

Exercices d'introduction aux calculs parallèles

Exercices avec Python

(CPE 21-22)

(Version élèves)

Mai 2022

I Plan (de ce document)

- I) Voir les documents pdf des 2 cours sur la programmation concurrente.

🔊 **Chaque exercice porte un barème.**

Rendre assez d'exercices pour constituer 20 points. Tout point supplémentaire est en bonus.

- II) **Exercices à réaliser** avec la package multiprocessing.
 1. Course Hippique et affichage à l'écran (éventuellement avec le module *curses* de Python)
 2. Client-serveur de calculs
 3. Gestion des ressources (Billes)
 4. Calcul parallèle d'estimation de PI
 - Il y a différentes autres techniques de calcul de PI
 5. Merge-sort par de multiples Processus
 6. Quick-sort par de multiples Processus
 7. Compléments sur le calcul de Pi

Une 2nde séquence d'exercices est proposée par la suite.

II Quelques fonctions utiles

Dans la package **multiprocessing**, on dispose de quelques fonctions utilitaires :

- *multiprocessing.active_children()* : renvoie la liste des processus fils encore en vie du processus courant.
- *multiprocessing.cpu_count()* : renvoie le nombre de CPU (processus réellement parallèles possibles) sur votre ordinateur.
- *multiprocessing.current_process()* : renvoie l'objet Processus correspondant a processus courant.
- *multiprocessing.parent_process()* : renvoie un object Process qui correspond au parent du processus courant. Pour le processus principal (main), le processus parent est None.
- voir les autres méthodes utilitaires en page

<https://docs.python.org/3.8/library/multiprocessing.html?highlight=queue#multiprocessing.Queue>

III Exercices à réaliser

III-A Exercice : Course Hippique

Difficulté : */*****, (3 points)

On souhaite réaliser, sur les machine Linux, une course hippique. L'image suivante donne une idée de cette application (dans une fenêtre *Terminal* sous *Ubuntu*).

```

Fichier  Édition  Affichage  Signets  Configuration  Aide
(A>
(B>
(C>
(D>
(E>
(F>
(G>
(H>
(I>
(J>
(K>
(L>
(M>
(N>
(O>
(P>
(Q>
(R>
(S>
(T>

Best : (J > en ligne 10 position 49, Worst en position 40
Worst en position 40

tous lancés

```

Pour ne pas compliquer la "chose", on n'aura pas recours aux outils d'affichages graphiques. Les affichages se feront en console avec des séquences de caractères d'échappement (voir plus loin).

Chaque cheval est représenté simplement par une lettre (ici de 'A' à 'T') que l'on a entouré ici par '(' et '>' : ce qui donne par exemple '(A>' pour le premier cheval (vous pouvez laisser libre cours à vos talents d'artiste).

A chaque cheval est consacré une ligne de l'écran et la progression (aléatoire) de chaque cheval est affichée.

☞ Pour l'instant, ignorez les autres lignes affichées ("Best : .." et les suivantes).

Une version basique de cette course est fournie sur le site CPE.

☞ Ce code ne devrait pas fonctionner sous Windows de Microsoft (qui ne dispose pas d'écran VT100 ou similaire).

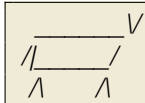
☞ Pour en savoir plus sous Linux, reportez-vous à la page du manuel de "screen" (ou regarder sur le WEB).

III-A-1 Travail à réaliser

Compléter ce code pour réaliser les points suivants :

- Vous constatez que les affichages à l'écran ne sont pas en exclusion mutuelle.
Réglez ce problème en utilisant un verrou.
- Ajouter un processus supplémentaire appelé *arbitre* qui affiche en permanence le cheval qui est en tête ainsi que celui qui est dernier.
A la fin de la course, il affichera le gagnant (et les éventuels canassons ex aequos).
- Éventuellement, permettre dès le départ de la course, de prédire un gagnant (au clavier)
- Essayer d'améliorer le dessin de chaque cheval en utilisant seulement des caractères du clavier (pas de symbole graphiques).

Par exemple, un cheval peut être représenté par (on dirait une vache !) :



III-B Exercice : faites des calculs

Difficulté : **/*****, (3 à 5 points)

Problème traité sous forme simple en cours : échange entre un client demandeur, un serveur calculateur. Consulter les pdfs du cours.

On souhaite réaliser **plusieurs calculs en parallèle** demandés par un à plusieurs demandeurs.

- **Version 1 : Un demandeur, n calculateurs. (3 points)**

Le processus demandeur dépose une expression arithmétique (par exemple $2 + 3$) dans une file d'attente (*multiprocessing.Queue*) des demandes.

Côté serveurs, chaque processus calculateur récupère une expression, évalue l'expression et dépose le résultat dans une file d'attente des résultats.

Réalisez cette version.

