Exercices d'introduction aux calculs parallèles

Partie 1

Parallélisme et concurrence avec Python

(CPE 24-25)

(Version élèves)

Mai-Juin 2025

I Plan (de ce document)

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

☞ Chaque exercice porte un barème en nombre d'astérisques (* à *****).

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

• II) **Exercices à réaliser** avec la package multiprocessing (avec niveau de difficulté).

1. Course Hippique et affichage à l'écran (*, 3 pts)
2. Client-serveur de calculs (* à **, 3-7 pts)
3. Gestion des ressources Billes (**, 4 pts)
4. Calcul parallèle d'estimation de PI (* 3 pts)
   ➛ Il y a différentes autres techniques de calcul de PI
5. Clustering K-Means (** 5 pts)
6. Schéma Lecteurs / Rédacteurs (** 5 pts)
7. Annexes : Compléments sur le calcul de Pi
8. Annexes : Pour ceux qui n'ont pas peur de difficultés :
   1- Merge-sort par de multiples Processus
   2- Quick-sort par de multiples Processus

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

Pour ceux qui travaillent sur un MAC (sous macos) :
Pour avoir des résultats corrects et équivalents aux versions sous Linux, placer dans votre code (dans la section "main") :

```
if __name__ == "__main__" :
        import platform
        if platform.system() == 'Darwin'
            mp.set_start_method('fork')
        votre code ...
```

II Quelques fonctions utiles

Dans la package **multiprocessing**, on dispose de quelques fonctions utilitaires. Voir également le cours 2 sur le multiprocessing pour plus de détails.

Voir aussi https ://docs.python.org/3/library/multiprocessing.html

- ***multiprocessing.cpu_count()*** : renvoie le nombre de CPU (processus matériellement parallèles possibles) sur votre ordinateur.
- ***multiprocessing.active_children()*** : renvoie la liste des processus fils encore en vie du processus courant.
- ***multiprocessing.current_process()*** : renvoie l'objet Processus correspondant a processus courant.
- ***multiprocessing.parent_process()*** : renvoie un object Process qui correspond au parent du processsus courant. Pour le processus principal (main), le processus parent est None.
- Après avoir créé un processus par *pid = multiprocessing.Process()*, les méthodes les plus importants sont :

  pid.start(), pid.run(), pid.join(), pid.terminate(), pid.kill(), pid.is_alive(), ...
- Pour accéder au nom, pid ou exitcode d'un processus, on peut utiliser les attributs **name, pid , exitcode.**

  Par exemple *multiprocessing.current_process().name / .pid / .exitcode, ...*
- voir les autres méthodes utilitaires en page

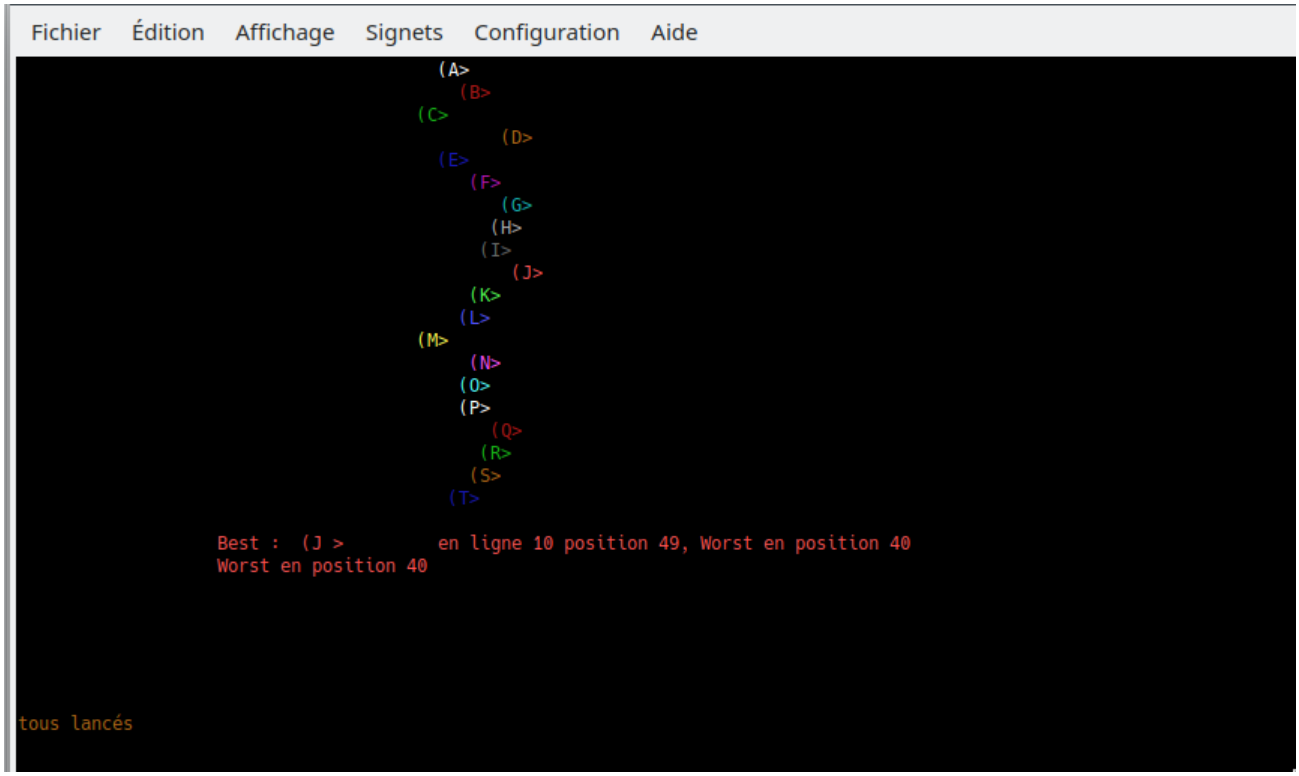
  https ://docs.python.org/3.8/library/multiprocessing.html ?highlight=queue#multiprocessing.Queue

