Approximations de π

1 Méthode d'Archimède

1.1 Le principe

La méthode d'Archimède de calcul de π (historiquement la première que nous connaissions) est basée sur la remarque suivante : considérons un polygone régulier à n sommets inscrit dans un cercle $\mathcal{C}$. Plus n augmente, plus la forme du polygone se rapproche de celle du cercle $\mathcal{C}$.

Si on appelle P_n le périmètre du polygone régulier à n côtés, on peut supposer (par observation) que (P_n) est croissante et bornée. On dira alors par définition que sa limite est la longueur du cercle $\mathcal{C}$.

Partant de cette idée, Archimède a considéré la suite obtenue à partir d'un demi-hexagone régulier en multipliant le nombre de côtés par deux à chaque étape, autrement dit, la suite des polygones à 3×2^n côtés.

Remarque : En fait, la méthode d'Archimède répond à deux questions :

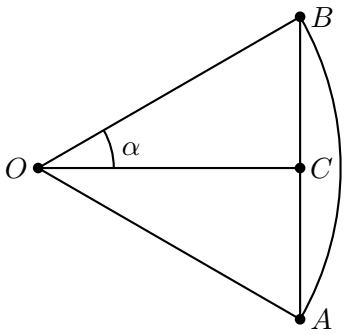
- L'une, pratique : déterminer par le calcul une valeur approchée de π.
- L'autre, théorique : définir mathématiquement la longueur d'un arc.

En effet, définir la longueur d'un arc n'est pas si simple. La première définition qui vient à l'esprit est expérimentale : par exemple, on entoure un objet circulaire avec une ficelle, puis on tend la ficelle pour mesurer sa longueur. Cette définition est insatisfaisante, car la précision de la mesure est limitée par de nombreux facteurs, l'élasticité de la ficelle, le caractère plus ou moins circulaire de l'objet utilisé, etc.

La méthode d'Archimède ne fait appel à aucune expérience physique.

Supposons que le cercle soit de rayon 1. Le dessin ci-contre suggère que la longueur de l'arc $\overset{\frown}{AB}$ est proche de la longueur du segment (appelé corde) $[AB]$ lorsque l'angle $\widehat{AOB}$ est petit. Notons 2α la longueur de $\overset{\frown}{AB}$; comme $\mathcal{C}$ est de rayon 1, 2α est aussi une mesure en radians de $\widehat{AOB}$. Ainsi, $\overset{\frown}{AB} \simeq AB$ se traduit par : $2\sin(\alpha) \simeq 2\alpha$, soit : $\sin(\alpha) \simeq \alpha$.

On peut traduire cette approximation en termes de limite : $\lim\limits_{\alpha \to 0} \dfrac{\sin(\alpha)}{\alpha} = 1$



C'est une propriété fondamentale, mais dont Archimède se sert comme définition : Pour mesurer la longueur d'un arc, on le partage en un très grand nombre n de parties égales, on approche chaque partie par la corde correspondante et on multiplie par n.

Application : Supposons qu'on ait partagé le demi-cercle en 2^n parties. On obtient en posant $\alpha = \dfrac{\pi}{2^n}$:

$$\lim_{n \to +\infty} 2^n \sin\left(\frac{\pi}{2^n}\right) = \pi. \tag{1}$$

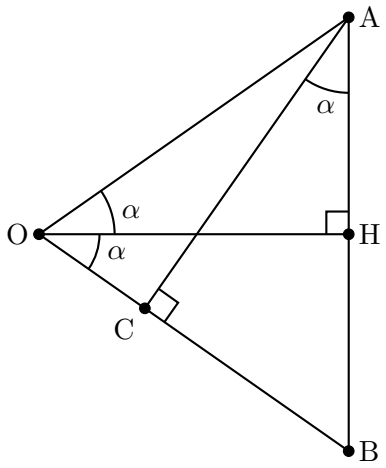
Si on sait évaluer $\sin\left(\dfrac{\pi}{2^n}\right)$, on obtient alors une approximation de π.

1.2 Recherche d'une relation de récurrence

Bien. Mais il faut aussi pouvoir passer de $\sin\left(\dfrac{\pi}{2^n}\right)$ à $\sin\left(\dfrac{\pi}{2^{n+1}}\right)$.

Dans le triangle ci-dessous, on a : OA = OB = 1, $\widehat{\text{AOH}} = \widehat{\text{HOB}} = \alpha$. Le point C est le projeté orthogonal de A sur (OB).

On veut exprimer $\cos(2\alpha)$ en fonction de $\cos(\alpha)$ et $\sin(\alpha)$.