- **Version 2 : m demandeurs, n calculateurs. (5 points)**

Dans cette version, on a plusieurs demandeurs $d_i, i = 1..m$.

Dans ce cas, pour que les demandeurs récupèrent leurs propres résultats, il faudra identifier les demandes par le demandeur d_i .

Lorsque le résultat est calculé et déposé par un processus calculateur, il ajoute l'identifiant d_i du demandeur. Ainsi, le demandeur peut filtrer la Queue des résultats pour trouver les réponses à ses demandes.

Réaliser cette version en créant plusieurs demandeurs et plusieurs calculateurs capables de traiter des demandes de calculs fréquentes.

- Ajouter une variante où au lieu d'une expression, le demandeur communique une **fonction** particulière à appliquer (*lambda function*) (2 points).

☞ L'exemple suivant donne une version avec **os.fork()** ou un demandeur (père) et un seul calculateur (le fils) communiquent via un **os.pipe()**. Rappelez-vous que les pipes sont utilisés pour la communication entre deux processus. Vous devez utiliser *multiprocessing.Queue* pour pouvoir établir une communication indifférenciée entre de multiples processus. Voir le cours pour des exemples de Queue.

- Le fils (qui fait les calculs)

```
import time,os,random
def fils_calcullette(rpipe_commande, wpipe_reponse):
    print("Bonjour du Fils", os.getpid())

    while True:
        cmd = os.read(rpipe_commande, 32)
        print("Le fils a reçu ", cmd)
        res=eval(cmd)
        print("Dans fils, le résultat =", res)
```

```

os.write(wpipe_reponse, str(res).encode())
print("Le fils a envoyé", res)
time.sleep(1)

os._exit(0)

```

- Le père :

- Prépare une opération arithmétique (p. ex. 2+3) ; la transmet au fils
- Récupère le résultat sur un *pipe*.

```

if __name__ == "__main__":
    rpipe_reponse, wpipe_reponse = os.pipe()
    rpipe_commande, wpipe_commande = os.pipe()

    pid = os.fork()

    if pid == 0:
        fils_calculer(rpipe_commande, wpipe_reponse)
        assert False, 'fork du fils n a pas marché !' # Si échec, on affiche un message

    else:
        # On ferme les "portes" non utilisées
        os.close(wpipe_reponse)
        os.close(rpipe_commande)

        while True:
            # Le pere envoie au fils un calcul aléatoire à faire et récupère le résultat
            opd1 = random.randint(1,10)
            opd2 = random.randint(1,10)
            operateur=random.choice(['+', '-', '*', '/'])
            str_commande = str(opd1) + operateur + str(opd2)

            os.write(wpipe_commande, str_commande.encode())
            print("Le père va demander à faire : ", str_commande)
            res = os.read(rpipe_reponse, 32)
            print("Le Pere a reçu ", res)
            print('-'* 60)
            time.sleep(1)

```

- Trace :

```

Le père va demander à faire : 5/9
Bonjour du Fils 12851
Le fils a reçu 5/9
Dans fils, le résultat = 0.5555555555555556
Le fils a envoyé 0.5555555555555556
Le Pere a reçu 0.5555555555555556

```

```

-----
Le père va demander à faire : 5/2
Le fils a reçu 5/2
Dans fils, le résultat = 2.5
Le fils a envoyé 2.5
Le Pere a reçu 2.5

```

```

-----
Le père va demander à faire : 8*6
Le fils a reçu 8*6
Dans fils, le résultat = 48
Le fils a envoyé 48
Le Pere a reçu 48

```

```

...

```

III-C Gestionnaire de Billes

Difficulté : ***/****, (5 points)

On souhaite réaliser l'exemple suivant (lire à la fin de ce sujet à propos des *variables de condition*).

- N processus (p. ex. $N = 4$) ont besoin chacun d'un nombre k d'une ressource (p. ex. des Billes) pour avancer leur travail
- Cette ressource existe en un **nombre limité** : on ne peut satisfaire la demande de tout le monde en même temps.

Par exemple, la demande de $Process_{i=1..4}$ est de (4, 3, 5, 2) billes et on ne dispose que de $nb_max_billes = 9$ billes

- Chaque Processus répète la séquence (p. ex. m fois) :
"demander k ressources, utiliser ressources, rendre k ressources"
- Le "main" crée les N processus.
Il crée également un processus *contrôleur* qui vérifie en permanence si le nombre de Billes disponible est dans l'intervalle $[0..nb_max_billes]$. En cas d'incohérence de la valeur du nombre de billes disponibles, il affiche un message et arrête tout !
- Pour chaque P_i , l'accès à la ressource se fait par une fonction "demander(k)" qui doit bloquer le demandeur tant que le nombre de billes disponible est inférieur à k
- P_i rend les k billes acquises après son travail et recommence sa séquence

Pseudo algorithmes :