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*).



Pour ne pas compliquer l'exercice, on n'aura pas recours aux outils d'affichages graphiques (Tkinter, QT, ...). 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 en rouge ci-dessus ("Best : .." et les suivantes).

Ci-joint, une version basique de cette course pour 2 chevaux.

☞ Ce code ne devrait pas fonctionner sous Windows de Microsoft (qui ne dispose pas d'un terminal VT100 ou Xterm comme sous Linux).

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

La première partie du code ci-dessous décrit les codes d'échappement qui permettent de faire différentes opérations sur un terminal telle que l'effacement d'écran, l'effacement (du reste) d'une ligne, se placer sur une case (ligne / colonne) du terminal, etc.

```
# Course Hippique (version élèves)
# Version très basique, sans mutex sur l'écran, sans arbitre, sans annoncer le gagnant, ... ...
# ----------------------------------------
# Quelques codes d'échappement (tous ne sont pas utilisés)
CLEARSCR="\x1B[2J\x1B[;H"        #  Clear SCReen
CLEAREOS = "\x1B[J"              #  Clear End Of Screen
CLEARELN = "\x1B[2K"             #  Clear Entire LiNe
CLEARCUP = "\x1B[1J"             #  Clear Curseur UP
GOTOYX   = "\x1B[%.2d;%.2dH"     #  ('H' ou 'f') : Goto at (y,x), voir le code

DELAFCURSOR = "\x1B[K"           #  effacer après la position du curseur
CRLF  = "\r\n"                   #  Retour à la ligne

# VT100 : Actions sur le curseur
CURSON   = "\x1B[?25h"           #  Curseur visible
CURSOFF  = "\x1B[?25l"           #  Curseur invisible

# VT100 : Actions sur les caractères affichables
NORMAL = "\x1B[0m"               #  Normal
BOLD = "\x1B[1m"                 #  Gras
UNDERLINE = "\x1B[4m"            #  Souligné


# VT100 : Couleurs : "22" pour normal intensity
CL_BLACK="\033[22;30m"           #  Noir. NE PAS UTILISER. On verra rien !!
CL_RED="\033[22;31m"             #  Rouge
CL_GREEN="\033[22;32m"           #  Vert
CL_BROWN = "\033[22;33m"         #  Brun
CL_BLUE="\033[22;34m"            #  Bleu
CL_MAGENTA="\033[22;35m"         #  Magenta
CL_CYAN="\033[22;36m"            #  Cyan
CL_GRAY="\033[22;37m"            #  Gris

# "01" pour quoi ? (bold ?)
CL_DARKGRAY="\033[01;30m"        #  Gris foncé
CL_LIGHTRED="\033[01;31m"        #  Rouge clair
CL_LIGHTGREEN="\033[01;32m"      #  Vert clair
CL_YELLOW="\033[01;33m"          #  Jaune
CL_LIGHTBLU= "\033[01;34m"       #  Bleu clair
CL_LIGHTMAGENTA="\033[01;35m"    #  Magenta clair
CL_LIGHTCYAN="\033[01;36m"       #  Cyan clair
CL_WHITE="\033[01;37m"           #  Blanc

#----------------------------------------------------------
import multiprocessing as mp
import os, time,math, random, sys, ctypes, signal

# Une liste de couleurs à affecter aléatoirement aux chevaux
lyst_colors=[CL_WHITE, CL_RED, CL_GREEN, CL_BROWN , CL_BLUE, CL_MAGENTA, CL_CYAN, CL_GRAY,
        CL_DARKGRAY, CL_LIGHTRED, CL_LIGHTGREEN,  CL_LIGHTBLU, CL_YELLOW, CL_LIGHTMAGENTA, CL_LIGHTCYAN]

def effacer_ecran() : print(CLEARSCR,end='')
def erase_line_from_beg_to_curs() : print("\033[1K",end='')
def curseur_invisible() : print(CURSOFF,end='')
def curseur_visible() : print(CURSON,end='')
def move_to(lig, col) : print("\033[" + str(lig) + ";" + str(col) + "f",end='')

def en_couleur(Coul) : print(Coul,end='')
def en_rouge() : print(CL_RED,end='') # Un exemple !
def erase_line() : print(CLEARELN,end='')

# La tache d'un cheval
def un_cheval(ma_ligne : int, keep_running) : # ma_ligne commence à 0
