Introduction à l'algorithmique avec Python

PROT Olivier

13 novembre 2009

Table des matières

1 Introduction à Python **1**
1.1 Syntaxe . . . 1
1.1.1 Exécution d'un programme Python . . . 1
1.1.2 Variables, opérations et typage . . . 2
1.1.3 Fonctions . . . 2
1.1.4 Boucles FOR et WHILE . . . 3
1.1.5 Instructions conditionnelles : IF - ELIF - ELSE . . . 3
1.1.6 Illustration - Calcul d'une moyenne . . . 4

2 Algorithmes pour le calcul de π **4**
2.1 Méthode d'Archimède . . . 4
2.1.1 Étude de l'algorithme . . . 5
2.1.2 Algorithme corrigé . . . 6
2.1.3 Erreurs numériques . . . 7
2.1.4 La méthode de Viète . . . 8
2.1.5 Le vrai algorithme d'Archimède . . . 8
2.2 Méthode des fléchettes (Monte-Carlo) . . . 8
2.2.1 Construction de l'algorithme . . . 9
2.2.2 Étude de la convergence au sens des probabilités . . . 10

1 Introduction à Python

Python (`www.python.org`) est un langage de programmation moderne, favorisant la programmation structurée (à l'aide de fonctions) et *orienté-objet*. C'est également un logiciel libre pouvant s'installer sur la plupart des ordinateurs.

Python est ce que l'on appelle un *langage interprété*, cela signifie qu'un programme est traduit en langage-machine au moment de son exécution. Lorsque l'on ouvre une session Python, chaque ligne de programme saisie est immédiatement traduite et exécutée. On peut bien sûr, sauver le programme dans un fichier afin de l'exécuter ultérieurement, mais à chaque appel le programme est toujours traduit avant d'être exécuté. Ceci est en opposition avec les *langages compilés* (par exemple `FORTRAN`, `C` ou `C++`) où le programme est traduit en langage machine une fois pour toutes.

Enfin, Python est ce que l'on appelle un langage *haut-niveau*, il permet de faire simplement des opérations complexes et cela implique que son fonctionnement soit assez éloigné de celui de la machine. Au contraire, un langage *bas-niveau* (par exemple le C) est un langage dont les opérations sont assez proches de ce que fait la machine et qui par conséquent ne permet pas de faire des opérations complexes de manière simple. L'avantage de l'utilisation d'un langage haut-niveau est qu'il permet d'écrire des programmes plus rapidement en évitant les erreurs.

1.1 Syntaxe

1.1.1 Exécution d'un programme Python

Comme signalé plus haut, il est possible d'exécuter un programme Python en tapant les lignes de code une à une dans l'interpréteur. Cela est fastidieux car pour ré-exécuter le programme il faut re-taper tout le code en entier

à chaque fois. Une meilleure solution consiste à ouvrir un programme d'édition de texte et de sauver le programme dans un fichier. Par convention les fichiers de programme Python utilisent l'extension '`.py`', comme par exemple '`prog1.py`'. Pour éditer du code Python de manière efficace il existe d'excellents logiciels libres :

- *Emacs* et *Vim* sont certainement les éditeurs de texte les plus puissants mais ils sont également difficiles à prendre en main.
- Il y a également *Gedit* sous GNOME ou *Kate* sous KDE.
- Il existe des éditeurs dédiés à Python : *DrPython, Eric3, SPE.*
- Enfin, sous windows, et si aucun des éditeurs de texte cités ci-dessus n'est installé, vous pouvez utiliser le bloc-notes.

Une fois le programme tapé et sauvegardé dans un fichier nommé `prog.py` par exemple, il suffit pour l'exécuter de saisir la commande `execfile('prog.py')` dans l'interpréteur Python .

1.1.2 Variables, opérations et typage

Voici un petit exemple permettant de faire une introduction aux variables et au typage sous Python. Les lignes commençant par '#' sont des commentaires qui sont ignorés par Python, le but des commentaires est de rendre le code plus lisible et plus clair.

```
# definition d'un entier valant 1
a=1
b=a/2
print b

# definition d'un nombre réel
a=1.1
b=a/2
print b
```

Ce programme illustre le fait que Python est *orienté-objet*. En effet l'instruction `a=1` définit dans la mémoire un entier `a` valant 1, alors que l'instruction `a=1.1` définit un nombre réel `a` valant 1.1. Ainsi, Python détecte automatiquement le type des variables et cela à de grandes conséquences dans la suite du programme :

- lorsque `a` est un entier l'opération `b=a/2` calcule la division euclidienne sur les entiers de 1 par 2, le résultat est donc 0.
- lorsque `a` est un réel l'opération effectuée est la division classique sur les réels.