• Main :

```
MAIN :
  Creer N processus travailleurs (avec mp.Process)
  Lancer ces N processus
  Creer un processus contrôleur
  Lancer contrôleur
  ... tourner les pouces un peu ...
  Attendre la fin des N processus
  Terminer le processus contrôleur
```

• Travailleur :

```
Travailleur(k_billes):
  Iterer m fois :
    demander k_billes # peut être bloquant si k_billes non disponibles
    simuler le travail avec un delai (sleep(nb_secondes) en fonction de la valeur k_billes)
    rendre k_billes # jamais bloquant
```

• Demander k billes :

👉 Notez bien qu'il faudra un "Tant que" et non un simple "Si".

```
Demander(k_billes): # Il faudra ajouter un verrou d'accès au nombre de billes disponibles
  Tant que nbr_disponible_billes < k_billes : # Ce test doit être fait dans une Section Critique (SC)
    se bloquer (sur un semaphore d'attente) <- Ne pas oublier de libérer le verrou avant de vous mettre en attente !
  nbr_disponible_billes = nbr_disponible_billes - k_billes
```


Cette version est une version avec "attente passive" : on ne consomme pas du temps CPU. Voir ci-dessous.

- Rendre k billes (voir ci-dessous "attente active / passive")

```
rendre(k_billes):
    Dans une SC :
        nbr_disponible_billes = nbr_disponible_billes + k_billes
    Selon la méthode choisie, penser à libérer ceux qui sont bloqués (sur un semaphore d'attente) # voir "demander"
```

☞ Noter que **Demander(k_billes)** sera équivalent à **sem.acquire(k_jetons)** (fonction qui n'existe pas dans le package multiprocessing) ! De même pour **rendre** et **release()**

- Contrôleur :

```
Contrôleur(max_billes):
    Iterer toujours :
        Dans une SC : # SC = Section Critique à protéger avec un verrou / sémaphores / etc..
            Verifier que 0 <= nbr_disponible_billes <= max_billes
        delai(1 sec)
```

☞ A propos de demander() / rendre() :

- Dans *demander()*, un *if* à la place de *while* ne suffit pas.
Lorsqu'on sera réveillé (depuis *rendre()*) sur un sémaphore d'attente, il se pourrait que le nombre de billes libérées soit encore insuffisant pour satisfaire notre demande. Le *while* permet de refaire ce test.
- Ce mode de fonctionnement (se bloquer sur le sémaphore d'attente) est une **attente passive** : on ne consomme pas du temps CPU et on attend que l'on nous réveille.¹

Attente active / Passive :

Dans une version avec une **attente active**, on peut écrire (code détaillé) :

```
Demander_attente_active (k_billes): # Dans ce pseudo-code, la gestion du verrou/sémaphore est absente
    demander verrou sur nb_billes
    Tant que nbr_disponible_billes < k_billes :
        libérer verrou sur nb_billes
        attendre un peu # sleep(0.5) par exemple
    demander verrou sur nb_billes
    nbr_disponible_billes = nbr_disponible_billes - k_billes
```

Dans cette version, il n'y a plus de sémaphore d'attente et par conséquent, la fonction *rendre()* ne contiendra plus une libération sur le même sémaphore d'attente.

```
rendre_attente_active(k_billes):
    Dans une SC :
        nbr_disponible_billes = nbr_disponible_billes + k_billes
```

☞ **Voir cours** : il existe également les **variables de condition** qui réalisent l'effet d'une attente passive (avec les primitive *attendre(condition)* / *signaler(condition)* / *signaler_à_tous(condition)* où p. ex. *condition* = *nbr_disponible_billes* >= *k_billes*)

1. Un exemple équivalent : supposons attendre un paquet par la poste. Dans une attente active, on regarde sans cesse par la fenêtre pour savoir si le facteur passe ! Dans une attente passive, on se met d'accord avec le facteur pour qu'il sonne quand il passe (pour nous réveiller).

III-D Estimation de PI

☞ Ci-dessous, la version séquentielle. Voir le barème sur l'exercice qui vient ensuite..

- La valeur de PI peut être estimée de multiples manières. Nous en exposons une ci-dessous. D'autres méthodes sont exposées plus loin.
- Nous étudions ci-dessous la méthode qui utilise un cercle unitaire par une méthode séquentielle avant de donner (exercice) une solution parallèle (de la même méthode).

III-D-1 Exemple : Calcul de PI par un cercle unitaire

On peut calculer une valeur approché de PI à l'aide d'un cercle unitaire et la méthode Monte-Carlo (MC).

Principe : on échantillonne un point (couple de réels $(x, y) \in [0.0, 1.0]$) qui se situe dans $\frac{1}{4}$ du cercle unitaire et on examine la valeur de $x^2 + y^2 \leq 1$ (équation de ce cercle).