    col=1

    while col < LONGEUR_COURSE and keep_running.value :
        move_to(ma_ligne+1,col)        # pour effacer toute ma ligne
        erase_line_from_beg_to_curs()
        en_couleur(lyst_colors[ma_ligne%len(lyst_colors)])
        print('('+chr(ord('A')+ma_ligne)+'>')

        col+=1
```

```
        time.sleep(0.1 * random.randint(1,5))

    # Le premier arrivée gèle la course !
    # J'ai fini, je me dis à tout le monde
    keep_running.value=False

#-----------------------------------------------
def detourner_signal(signum, stack_frame) :
    move_to(24, 1)
    erase_line()
    move_to(24, 1)
    curseur_visible()
    print("La course est interrompu ...")
    exit(0)
# -----------------------------------------------
# La partie principale :
if __name__ == "__main__" :

    import platform
    if platform.system() == "Darwin" :
        mp.set_start_method('fork') # Nécessaire sous macos, OK pour Linux (voir le fichier des sujets)

    LONGEUR_COURSE = 50 # Tout le monde aura la même copie (donc no need to have a 'value')
    keep_running=mp.Value(ctypes.c_bool, True)

    Nb_process=20
    mes_process = [0 for i in range(Nb_process)]


    effacer_ecran()
    curseur_invisible()

    # Détournement d'interruption
    signal.signal(signal.SIGINT, detourner_signal) # CTRL_C_EVENT   ?

    for i in range(Nb_process):  # Lancer   Nb_process  processus
        mes_process[i] = mp.Process(target=un_cheval, args= (i,keep_running,))
        mes_process[i].start()

    move_to(Nb_process+10, 1)
    print("tous lancés, CTRL-C arrêtera la course ...")

    for i in range(Nb_process): mes_process[i].join()

    move_to(24, 1)
    curseur_visible()
    print("Fini ... ", flush=True)
```

III-*A-1* *Travail à réaliser*

Compléter ce code pour réaliser les points suivants :

- Ajouter un processus *arbitre* qui affiche en permanence le cheval qui est en tête ainsi que celui qui est dernier comme dans la figure ci-dessus. A la fin de la course, il affichera les éventuels canassons ex aequos.

  Pour cela, créer un processus supplémentaire et lui associer la fonction "arbitre". L'arbitre partage un **tableau** avec les chevaux dans lequel il peut trouver la position de chaque cheval à tout moment.
  ☞ Cette information est donnée par chaque cheval qui inscrit sa position dans le tableau partagé à la case du qui lui est dédiée.
  Cette action de chaque cheval simule la situation où un observateur placé au bord du terrain note la position de chaque cheval et l'inscrit dans un tableau. Ce tableau est consulté par l'arbitre pour décider qui est en tête de la course.

- 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 !) :

```
  ________/
 /----- _.\
/|______/
 /\ /\
```

  ☞ Attention : le dessin est sur plusieurs lignes. Donc, un cheval occupera plusieurs lignes. Ainsi, le Principe "une ligne par cheval" doit être abandonné. Avec le dessin ci-dessus, chaque cheval, occupera 6 lignes !

- Vous constatez que les affichages à l'écran ne sont pas en exclusion mutuelle. Réglez ce problème en utilisant un mutex.

III-*B* *Exercice : faites des calculs*

Difficulté : ** **(3 à 7 points)**

Problème traité en cours sous forme simplifiée et triviale : échange entre un client demandeur et 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 calculteurs.** (**3 points**)

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

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

(1) Réalisez cette version.