Les opérations classiques $+, -, \times, /$, de Python sont donc différentes suivant le type des variables utilisés. En python, l'opération puissance se note `**`, la racine carrée `sqrt()`.

1.1.3 Fonctions

Python favorise la programmation structurée, c'est pourquoi la définition de fonction se fait de manière très simple avec Python via le mot clé `def`. Voici un exemple où nous définissons une fonction qui prend en entrée un x et qui retourne $x^2 + x/2$, le mot clé de retour en Python est `return`. Le type de x n'est pas défini dans la déclaration de fonction, le calcul effectué sera donc différent suivant le type de x.

```
# définition de la fonction
def myfunction(x):
   r=x*x+x/2
   return r

# calcul pour un réel
a=1.2
b1=myfunction(a)

# puis pour un entier
a=1
b2=myfunction(a)

# affichage des résultats
```

```
print "nombre reel:", b1
print "nombre entier", b2
```

Les tabulations dans la définition de la fonction sont obligatoires pour que Python sache où se termine la définition de cette fonction. De manière générale avec Python, la tabulation sert à délimiter les blocs de code. Le début d'un bloc d'instruction est signalé avec le symbole `:`.

1.1.4 Boucles FOR et WHILE

Nous souhaitons tracer la courbe représentative de la fonction $x \mapsto x^2 + x/2$ sur $[0, 9]$. Nous allons demander à Python de calculer les valeurs de cette fonction pour $x = 0, \cdots, 9$ et de les afficher. L'instruction `for` est très pratique pour cela :

```
# Boucle FOR
for i in range(0,10):
   print "i=", i, "f(i)=", myfunction(i)
```

La variable `i` est le *compteur* de la boucle `for` ; au cours des itérations de la boucle, il va prendre toutes les valeurs comprises entre 0 et 9 (cet intervalle de valeur étant décrit par l'instruction `range(0,10)`).

Une autre manière de créer une boucle est d'utiliser le mot clé `while`. Cette instruction nécessite de tester une condition, le programme exécute le bloc de la boucle seulement si la condition est vraie.

```
# Exemple boucle WHILE
# définition d'un compteur
k=0
condition=True
while condition:
   print "i=", k, "f(i)=", myfunction(k)
   # mise à jour du compteur
   k = k+1
   # arrêt si k>9
   condition = (k < 10)
```

Ce programme fait la même chose que le précédent, cela montre qu'il existe plusieurs façons d'écrire un programme.

Exercice 1

1. A l'aide d'une boucle FOR ou WHILE, calculer la somme des 1000 premiers entiers
2. Même question pour la somme des 100 premiers entiers au carré.

1.1.5 Instructions conditionnelles : IF - ELIF - ELSE

Nous voulons programmer une fonction qui prend les valeurs x si $x < 0$, 1 si $x \geq 0$ et $x < 1$, et x^2 sinon. Voici comment implémenter cette fonction avec les instructions `if`, `elif`, `else`

```
def fonction2(x):
   if (x<0):
      y=x
   elif (x<1):
      y=1
   else:
      y=x*x
   return y
```

Une manière de bien comprendre ce programme est de remplacer `if` par 'si', `elif` par 'ou si' et `else` par 'sinon'. Il est possible, après un IF de tester autant de conditions que l'on souhaite avec des `elif`. Par contre il ne peut y avoir qu'un seul `else` à la fin. Pour tester des conditions plus élaborées, il est pratique d'utiliser les opérations logiques ET, OU, ainsi que la négation. En Python ces opérations se notent '`&`', '`|`', et '`not`'.

1.1.6 Illustration - Calcul d'une moyenne

```
# programme Exemple - Calcul d'une moyenne

def moy(L):
   # calcule le nombre d'éléments dans la liste L
   l=len(L)

   # calcul de la moyenne
   # calcul de la somme des éléments de L
   m=0.0;
   for i in range(0,l):
      m=m+L[i]
   # on divise par le nombre d'éléments
   m=m/l
   # m est donc la moyenne arithmétique

   return m


# définition d'une liste vide
L=list()
OK=True
while OK:
   # demande un nombre au clavier
   a=input("rentrer un nombre = ")

   # Vérifions que a est bien un nombre
   if ((isinstance(a,int)) | (isinstance(a,float))):
      # ajout du nombre dans L
      L.append(a)

      print "La moyenne est : ", moy(L)
   else:
      print "Fin du programme"
      OK=False
```