- Si "vrai", le point est dans le quart du cercle unitaire (on a un *hit*)
- Sinon (cas de *miss*), ...
- Après N (grand) itérations, le nombre de *hits* approxime $\frac{1}{4}$ de la surface du cercle unitaire, d'où la valeur de π . Notez que l'erreur de ce calcul peut être estimée à $\frac{1}{\sqrt{N}}$.
- On programmera cette méthode, d'abord en Mono-processus (voir ci-dessous) puis en multi-processus. On comparera ensuite les temps de calculs.

III-D-2 Principe Hit-Miss (Monte Carlo)

- Le code Python suivant permet de calculer Pi selon le principe de hit-miss ci-dessus.

```
import random, time

# calculer le nbr de hits dans un cercle unitaire (utilisé par les différentes méthodes)
def frequence_de_hits_pour_n_essais(nb_iteration):
    count = 0
    for i in range(nb_iteration):
        x = random.random()
        y = random.random()

        # si le point est dans l'unit circle
        if x * x + y * y <= 1: count += 1
    return count

# Nombre d'essai pour l'estimation
nb_total_iteration = 10000000

nb_hits=frequence_de_hits_pour_n_essais(nb_total_iteration)

print("Valeur estimée Pi par la méthode Mono-Processus : ", 4 * nb_hits / nb_total_iteration)

#TRACE :
# Calcul Mono-Processus : Valeur estimée Pi par la méthode Mono-Processus : 3.1412604
```

- ☞ Ajouter la mesure du temps du calcul (pour une comparaison ultérieure).

III-E Exercice : Estimation parallèle de Pi

Difficulté : */*****, (3 points)

Exercice-3 : modifier le code précédent pour effectuer le calcul à l'aide de plusieurs Processus.

☞ Mesurer le temps et comparer.

N.B. : dans la méthode envisagée, on fixe un nombre (p. ex. $N = 10^6$) d'itérations.

Si on décide de faire ce calcul par k processus, chaque processus effectuera $\frac{N}{k}$ itérations.

Utiliser la fonction `time.time()` pour calculer le temps total nécessaire pour ce calcul. Vous constaterez que ce temps se réduira lors d'utilisation des processus dans un calcul parallèle.

- Variantes de cette méthode : d'une des manières suivantes :
 - 4 Processus où chacun effectue ces calculs sur un quart du cercle unitaire.
 - Plusieurs Processus calculent sur le même quart du cercle et on prend la moyenne
 - etc.
- Voir le section III-H en page 15 pour les autres méthodes de calcul de Pi.

III-F Exercice : Calcul parallèle du Merge Sort

Difficulté : ***/*****, (5 points)

Le principe du tri fusion : pour trier un tableau T de N éléments,

- Scinder T en deux sous-tableaux $T1$ et $T2$
- Trier $T1$ et $T2$
- Reconstituer T en fusionnant $T1$ et $T2$

→ $T1$ et $T2$ sont chacun triés et leur fusion tient compte de cela.

III-F-1 Version séquentielle de base

- Une version séquentielle de cette méthode de tri est déposée sur le site de CPE.

Exercice-4 : Transformer d'abord cette version de base en une version de tri **sur-place**.

C-à-d, pour être plus efficace, ne pas découper le tableau à trier en sous tableaux mais travailler avec des 'tranches' de ce dernier. Par conséquent, un sous tableau sera repéré par deux indices début - fin (= une zone du tableau global).

Exercice-5 : Écrire ensuite une version parallèle avec des Processus de cette méthode de tri = la version parallèle de l'exercice 4.

☞ En général, chaque Processus sous-traite à un processus fils la moitié du tableau qui lui est assigné et s'occupe lui-même de l'autre moitié.

☞ Au total, ne dépassez pas 8 processus pour un processeur Intel I7, 4 pour les modèles I5 ou I3.

☞ Pour représenter le tableau à trier, vous pouvez utiliser un 'Array' mais le gain de performance ne sera pas sensible. Par contre, l'utilisation d'un *SharedArray* vous garantira un gain substantiel.

Pour une information plus complète sur ce module, voir le site

<https://pypi.python.org/pypi/SharedArray>

Extrait de ce site : un exemple d'utilisation de *SharedArray*.

```
import numpy as np
import SharedArray as sa

# Create an array in shared memory
a = sa.create("shm://test", 10)

# Attach it as a different array. This can be done from another
# python interpreter as long as it runs on the same computer.
b = sa.attach("shm://test")

# See how they are actually sharing the same memory block
a[0] = 42
print(b[0])

# Destroying a does not affect b.
del a
```

```
print(b[0])

# See how "test" is still present in shared memory even though we
# destroyed the array a.
sa.list()

# Now destroy the array "test" from memory.
sa.delete("test")

# The array b is not affected, but once you destroy it then the
# data are lost.
print(b[0])
```

III-G Exercice : tri-rapide

Difficulté : **/*****, (5 points)

Exercice-6 : Faites de même avec la méthode de Tri Quick-Sort dont le principe et la version séquentielle de base sont rappelés ci-dessous.