(2) Que faire si l'on doit vérifier la justesse de l'expression. Par exemple, rejeter une expression telle que "$2 + 3 -$ "

• **Version 2 : m demandeurs, n calculteurs.** (**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.

Dans un premier temps, on peut gérer une Queue par demandeur. Lorsque un résultat est calculé et déposé par un processus calculateur, le demandeur peut récupérer ce résultat dans la file (Queue) qui lui est dédiée.

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

(4) Comment faire si ne souhaite pas créer une Queue par demandeur ?

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

• 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()** et **os.pipe()** où 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 (en particulier cours 2) pour des détails et exemples de Queue.

- Le fils (qui fait les calculs)

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

    while True:
        cmd = os.read(rpipe_commande, 32)
        print("Le fils a recu ", 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_calculette(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 recu ", res)
            print('-' * 60)
            time.sleep(1)
```

- Trace :

```
Le père va demander à faire :  5/9
Bonjour du Fils 12851
Le fils a recu  5/9
Dans fils, le résultat = 0.5555555555555556
Le fils a envoyé 0.5555555555555556
Le Pere a recu  0.5555555555555556
------------------------------------------------------------
Le père va demander à faire :  5/2
Le fils a recu  5/2
Dans fils, le résultat = 2.5
Le fils a envoyé 2.5
Le Pere a recu  2.5
------------------------------------------------------------
Le père va demander à faire :  8*6
Le fils a recu  8*6
Dans fils, le résultat = 48
Le fils a envoyé 48
Le Pere a recu  48
------------------------------------------------------------
...
```

III-C *Gestionnaire de Billes*

Difficulté : ** , **(4 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 4 processus.

  Il crée également un processus *controleur* qui vérifie en permanence si le nombre de Billes disponible est dans l'intervalle $[0..nb\_max\_billes]$
- 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

Psoeudo algorithmes :

• Main :

```
MAIN :
   Creer 4 processus travailleurs (avec mp.Process)
   Lancer ces 4 processus
   Creer un processus controleur
   Lancer controleur
   ... tourner les pouces un peu ...
   Attendre les fin des 4 processus
   Terimer le processus controleur
```

• Travailleur :

```
Travailleur(k_bills):
   Iterer m fois :
      demander k_bills
      simuler le travail avec un delai (sleep k_bills sec.)
      renre k_bills
```

• Demander k billes (attente passive) :

```
Demander(k_bills): # Dans ce psoeudo–code, la gestion deu verrou/sémaphore est absente
   demander verrou (exclusion mutuelle)
   Tantque nbr_disponible_billes < k_bills :      # Ce test doit être fait dans une Section Critique (SC)
      libérer verrou
      se bloquer (sur un semaphore d'attente)  <– Ne pas oublier de libérer le verrou avant de vous mettre en attente !
      demander verrou (pour accéder à la variable partagée nbr_disponible_billes)
   Dans une section critique en exclusion mutuelle :
      nbr_disponible_billes = nbr_disponible_billes – k_bills
```

• rendre k billes :

```
rendre(k_bills):
   Dans une section critique en exclusion mutuelle :
      nbr_disponible_billes = nbr_disponible_billes + k_bills
   libérer ceux qui sont bloqués  (sur un semaphore d'attente)  # voir "demander"
```

☞ Noter que **Demander(k_bills)** sera équivalent à **sem.acquire(k_jetons)** mais cette possibilité n'existe pas dans le package multiprocessing !

☞ De même pour **rendre()** et **release()**

• Contrôleur :

```
Controleur(max_billes):
   Iterer toujours :
      Dans une SC :
         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 le 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]

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

```
Demander_attente_active (k_bills): # Dans ce psoeudo-code, la gestion deu verrou/sémaphore est absente
   demander verrou sur nb_billes
   Tantque nbr_disponible_billes < k_bills :
      libérer verrou sur nb_billes
      attednre un peu # sleep(0.5) par exemple
      demander verrou sur nb_billes
   nbr_disponible_billes = nbr_disponible_billes – k_bills
```