2 Algorithmes pour le calcul de π

2.1 Méthode d'Archimède

La méthode d'Archimède pour calculer une évaluation du nombre π est basée sur l'utilisation de polygones réguliers comme approximation du cercle. Nous allons nous en inspirer pour établir un algorithme d'évaluation du nombre π, en approchant la surface du disque unité par celle d'un polygone. L'algorithme est le suivant :

Algorithm 1 Calcul de π par la méthode d'Archimède

Initialisation. Choisir $k_0 \in \mathbb{N}$.
loop
1. Construire un polygone régulier P_n avec k_n cotés, dont l'aire est une évaluation de celle du disque unité.
2. Calculer l'aire a_n du polygone P_n.
3. Multiplier le nombre de cotés par 2, $k_{n+1} = 2k_n$

end loop

L'algorithme 1 est une ébauche et est inutilisable tel quel pour calculer une approximation de π à l'aide d'un ordinateur. La suite $(a_n)_{n\in\mathbb{N}}$ générée par l'algorithme est clairement issue d'une récurrence. Choisissons $k_0 = 4$, le polygone P_0 est donc un carré que nous choisissons comme dans la Figure 2.1 (Gauche). D'après cette figure

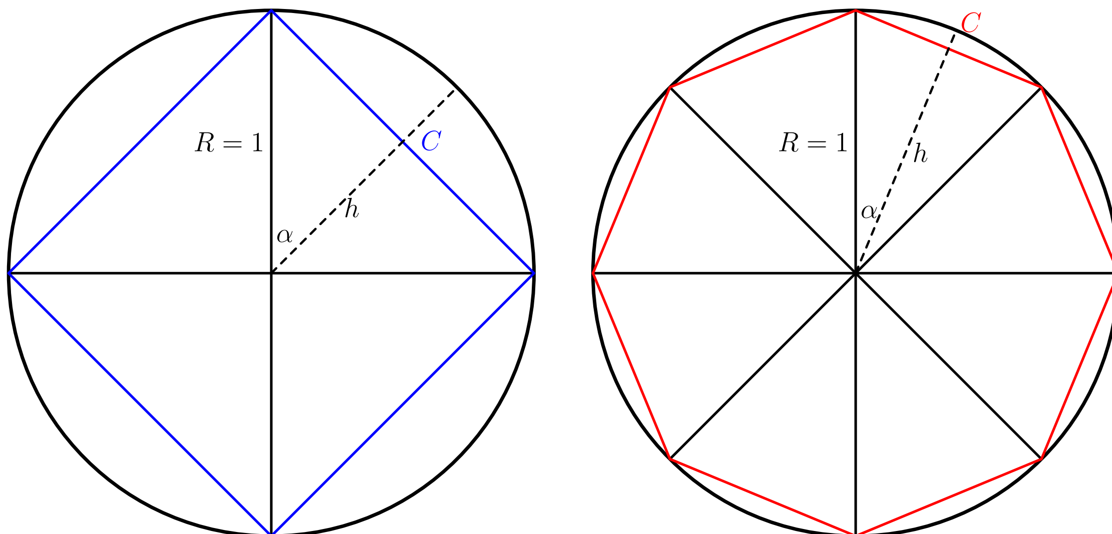

FIG. 1 – Gauche : Polygone P_0. Droite : Polygone P_1

nous voyons que le polygone est constitué de quatre triangles isocèles. Etant donné que l'angle α vaut $\frac{\pi}{4}$, que $h = R\cos(\alpha)$ et que $C = 2R\sin(\alpha)$, nous en déduisons

$$a_0 = k_0 R^2 \cos(\alpha)\sin(\alpha) = 2\sin(2\alpha) = 2$$

ce qui est une mauvaise évaluation de π !

Passons à l'étape suivante $k_1 = 8$, le polygone est donc un octogone, comme le montre la Figure 2.1 (Droite). Pour calculer son aire nous procédons comme précédemment, nous le découpons en 8 triangles, l'angle α nous permettant de calculer l'aire de chacun des triangles. Ainsi, comme $\alpha = \frac{\pi}{8}$, nous obtenons

$$a_1 = k_0 R^2 \cos(\alpha)\sin(\alpha) = 4\sin(2\alpha) = 2\sqrt{2} \approx 2.8284$$

ce qui est déjà un peu mieux.