Principe de la méthode :

Pour trier un tableau T de N éléments,

- Désigner une valeur du tableau (dit le *Pivot* p)
- Scinder T en deux sous-tableaux $T1$ et $T2$ tels que les valeurs de $T1$ soient $\leq p$ et celles de $T2$ soient $> p$
- Trier $T1$ et $T2$
- Reconstituer T en y plaçant $T1$ puis p puis $T2$

☞ Pour trier chacun des sous-tableaux, procéder de la même manière

☞ Pour le choix du pivot, on désigne en général le premier élément du tableau (sans garantie d'équité en tailles de $T1$ et $T2$)

☞ Au lieu de créer autant de processus que de sous tableaux, une gestion plus modérée des processus (pour ne pas en créer beaucoup) est recommandée. On se limitera à 8 sur un I7.

Algorithme de la version de base

```
def qsort_serie_sequentiel_avec_listes(liste):  
    if len(liste) < 2: return liste  
  
    # Pivot = liste[0]  
    gche = [X for X in liste[1:] if X <= liste[0]]  
    drte = [X for X in liste[1:] if X > liste[0]]  
  
    # Trier chaque moitié "gauche" et "droite" pour regrouper en plaçant "gche" "Pivot" "drte"  
    return qsort_serie_sequentiel_avec_listes(gche) + [liste[0]] + qsort_serie_sequentiel_avec_listes(drte)
```

III-H Autres méthodes de calcul de PI

Pour indication et complément : il existe de multiples méthodes de calcul de PI. En voici quelques unes. Un seul exercice d'estimation de PI peut-être rendu. Les méthodes ci-dessous sont informatives et n'ont donc pas de barème.

III-I PI par la méthode arc-tangente

👁 Vu en cours.

Méthode arc-tangente :

- On peut calculer une valeur approchée de PI par la méthode suivante :

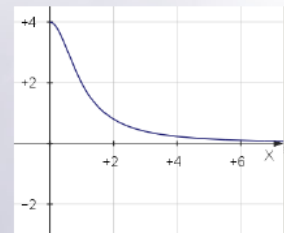
$$\pi \approx \int_0^1 \frac{4}{1+x^2} dx \quad \text{ou la version discrète} \quad \pi \approx 1/n \sum_{i=1}^n \frac{4}{1+x_i^2}$$

où l'intervalle $[0, 1]$ est divisé en n partitions (bâton) égales.

N.B. : pour que la somme des bâtons soit plus proche de l'aire sous la courbe, considérons le milieu des bâtons :

$$\sum_{i=1}^n \frac{4}{1+x_i^2} \approx \sum_{i=1}^n \frac{4}{1+\left(\frac{i-0.5}{n}\right)^2} = \sum_{i=0}^{n-1} \frac{4}{1+\left(\frac{i+0.5}{n}\right)^2}$$

→ $\left(\frac{i-0.5}{n}\right)$ ramène i dans $[0, 1]$ (i.e. x_i)



👁 Une fois pour toutes, vérifier et donner la bonne formule.

→ En bas, redonner $1/N$, les élèves l'oublient !

Indication : pour vous aider, voilà l'exemple partiel du code (séquentiel) pour n donné qui met en place la formule ci-dessus :

```
def arc_tangente(n):
    pi = 0
    for i in range(n):
        pi += 4/(1+ ((i+0.5)/n)**2)
    return (1/n)*pi
```

III-J *Par la méthode d'espérance*

Difficulté : */****

On tire N valeurs de l'abscisse X d'un point M dans $[0; 1]$

On calcule la somme S de N valeurs prises par $f(X) = \sqrt{1 - X^2}$

La moyenne des ces N valeurs de $f(X)$ est une valeur approchée de la moyenne de f et donc de l'aire du quart de cercle : $\frac{S}{N} = \pi/4$.

→ La division par N (= nombre de *pas*) pour obtenir la surface des battons

III-K *Par la loi Normale*

Difficulté : */****

Pour x centrée ($\mu = 0$) suivant une loi *Normale*, $f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}}$

Si $\int_{-\infty}^{+\infty} f(x) dx = 1$, on peut approximer la valeur de π

☞ On peut utiliser une variable centrée réduite ($\sigma = 1, \mu = 0$) pour simplifier les calculs avec

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \quad \text{d'où} \quad \int_0^{\infty} f(x) dx = \frac{1}{2}$$

III-L Approximation de π par les Aiguilles de Buffon

On lance un certain nombre de fois une aiguille sur une feuille cadriée et l'on observe si elle croise des lignes horizontales. On peut également utiliser des stylos sur les lattes d'un parquet.