1. Démontrer que $\widehat{\text{CAB}} = \alpha$.
2. Démontrer que $\text{AB} = 2\sin(\alpha)$.
3. Démontrer que $\text{CB} = 2\sin(\alpha) \times \sin(\alpha)$.
4. Démontrer que $\text{OC} = \cos(2\alpha)$.
5. En déduire que pour $\alpha \in \left[0\,;\frac{\pi}{4}\right]$, $\cos(2\alpha) = 1 - 2\sin^2(\alpha)$.
6. On rappelle que pour tout α : $\cos^2(\alpha) + \sin^2(\alpha) = 1$

   En déduire que pour $\alpha \in \left[0\,;\frac{\pi}{4}\right]$: $\sin^2(\alpha) = \dfrac{1 - \sqrt{1 - \sin^2(2\alpha)}}{2}$.

Appliquons cette formule avec $\alpha = \dfrac{\pi}{2^{n+1}}$.

On peut alors écrire $2\alpha = 2 \times \dfrac{\pi}{2^{n+1}} = \dfrac{2 \times \pi}{2 \times 2^n} = \dfrac{\pi}{2^n}$.

on obtient :

$$\sin^2\left(\frac{\pi}{2^{n+1}}\right) = \frac{1 - \sqrt{1 - \sin^2\left(\dfrac{\pi}{2^n}\right)}}{2}. \tag{2}$$

et il faut prendre $0 \leq 2\alpha \leq \dfrac{\pi}{2}$ donc $n \geq 1$.

Posons pour $n \in \mathbb{N}^*$: $u_n = \sin^2\left(\dfrac{\pi}{2^n}\right)$. Comme $0 \leq \dfrac{\pi}{2^n} \leq \dfrac{\pi}{2}$, $\sin\left(\dfrac{\pi}{2^n}\right)$ est positif, donc $\sin\left(\dfrac{\pi}{2^n}\right) = \sqrt{u_n}$.

D'après l'égalité (2), $u_{n+1} = \dfrac{1}{2}\left(1 - \sqrt{1 - u_n}\right)$; d'autre part, $u_1 = \sin^2\left(\dfrac{\pi}{2}\right) = 1$.

Posons ensuite : $p_n = 2^n \sin\left(\dfrac{\pi}{2^n}\right) = 2^n \times \sqrt{u_n}$.

D'après l'équation (1), on a : $\lim\limits_{n\to+\infty} p_n = \pi$.

1.3 Calcul

On dispose donc de deux suites (u_n) et (p_n) définies ainsi :

$$\begin{cases} u_1 = 1 \\ u_{n+1} = \dfrac{1}{2}(1 - \sqrt{1 - u_n}) \end{cases} \quad \text{et} \quad p_n = 2^n\sqrt{u_n}.$$

On obtient donc un algorithme ressemblant à ceci :

```
u <- 1
Pour n de 2 à n_max:
      u <- (1-racine_carrée(1-u))/2
      p <- 2^n*sqrt(u)
      Afficher p
```

Programmer cet algorithme et vérifier...

Solution possible :

```
from math import sqrt

def pi_archimede_1(n_max):
    u=1
    for n in range(2,n_max+1):
        u=(1-sqrt(1-u))/2
        p=2**n*sqrt(u)
        print(n, '--->', p)
```

Tapons par exemple : `pi_archimede_1(30)`

On obtient :

2 —> 2.8284271247461903

3 —> 3.061467458920718

4 —> 3.121445152258053

5 —> 3.1365484905459406

6 —> 3.140331156954739

7 —> 3.141277250932757

8 —> 3.1415138011441455

9 —> 3.1415729403678827

10 —> 3.141587725279961

11 —> 3.141591421504635

12 —> 3.141592345611077

13 —> 3.1415925765450043

14 —> 3.1415926334632482

15 —> 3.141592654807589

16 —> 3.1415926453212153

17 —> 3.1415926073757197

18 —> 3.1415929109396727

19 —> 3.141594125195191

20 —> 3.1415965537048196

21 —> 3.1415965537048196

22 —> 3.1416742650217575

23 —> 3.1418296818892015

24 —> 3.142451272494134

25 —> 3.142451272494134

26 —> 3.1622776601683795

27 —> 3.1622776601683795

28 —> 3.4641016151377544

29 —> 4.0

30 —> 0.0

Manifestement, il y a un problème.Par comparaison avec la valeur approchée de π donnée par une calculatrice (ou par python lui-même, dans le module `math`), on voit que tout se passe bien jusqu'à $n = 15$, puis le système semble se dérégler.