Nous pouvons poursuivre cette récurrence pour calculer la valeur des aires $(a_i)_{i\in\mathbb{N}}$ suivantes, cela conduit à l'écriture de l'algorithme 2. Cet algorithme est plus précis, mais il est complètement inutile étant donné qu'il est nécessaire d'initialiser l'angle α à la valeur $\pi/4$, alors que le but de l'algorithme est de calculer π ! Nous remarquons aussi qu'il ne dispose pas de condition d'arrêt et génère donc une suite infinie de nombres $(a_n)_{n\in\mathbb{N}}$. Plusieurs questions mathématiques se posent : la suite $(a_n)_{n\in\mathbb{N}}$ converge-t-elle ? Est-ce qu'elle converge vers le nombre π comme nous le souhaitions ? Enfin, l'algorithme est il un "bon" moyen de calculer une approximation de π ?

Algorithm 2 Calcul de π par la méthode d'Archimède - II

Initialisation de l'angle : $\alpha \leftarrow \frac{\pi}{4}$
Initialisation du nombre de cotés : $k \leftarrow 4$
loop
  $a \leftarrow \frac{k}{2}\sin(2\alpha)$
  $k \leftarrow 2k$
  $\alpha \leftarrow \frac{\alpha}{2}$
end loop

2.1.1 Étude de l'algorithme

D'après la définition de l'algorithme 2, les suites (α_n) et (k_n) sont géométriques. Nous en déduisons

$$\forall n \in \mathbb{N}, \quad \alpha_n = \frac{\pi}{2^{2+n}}, \quad k_n = 2^{2+n}$$

d'où

$$\forall n \in \mathbb{N}, \quad a_n = \frac{k_n}{2}\sin(2\alpha_n) = 2^{n+1}\sin\left(\frac{\pi}{2^{n+1}}\right) \tag{1}$$

Nous connaissons donc le terme général de la suite $(a_n)_{n\in\mathbb{N}}$ générée par l'algorithme 2, étudions là. La Figure 2.1, nous laisse clairement imaginer que la suite des aires va être croissante et tendre vers π. Pour montrer la croissance il suffit de montrer que pour tout $n \in \mathbb{N}$,

$$a_{n+1} - a_n = 2^{n+1}\left(2\sin\left(\frac{\pi}{2^{n+2}}\right) - \sin\left(\frac{\pi}{2^{n+1}}\right)\right) \geq 0,$$

or cela se vérifie par une simple étude de la fonction $x \mapsto 2\sin\left(\frac{x}{2}\right) - \sin(x)$, pour $x > 0$, au voisinage de 0.

Montrons maintenant la convergence de la suite, comme il s'agit d'une suite croissante, il suffit de montrer qu'elle est majorée pour démontrer sa convergence. Nous savons que pour tout $x \in \mathbb{R}^+$, $\sin(x) \leq x$, ainsi

$$\forall n \in \mathbb{N}, \quad a_n \leq 2^{n+1}\frac{\pi}{2^{n+1}} = \pi,$$

La suite (a_n) est donc convergente. Il ne reste plus qu'à démontrer que sa limite est π. Pour cela nous utilisons la formule de Taylor-Lagrange sur la fonction sinus, à l'ordre 2. Nous obtenons l'encadrement suivant :

$$\forall x \in \mathcal{V}(0), \quad x - \frac{1}{6}|x|^3 \leq \sin(x) \leq x + \frac{1}{6}|x|^3.$$

Ainsi pour n assez grand

$$\begin{array}{rcl} \frac{\pi}{2^n} - \frac{1}{6}\left(\frac{\pi}{2^n}\right)^3 \leq & \sin\left(\frac{\pi}{2^n}\right) & \leq \frac{\pi}{2^n} + \frac{1}{6}\left(\frac{\pi}{2^n}\right)^3, \\ \pi - \frac{1}{6}\frac{\pi^3}{2^{2n}} \leq & a_{n-1} & \leq \pi + \frac{1}{6}\frac{\pi^3}{2^{2n}}, \end{array}$$

donc $\lim a_n = \pi$. En guise de conclusion, l'algorithme 2 permet de générer une suite de nombres $(a_n)_{n\in\mathbb{N}}$ qui converge vers le nombre π. La formule de Taylor-Lagrange nous donne également une information de la vitesse de convergence de l'algorithme car :

$$\forall n \in \mathbb{N}^*, \quad |a_n - \pi| \leq \frac{1}{6}\frac{\pi^3}{2^{2n}}.$$

Donc, à chaque itération la majoration de la distance entre a_n et π est divisée par 4.