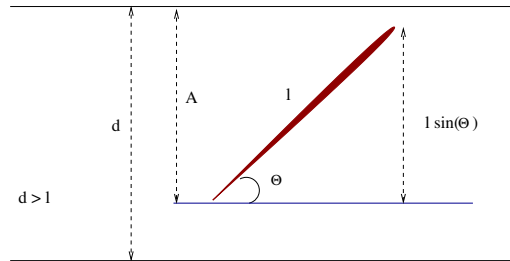


FIGURE 1 – Problème des Aiguilles de Buffon

Si la pointe est supposée fixe, la condition pour que l'aiguille croise une des lignes sera :

$$A < l \sin(\theta)$$

La position de l'aiguille relative à la ligne la plus proche est un vecteur aléatoire

$$V_{(A,\theta)} \quad A \in [0, d] \quad \text{et} \quad \theta \in [0, \pi]$$

V est distribué uniformément sur $[0, d] \times [0, \pi]$

La fonc. de densité de probabilité (PDF) de V : $\frac{1}{d \cdot \pi}$ ($= \frac{1}{d} \times \frac{1}{\pi}$)

→ N.B. : A et θ sont indépendants (d'où la multiplication).

La probabilité pour que l'aiguille croise une des lignes sera :

$$p = \int_0^\pi \int_0^{l \sin(\theta)} \frac{1}{d \cdot \pi} dA d\theta = \frac{2l}{d \cdot \pi} \quad [1]$$

Détails de la formulation de p précédente :

Le nombre de cas possibles pour la position du couple (A, θ) est représenté par l'aire du pavé :

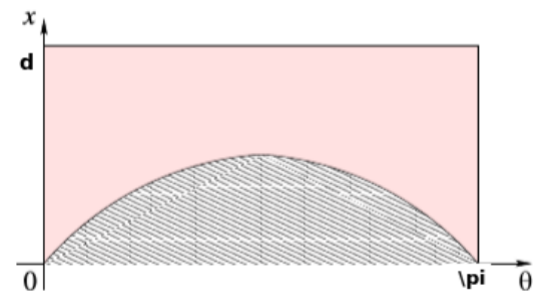
$$U = [0, d] \times [0, \pi]$$

Le nombre de cas où l'aiguille coupe une ligne horizontale est représenté par l'aire du domaine :

$$V = \{(A, \theta) \in [0, d] \times [0, \pi] : A < l \sin(\theta)\}$$

"L'aiguille coupe une ligne horizontale" avec la probabilité $Pr_{\text{croisement}}$:

$$Pr_{\text{croisement}} = \frac{\text{aire}(V)}{\text{aire}(U)} = \frac{1}{\pi \cdot d} \int_0^\pi l \sin(\theta) d\theta = \frac{2l}{d \cdot \pi}$$



☞ Mais le problème est que pour estimer π , on a besoin de π (pour les tirages aléatoires) !

III-L-1 Travail à réaliser

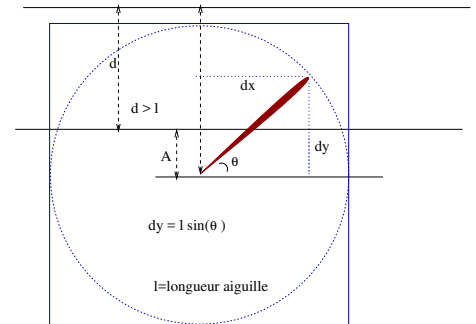
Difficulté : **/*****

- Écrire le code séquentiel d'estimation de π (en utilisant π lui même pour les tirages !)
 - Prévoir une fonction **frequence_hits(n)** qui procède au tirages par Monte Carlo et renvoie $Pr_croisement = \frac{aire(V)}{aire(U)}$ avec n tirages
 - Comme indiqué ci-dessus, la condition de croisement est $A < \ell \sin(\theta)$ où
 - $A \in [0, d]$ uniformément réparti
 - $\theta \in [0, \pi]$ uniformément réparti (oui, on utilise encore ici π)
 - ℓ et d (avec $\ell \leq d$) sont nos paramètres (resp. longueur aiguille et écart lattes parquet)
- Estimer $\pi = \frac{2.n}{Pr_croisement} \cdot \frac{\ell}{d}$

👉 **Nous allons maintenant éviter l'utilisation de π en faisant des tirages de l'angle θ .**

Pour cela, procédez comme suit :

- Quand on lance notre aiguille, on imagine un cercle de rayon ℓ dont le centre est la pointe de l'aiguille. Il nous reste à trouver les coordonnées (dx, dy) de son autre extrémité pour pouvoir obtenir l'angle θ . (la pointe centre se déplace) de sorte que (dx, dy) soit bien son autre extrémité !



