De plus, utilisée de façon extrêmement rusée, cette formule permet d'utiliser un résultat obtenu à une certaine précision pour obtenir un résultat à une meilleure précision.

Depuis quelques années, les records dans le calcul des décimales de π sont obtenus par des particuliers sur des PC survitaminés. Le record actuel est, à ma connaissance, de 22.459.157.718.361 décimales (22 trillions). Le calcul a demandé 105 jours. Pourquoi ce nombre précis de décimales ? Regardez ci-dessous :-).

```
In [59]: math.pi ** math.e

Out[59]: 22.45915771836104

In [ ]:
```