Exercice 2

1. Programmer l'algorithme 2 avec Python, utiliser la commande '`from math import *`' pour que Python soit capable de calculer les sinus. Utiliser une boucle FOR ou WHILE pour forcer l'algorithme à s'arrêter. Le nombre π est accessible en python via la variable `pi`.
2. Écrire un programme qui affiche à chaque itération la distance entre a_n et π et comparer les résultats avec l'encadrement obtenu précédemment.
3. Relancer le programme après avoir modifié la valeur inititale de l'angle α.

2.1.2 Algorithme corrigé

L'algorithme 2 n'est pas correct au sens où

- on utilise la valeur de $\pi/4$ pour calculer π ?!
- on utilise la fonction sinus qui possède de nombreux liens avec le nombre π et qui est de plus une fonction complexe à calculer à partir des opérations arithmétiques de base.

Nous allons maintenant corriger l'algorithme afin de ne plus utiliser la fonction sinus, ni une initialisation avec un multiple rationnel de π. Pour pallier ces défauts, nous remarquons que nous souhaitons seulement calculer des valeurs bien spécifiques de la fonction sinus, à savoir les nombres $\sin\left(\frac{\pi}{2^n}\right)$ pour $n > 1$. Lorsque $n = 2$, la valeur de $\sin\left(\frac{\pi}{4}\right)$ est bien connue et vaut $\sqrt{2}/2$ (la longueur de la diagonale d'un carré de coté 1/2). Ensuite nous utilisons la formule trigonométrique

$$\cos a \cos b = \frac{\cos(a-b) + \cos(a+b)}{2}$$

pour en déduire

$$\cos\left(\frac{a}{2}\right) = \sqrt{\frac{1+\cos(a)}{2}}, \tag{2}$$

le théorème de Pythagore permet ensuite d'obtenir pour $a \in [0, \pi]$

$$\sin\left(\frac{a}{2}\right) = \sqrt{1-\cos^2\left(\frac{a}{2}\right)} \tag{3}$$

Les Équations (2) et (3) vont nous permettre de calculer les valeurs désirées de la fonction sinus (de manière récurrente), en utilisant seulement des opérations arithmétiques élémentaires. L'algorithme 3 est modifié en ce sens.

Algorithm 3 Calcul de π par la méthode d'Archimède - III

Initialisation du sinus : $s \leftarrow \sqrt{2}/2$
Initialisation du cosinus : $c \leftarrow \sqrt{2}/2$
Initialisation du nombre de cotés : $k \leftarrow 4$
loop
 $a \leftarrow ks$
 $k \leftarrow 2k$
 $c \leftarrow \frac{1+c}{2}$
 $s \leftarrow \sqrt{1-c}$
 $c \leftarrow \sqrt{c}$
end loop

Exercice 3

1. Programme cet algorithme avec Python. Utiliser une boucle WHILE pour arrêter le programme lorsque la condition $|a_n - a_{n-1}| < 1e-16$ devient vraie.
2. Comparer la valeur obtenue de π avec celle obtenue par l'algorithme 2 afin de constater que la précision est très mauvaise.
3. Programmer la méthode de Viète (algorihtme 4) et constater que la précision est bien meilleure.

2.1.3 Erreurs numériques

L'algorithme 3 que nous venons de construire permet de calculer des valeurs approchées de π à l'aide d'opérations arithmétiques simples. La preuve de convergence et l'étude de la vitesse de convergence restent valables car en théorie l'algorithme construit exactement la même suite $(a_n)_{n\in\mathbb{N}}$.

En pratique la situation est différente car les nombres réels ne peuvent pas être représentés dans un ordinateur. Il y a donc forcément des erreurs d'approximation. Les nombres utilisés par les ordinateurs sont appelés les nombres flottants, ils sont définis par une mantisse m, un exposant e et une base b (généralement $b = 2$ pour les ordinateurs). Ce triplet représente le nombre réel $m \cdot b^e$, la taille de la mantisse est fixée ce qui permet de définir la *précision* des nombres flottants. L'utilisation des nombres flottants a de nombreuses conséquences pour l'implémentation des algorithmes sur machine.

- Les nombres ne peuvent pas être représentés de manière exacte.
- Il faut prendre en compte les erreurs de troncations/arrondis.
- Les opérations machine $*, \times$ ne sont plus ni associatives ni commutatives.