- Remplacer dans votre code l'usage de $\sin(\theta)$ de la manière suivante :
 - Prévoir une fonction **sin_theta()** qui procède au tirage d'un couple $(dx, dy) \in [0, \ell] \times [0, \ell]$, un point dans le cercle supposé être centré sur la pointe de l'aiguille.
 - ➔ Il faut bien vérifier que (dx, dy) est bien dans le cercle (certains de ces points sont dans le carré $\ell \times \ell$ et pas dans le cercle.)
 - A l'aide de ces valeurs qui définissent une aiguille lancée, calculer h l'hypoténuse du triangle droit dont les côtes sont dx, dy, h ; on comprend que h est "porté" par l'aiguille sur sa longueur ℓ ($h \leq \ell$) : par abus de notation, les vecteurs \vec{h} et $\vec{\ell}$ sont confondus.
 - On peut alors calculer $\sin(\theta) = \frac{dy}{h}$ que l'on notera **sin_theta**.
 - Générer $A \in [0, d]$, ce qui place l'aiguille sur le parquet !
 - ➔ N.B. : on peut penser qu'il aurait fallu tirer (dx, dy) après le tirage aléatoire de A qui définirait les coordonnées de la tête (*tips*) de l'aiguille. Mais on peut aussi bien faire le tirage de (dx, dy) dans un repère avec tête de l'aiguille placée en $(0, 0)$ avant de translater ce repère après le tirage de A (une homothétie).
 - Si on a $A < \ell \cdot \sin_theta$, on a un *hit* de plus.
 - On procède à n itération des étapes (3) à (6) pour obtenir $Pr_croisement$
- 👉 N.B. : au lieu du cercle de rayon ℓ , on peut aussi bien utiliser un cercla unitaire (accélère légèrement les calculs) comme ci-après.

```
def calcul_PI_OK_selon_mes_slides_on_evite_use_of_PI_version_sequentielle() :
    L_lg_needle=10 # cm
    D_dist_parquet= 10 # distance entre 2 lattes du parquet
    Nb_iteration=10**6

    def tirage_dans_un_cercle_unitaire_et_sinuso_evite_PI() :
        def tirage_un_point_dans_cercle_unitaire_et_calcul_sinuso_theta() :
            while True :
                dx = random.uniform(0,1)
                dy = random.uniform(0,1)
                if dx**2 + dy**2 <= 1 : break
                sinuso_theta = dy/(math.sqrt(dx*dx+dy*dy))
            return sinuso_theta

        nb_hits=0
        for i in range(Nb_iteration) :
            # Theta=random.uniform(0,180) ne marche pas, il faut PI à la place de 180
            # Mais puisqu'on veut le sinuso(theta), on se passe de theta et on calcule sinuso(theta) à
            # l'ancienne = (cote opposé / hypotenuse)
            sinuso_theta = tirage_un_point_dans_cercle_unitaire_et_calcul_sinuso_theta()
            A=random.uniform(0,D_dist_parquet)
            if A < L_lg_needle * sinuso_theta :
                nb_hits+=1

        return nb_hits

    nb_hits=tirage_dans_un_cercle_unitaire_et_sinuso_evite_PI()

    print("nb_hits : ", nb_hits, " sur ", Nb_iteration, " essais")
    Proba=(nb_hits)/Nb_iteration # +1 pour éviter 0

    print("Pi serait : ", (2*L_lg_needle)/(D_dist_parquet*Proba))

calcul_PI_OK_selon_mes_slides_on_evite_use_of_PI_version_sequentielle()

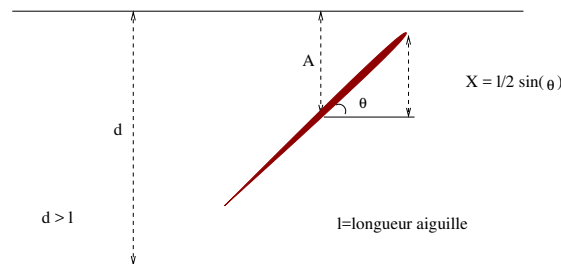
# Trace :
# nb_hits : 636512 sur 1000000 essais
# Pi serait : 3.1421245789553067
```

III-L-2 Addendum aux aiguilles de Buffon

Une autre formulation de p précédente :

On repère le point milieu de l'aiguille (facilite la formulation).

Comme dans le cas précédent, $\Theta \in [0, \pi]$ mais $A \in [0, d]$ représente la distance entre le milieu et la ligne horizontale (le plus proche).



- Cette fois, au lieu d'une double intégrale, nous utilisons une probabilité conditionnelle.

Dans un lancer donné, supposons $\Theta = \theta$ un angle particulier.

→ L'aiguille croisera une ligne horizontale si la distance A est plus petite que $X = \frac{l \cdot \sin(\theta)}{2}$ par rapport à une des 2 lignes horizontales limitrophes.