Voilà ce qui se passe : u_n tend vers 0. Quand un ordinateur ou une calculatrice calcule $1 - u_n$, le résultat va être rapidement arrondi à 1. Ainsi la quantité $1 - \sqrt{1 - u_n}$ va être arrondie à 0. Avec les calculatrices, ça se produit dès que u_n est inférieur à 10^{-10} ou 10^{-11}. Avec python, dès que u_n est inférieur à 10^{-16}.

Pour pallier ce défaut, on peut transformer l'écriture de u_{n+1} en utilisant la « quantité conjuguée » :

$$u_{n+1} = \frac{1 - \sqrt{1 - u_n}}{2} = \frac{\left(1 - \sqrt{1 - u_n}\right)\left(1 + \sqrt{1 - u_n}\right)}{2\left(1 + \sqrt{1 - u_n}\right)} = \frac{1^2 - \sqrt{1 - u_n}^2}{2\left(1 + \sqrt{1 - u_n}\right)} = \frac{1 - (1 - u_n)}{2\left(1 + \sqrt{1 - u_n}\right)}$$

$$u_{n+1} = \frac{u_n}{2\left(1 + \sqrt{1 - u_n}\right)}.$$

On obtient donc un algorithme modifié :

```
u <- 1
Pour n de 2 à n_max:
      u <- u/(2*(1+racine_carrée(1-u)))
      p <- 2^n*sqrt(u)
Afficher p
```

Programmer cet algorithme et vérifier...

La méthode d'Archimède marche plutôt bien. L'inconvénient est qu'elle repose sur beaucoup de calculs de racines carrées. Pour obtenir un grand nombre de décimales, le temps de calcul va sans doute augmenter nettement, mais surtout, on va être confronté rapidement aux erreurs d'arrondi.

On voit que les nombres python de type `float` comportent 16 décimales (en fait, 17 chiffres significatifs). Ils représentent les nombres à virgule flottante en double précision, tels que manipulés directement par la machine.

Pour plus de précisions :

https://docs.python.org/fr/3/reference/datamodel.html?highlight=type%20float

https://fr.wikipedia.org/wiki/IEEE_754

Et aussi : https://docs.python.org/fr/3.7/tutorial/floatingpoint.html

Pour avoir une valeur plus précise, on peut utiliser une bibliothèque de python : le module `decimal`.

https://docs.python.org/fr/3/library/decimal.html

L'intérêt de ce module est qu'il permet de faire des calculs exacts sur les décimaux à une précision donnée, et que cette précision (le nombre de décimales) peut être choisie arbitrairement grande (limitée simplement par la mémoire disponible).

Le décimal 1,2 (par exemple) sera entré sous la forme : `Decimal('1.2')` (bien noter les `' '`).

Pour les nombres entiers, il n'y a pas de différence : `Decimal('1.2')+1` et `Decimal('1.2')+Decimal('1')` donnent le même résultat.

En revanche, `Decimal('1.2')+3.3` provoque une erreur ; il faut taper `Decimal('1.2')+Decimal('3.3')`.

Pour la racine carrée, la syntaxe change également, il faut taper par exemple : `Decimal('1.2').sqrt()`.

```
from decimal import *

getcontext().prec = 52 # précision à 50 décimales

def pi_archimede_3(n_max):
    u=Decimal('1')
    for n in range(2,n_max+1):
        u=u/(2*(1+(1-u).sqrt()))
        p=Decimal('2')**n*u.sqrt()
        print(n, ' ---> ', p)
```

On teste, par exemple avec :`pi_archimede_3(50)`. Ça fonctionne.

Pour vérifier la valeur obtenue, on peut la comparer à celle donnée ici (en gardant les 50 premières décimales) : http://www.jlsigrist.com/pi.html

Il suffit d'ajouter à l'algorithme la ligne :

`Pi=Decimal('3.14159265358979323846264338327950288419716939937510')`

et de modifier la dernière ligne en : `print(n, ' ---> ', p, ' erreur commise : ', p-Pi)`

En tapant `pi_archimede_3(100)`, on obtient :

2 —> 2.828427124746190097603377448419396157139343750753896 erreur commise : -0.313...

3 —> 3.061467458920718173827679872243190934090756499885017 erreur commise : -0.080...

4 —> 3.121445152258052285572557895632355854843065884031277 erreur commise : -0.020...

—

99 —> 3.141592653589793238462643383279502884197169399375107 erreur commise : 7E-51

100 —> 3.141592653589793238462643383279502884197169399375106 erreur commise : 6E-51

L'erreur commise est inférieure à 10^{-50} ; on a donc bien obtenu 50 décimales exactes...