Nous avons vu que l'algorithme 3 ne fonctionne pas correctement car la valeur calculée est beaucoup moins précise que la précision machine. Remarquons que les formules utilisées (2) et (3) ne sont pas des approximations, mais permettent bien de calculer des valeurs exactes. Cela signifie que l'erreur provient de la représentation non-exacte utilisée via les nombres flottantss et que l'erreur s'amplifie et devient non-négligeable au cours du calcul. Pour comprendre cela revenons à la formule permettant de calculer le sinus

$$\sin\left(\frac{a}{2}\right) = \sqrt{1-c}, \text{ où } c = \cos^2\left(\frac{a}{2}\right).$$

Introduisons le nombre $\varepsilon > 0$ qui va représenter l'erreur d'arrondi, et remplaçons la valeur de c par celle entachée d'erreur $c - \varepsilon$. Nous pouvons calculer maintenant la différence E entre la valeur exacte du sinus et celle entachée d'erreur :

$$E(c,\varepsilon) := \sqrt{1-c} - \sqrt{1-c+\varepsilon}$$

Comme nous l'avons vu dans les sections précédentes, la valeur de l'angle α_n tend vers 0 quand $n \to \infty$. Le cosinus carré c de cette angle, tend donc vers 1. Pour calculer l'erreur de calcul sur le sinus, il suffit de déterminer la limite suivante

$$\lim_{c \to 1^-} \sqrt{1-c} - \sqrt{1-c+\varepsilon} = E(1,\varepsilon) = \sqrt{\varepsilon}.$$

Ainsi pour une petite erreur d'arrondi $\varepsilon > 0$ sur la valeur du cosinus, l'erreur devient de l'ordre de $\sqrt{\varepsilon}$ sur la valeur calculée de sinus. L'utilisation des nombres flottants introduit donc des erreurs que l'on ne peut négliger et qui dégrade la qualité de l'algorithme.

Exercice 4 Pensez-vous qu'il suffit d'augmenter la précision des nombres flottants pour éliminer ce problème .

2.1.4 La méthode de Viète

La méthode que nous avons utilisé pour calculer π à été découverte par François Vièta, dit Viète (1540-1603). En étudiant la surface des polygones réguliers inscrit dans le cercle, à $2n$ cotés, il à établi la formule suivante

$$\pi = 2 \times \frac{2}{\sqrt{2}} \times \frac{2}{\sqrt{2+\sqrt{2}}} \times \frac{2}{\sqrt{2+\sqrt{2+\sqrt{2}}}} \times \cdots$$

Cette formule ayant été établie en calculant la surface de la même suite de polygones que pour les algorithmes précédents, il est clair qu'elle fournit, de proche en proche, les mêmes évaluations du nombre π (modulo les erreurs de calculs avec les nombres flottants!). Il est également assez facile de démontrer la formule de Viète à partir de l'algorithme 3 en utilisant un raisonnement par récurrence. La démonstration de convergence reste donc valable pour la formule de Viète.

Il s'agit donc du même algorithme écrit différement, la formule de Viète à pourtant un avantage majeur : le problème de la sensibilité des calcul par rapport à la précision des nombres flottants à disparu.

Algorithm 4 Calcul de π - Formule de Viète

$a \leftarrow 2$
$k \leftarrow 1$
$c \leftarrow 0$
loop
  $c \leftarrow \sqrt{2+c}$
  $b \leftarrow \frac{2}{c}$
  $a \leftarrow ab$
end loop

2.1.5 Le vrai algorithme d'Archimède

La méthode d'Archimède n'est en fait pas basé sur le calcul des surfaces d'une suite de polygones, mais sur l'évaluation du périmètre de ces polygones. C'est la première méthode connue à avoir été proposée pour le calcul théorique exact du nombre π, sa description est donnée par l'algorithme 5. Nous ne démontrerons pas ici comment établir cet algorithme ni comment prouver sa convergence. Le lecteur curieux pourra se réferer à [J.P. Delahaye, "le fascinant nombre π"]. Il faut tout de même préciser que la vitesse de convergence de l'algorithme d'Archimède est le même que celui de la formule de Viète. Cela semblait prévisible étant donné que la même construction géométrique est à la base des deux algorithmes. Dans l'algorihtme 5, les nombres a et b fournissent un encadrement de la valeur de π. Avec cette méthode Archimède à déterminé un encadrement de π très précis

$$\frac{223}{71} \leq \pi \leq \frac{22}{7}.$$

2.2 Méthode des fléchettes (Monte-Carlo)

Nous venons de voir une méthode géométrique pour calculer π basée sur une approximation du disque unité par une suite de polygones. Nous allons maintenant présenter une méthode de Monte-Carlo pour le calcul du nombre π. Une méthode de Monte-Carlo est une méthode permettant de calculer des valeurs numériques en utilisant, pour ce

Algorithm 5 Calcul de π - algo. d'Archimède (le vrai)