Soit E l'événement : "l'aiguille croise une ligne". On a :

$$P(E|\Theta = \theta) = \frac{\frac{l \cdot \sin(\theta)}{2}}{d} + \frac{\frac{l \cdot \sin(\theta)}{2}}{d} = \frac{l \cdot \sin(\theta)}{d}$$

→ Chaque $\frac{l \cdot \sin(\theta)}{2}$ est la proba pour une des 2 lignes horizontales,

→ la division par d permet de normaliser (nécessaire dans le cas d'une probabilité).

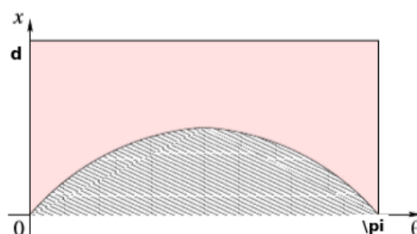
On a alors la formulation de probabilité "totale" : $P(E) = \int_0^\pi P(E|\Theta = \theta) f_\Theta(\theta) d\theta$

où f_Θ est la proba de $\theta = 1/\pi$

La probabilité de E est la loi de probabilité totale (équivalente à la double intégrale sur θ et A).

$$\text{On a } P(E) = \int_0^\pi \frac{l \cdot \sin(\theta)}{d} \frac{1}{\pi} d\theta = \frac{1}{\pi d} \int_0^\pi \sin(\theta) d\theta = \frac{2 \cdot l}{\pi d}$$

Pour estimer $P(E)$ par la méthode Monte Carlo, on pourrait procéder à un tirage aléatoire dans le rectangle $[0, \pi] \times [0, d]$ du rectangle de côtés $d \times \pi$ (lire $a = d$) :



III-L-3 Estimation Laplacienne de pi

Laplace (pour s'amuser !) a calculé une approximation de la valeur de π par ce résultat.

Soit M la variable aléatoire représentant le nombre de fois où l'aiguille a croisé une ligne ($E(M)$ l'espérance de M) :

$$\rightarrow \text{Probabilité de croiser une ligne} = \frac{E(M)}{n} \quad [2]$$

Les expressions [1] (vue ci-dessus) et [2] représentent la même probabilité : $\frac{E(M)}{n} = \frac{2l}{d\pi}$ d'où :

$$\pi = \frac{n}{E(M)} \cdot \frac{2l}{d} \quad \text{qui est un estimateur statistique de la valeur de } \pi.$$

Estimation de π :

Si on lance l'aiguille n fois, elle touchera une ligne m fois.

→ Dans $\pi = \frac{2.l.n}{E(M).d}$ on peut remplacer la variable aléatoire M par m pour obtenir une estimation de π : $\hat{\pi} \approx \frac{2.l.n}{m.d}$

En 1864, pendant sa convalescence, un certain Capitaine Fox a fait des tests et obtenu le tableau suivant :

n	m	l (cm)	d (cm)	Plateau	estimation (π)
500	236	7.5	10	stationnaire	3.1780
530	253	7.5	10	tournant	3.1423
590	939	12.5	5*	tournant	3.1416

Deux résultats importants à tirer de cette expérience :

(1) La première ligne du tableau : résultats pauvre

- Fox a fait tourner le plateau (son assise ?) entre les essais
- Cette action (confirmée par les résultats) élimine le **biais** de sa position (dans le tirages).
- Il est important d'éliminer le biais dans l'implantation de la méthode MCL. Le biais vient souvent des générateurs de nombres aléatoires utilisés (équivalent à la position du lanceur dans ces lancers).

(2) Dans ses expériences, Fox a aussi utilisé le cas $d < l$ (dernière ligne du tableau).

- L'aiguille a pu croiser plusieurs lignes (le cas * du tableau : $n=590, m=939$).
- Cette technique est aujourd'hui appelée la technique de **réduction de variance**.

Table des matières

I	Plan (de ce document)	2
II	Quelques fonctions utiles	3
III	Exercices à réaliser	4
III-A	Exercice : Course Hippique	4
III-A-1	Travail à réaliser	5
III-B	Exercice : faites des calculs	6
III-C	Gestionnaire de Billes	8
III-D	Estimation de PI	10
III-D-1	Exemple : Calcul de PI par un cercle unitaire	10
III-D-2	Principe Hit-Miss (Monte Carlo)	10
III-E	Exercice : Estimation parallèle de Pi	11
III-F	Exercice : Calcul parallèle du Merge Sort	12
III-F-1	Version séquentielle de base	12
III-G	Exercice : tri-rapide	14
III-H	Autres méthodes de calcul de PI	15
III-I	PI par la méthode arc-tangente	15
III-J	Par la méthode d'espérance	16
III-K	Par la loi Normale	16
III-L	Approximation de PI par les Aiguilles de Buffon	17
III-L-1	Travail à réaliser	18
III-L-2	Addendum aux aiguilles de Buffon	20
III-L-3	Estimation Laplacienne de pi	21