$a \leftarrow 2\sqrt{3}$
$b \leftarrow 3$
loop
  $a \leftarrow \frac{2}{\frac{1}{a}+\frac{1}{b}} s$
  $b \leftarrow \sqrt{ab}$
end loop

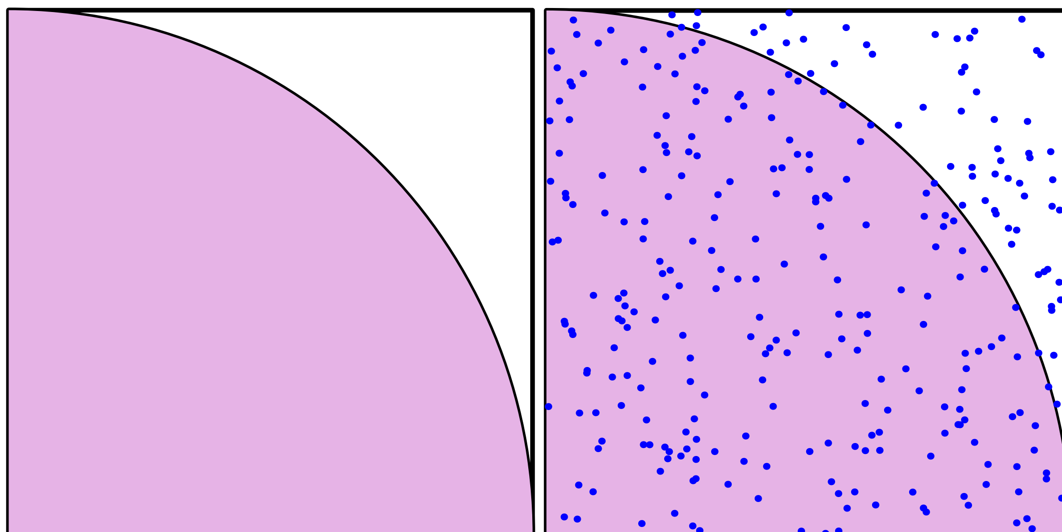

FIG. 2 – Gauche : Cible carrée et le quart de cercle (en couleur) dont nous voulons déterminer l'aire
. Droite : Algorithme réalisé avec 300 lancés de fléchettes, 234 sont dans le quart de cercle, ce qui donne 3.12 comme évaluation de π

faire, des procédés aléatoires. La méthode des fléchettes que nous allons présenter s'utilise également pour évaluer des intégrales, par exemple pour calculer la surface d'un lac ou d'une forêt à partir d'une prise de vue aérienne.

Le principe de la méthode est de jeter un grand nombres de fléchettes sur une cible carrée puis de déterminer la proportion de fléchettes qui sont arrivée dans le disque inscrit au carré. Ce dénombrement nous donnera une évaluation du rapport entre l'aire du disque et celle de la cible. Il faut bien sûr, faire l'hypothèse que les fléchettes sont lancées d'assez loin, cela afin que la disposition des fléchettes sur la cible suive une loi de probabilité uniforme. Nous supposons également que les tirages aléatoires sont indépendants.

Cet algorithme est complètement différent des algorithmes que nous avons vus précédemment : ce n'est pas un algorithme *déterministe*. Nous pouvons par exemple exécuter plusieurs fois l'algorithme de Monte-Carlo et toujours obtenir des résultats différents, ce qui est impossible avec la méthode géométrique précédente. En conséquence,il est nécessaire d'étudier la convergence de la suite générée par la méthode de Monte-Carlo à l'aide de la théorie des probabilités.

2.2.1 Construction de l'algorithme

Nous définissons la cible comme étant l'ensemble $[0,1]^2$ et nous allons mesurer le nombre de fléchettes qui arrivent dans le quart de cercle unité. Après avoir lancé $N \in \mathbb{N}^*$ fléchettes et noté leur positions (x_i, y_i), nous pouvons savoir où elles sont arrivées en étudiant la valeur de $R_i = x_i^2 + y_i^2$. Si $R_i < 1$ cela signifie que la fléchette i est dans le quart de cercle unité, par contre cela est faux si $R_i > 1$. Le cas $R_i = 1$ est négligé étant donné que la probabilité qu'une fléchette arrive exactement sur le cercle est égale à 0. L'algorithme 6 décrit cette méthode.

Dans l'algorithme (6), l'instruction *random* est un tirage aléatoire d'un nombre réel entre 0 et 1, avec une distribution de probabilité uniforme. Les variables x et y sont les abcisses et ordonnées des fléchettes lancées ; comme elles ont été choisies de manière aléatoire avec une probabilité uniforme nous en déduisons que le point $(x, y) \in [0,1]^2$ est choisi de manière uniforme sur la cible. A la fin de l'algorithme, la variable p contient l'évaluation calculée du nombre π, la multiplication par 4 du numérateur à la ligne précédente permet simplement de passer

Algorithm 6 Calcul de π - Méthode de Monte-Carlo

Initialisation du nombre de fléchettes : $N \in \mathbb{N}^*$
Initialisation du Numérateur : $c \leftarrow 0$
for $i = 1$ to N **do**
  $x \leftarrow random$
  $y \leftarrow random$
  $R \leftarrow x^2 + y^2$
  if $R < 1$ **then**
    $c \leftarrow n + 1$
  end if
end for
$c \leftarrow 4c$
$p \leftarrow \frac{c}{N}$

du quart de cercle au cercle complet.

2.2.2 Étude de la convergence au sens des probabilités

La distribution de probabilité étant uniforme sur la cible, la probabilité qu'une fléchette arrive dans le quart de cercle est donc égale à l'aire du quart de cercle divisé par l'aire de la cible. Notons X_i, pour $i = 1, \cdots, N$, la variable aléatoire qui vaut 1 si la fléchette est dans le cercle et 0 sinon. Nous avons

$$\mathbb{P}(X_i = 1) = \frac{\pi}{4}, \text{ et } \mathbb{P}(X_i = 0) = 1 - \frac{\pi}{4},$$

et nous en déduisons

$$\mathbb{E}(X_i) = \frac{\pi}{4}, \quad Var(X_i) = \frac{\pi}{4} - \frac{\pi^2}{16}.$$

Supposons que le nombre de tirages aléatoires est infini, la suite de $(X_i)_{i \in \mathbb{N}}$ est une suite de variables aléatoires indépendantes et identiquement distribuées, d'espérance $\frac{\pi}{4}$ et de variance $\frac{\pi}{4} - \frac{\pi^2}{16}$. Il ne reste plus qu'a appliquer la *loi des grands nombres* pour montrer la convergence, et en déduire que $\frac{4}{N} \sum_{i=1}^{N} X_i$ converge *presque-sûrement* vers π.

La loi des grands nombres nous permet de conclure sur la convergence de l'algorithme au sens des probabilités, mais elle ne nous donne pas d'information sur la vitesse de convergence. Nous ne pouvons pas dire si, par-exemple, il est préférable pour calculer π d'utiliser la méthode d'Archimède plutôt que celle de Monte-Carlo. Pour cela nous allons étudier la suite de variables aléatoires

$$S_n = \frac{4}{N} \sum_{i=1}^{n} X_i,$$

comme l'espérance est linéaire il est clair que

$$\forall n \in \mathbb{N}^*, \mathbb{E}(S_n) = \pi.$$

Calculons la variance des S_n, or la variance de la somme des X_i et égale à la somme des variances comme les variables aléatoires X_i sont indépendantes. Ainsi

$$Var(S_n) = \frac{16}{n^2} \sum_{i=1}^{n} Var(X_i) = \frac{16}{n} \left(\frac{\pi}{4} - \frac{\pi^2}{16} \right).$$

Nous allons maintenant pouvoir conclure en utilisant l'inégalité de Bienaymé-Tchebychev

$$\mathbb{P}(|S_n - \mathbb{E}(S_n)| \geq \alpha) \leq \frac{16}{n\alpha^2} \left(\frac{\pi}{4} - \frac{\pi^2}{16} \right),$$

un calcul rapide montre que pour $\alpha = \frac{17}{\sqrt{n}}$, nous avons

$$\mathbb{P}(|S_n - \mathbb{E}(S_n)| \geq \alpha) \leq 0.01.$$

Ce qui signifie qu'avec 99% de chance, la variable S_n est dans l'intervalle $[\pi-\alpha, \pi+\alpha]$. La convergence de l'algorithme de Monte-Carlo est donc moins bonne que celle d'Archimède. Il faut en effet, avec la méthode de Monte-Carlo, quadrupler le nombre d'itérations pour diviser par 2 la distance entre la valeur prise par S_n et π (avec 99% de chances). Alors que la méthode d'Archimède divise l'erreur par quatre à chaque itération.

Exercice 5

1. Programmer la méthode des fléchettes avec Python. Utiliser la commande '`from random import *`' pour avoir accès au générateur de nombres pseudo-aléatoires. La commande pour générer un nombre aléatoire avec distribution uniforme sur $[0, 1]$ est `random()`
2. Calculer des évaluations du nombre π pour $N = 200, 1000, 4000$. Que constatez-vous ?