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_bills):
   Dans une SC :
      nbr_disponible_billes = nbr_disponible_billes + k_bills
```

☞ 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 par exemple, la *condition* peut être la réalisation de *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*

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 $\frac{1}{\sqrt{N}}$.

A l'aide de l'exemple (solution séquentielle) ci-dessous, réaliser la versions multi-processus. On comparera ensuite les temps de calculs pour montrer le gain.

• La version séquentielle de la solution :

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

if __name__ == "__main__" :
    nb_total_iteration = 10000000 # Nombre d'essai pour l'estimation
    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 :  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 IV en page 19 pour les autres méthodes de calcul de Pi.

III-*F* *Exercice : K-means*

Difficulté : ** , **(5 points)**

L'algorithme K-Mens est une méthode de clustering (classification). Il s'agira de créer k groupes les plus homogènes parmi un échantillon d'observations.

On peut par exemple scinder un ensemble d'élèves en 2 groupes selon certains critères. Ce qui permet cette séparation en k groupes est l'ensemble des caractéristiques de chaque observation. Par exemple, pour les élèves, on peut disposer de l'age, la taille, la couleur de vêtement, la longueur de cheveux, le poids, la pointure des chaussures (!), les habitudes, etc [2].

Dans l'exercice ci-dessous, on travaille avec les coordonnées d'un ensemble de points dans un plan 2D. Ici, $k = 2$. La séparation en 2 groupes sera une séparation "géométrique" (donc en fonction des coordonnées)

On se donne une liste Lst de points $p_i : (x_i, y_i)$ de coordonnées 2D dans le plan. Ces points sont échantillonnés aléatoirement dans le plan 2D $[(0, 0) : (100, 100)]$

Procéder comme suit pour créer 2 paquets (2 clusters) avec ces points :

- 1- Fixer aléatoirement deux des points m_1 et m_2 de la liste Lst. Considérer m_1 et m_2 comme des centres (d'attraction) des autres points. Chaque centre attirera les points qui lui sont les plus proches.
- 2- Faire deux paquets (listes) $L1$ et $L2$ telles que $L1$ contienne les points p_i plus proche de m_1 que de m_2, L_2 contiendra les points p_j qui lui sont plus proches (que du centre m_1). La distance utilisée sera *Euclidienne*. A ce stade, nous avons 2 groupes de points.
- 3- Recalculer les nouveaux centre m'_1 et m'_2 dans chaque groupes (*clusters*) L_1 et L_2 en fonction de leur distance par rapport à m'_1 et m'_2. Ainsi, m_1 et m_2 auront changé de coordonnées pour devenir m'_1 et m'_2.
- 4- Expectation : Regrouper tous les points et refaire l'étape 2 et scinder ensemble les points en fonction de leur distance par rapport à chaque nouveau centre m'_1 et m'_2. Cela modifiera les liste L_1 et L_2 pour donner L'_1 et L'_2.
- 5- Maximisation : Répéter l'étape 3 en recalculant les nouveaux centres m_1" et m_2" de L'_1 et L'_2.
- 6- Recommencer à l'étape 4 jusqu'à ce que les centre ne bougent plus (à ϵ près)

Vous avez ainsi obtenu deux paquets séparés (deux *clusters*) relativement distincts avec des centres respectifs m_1 et $m2$. Afficher ces centres et les contenu des 2 paquets ainsi obtenus.

Mesurez la qualité de votre (regroupement) *clustering* par $SSE = \sum_{j \in 1..k} \left(\sum_{p_i \in L_j} (p_i - m_j)^2 \right)$ (erreur inter groupe).

2. Ces caractéristiques sont censées discriminer une fille d'un garçon!

Relancez plusieurs fois votre algorithme (sur le même échantillon) et désignez le meilleur *clustering*. Les **meilleurs clusterings** minimisent SSE tout en maximisant l'écart entre les groupes.

Pour ce dernier critère, il existe plusieurs solutions. Pour simplifier, tout en minimisant SSE, nous allons simplement maximiser la distance entre les centres des clusters finaux :

$$SSD = \sum_{i,j \in 1..k, i>j} (m_i - m_j)^2 \text{ (erreur intra groupes).}$$

Le pseudo algorithme de K-means :

```
def K_means() :
   Lst = échantillonner aléatoirement 100 points   # se limiter à [(0,0) .. (100,100)]
   m_1 = un élement  de Lst
   m_2 = un autre élément aléatoire de Lst (différent de m_1)
   fini = False
   old_m1 = m_1; old_m2 = m_2
   Répéter jusqu'à fini :
      # On utilise dist(.,.) Euclidienne
      L_1 = les éléments de Lst le plus proche de m_1 (que de m_2)
      L_2 = les éléments de Lst le plus proche de m_2 (que de m_1)
      m_1 = le centre de L_1      # m_1 = la moyenne de L_1
                  # m_1 ne sera pas forcément un des points de L_1
      m_2 = le centre de L_2
      Si dist(m_1, old_m1) < epsilon OU dist(m_2, old_m2) < epsilon : fini = True
      old_m1 = m_1; old_m2 = m_2
   Fin Répéter
# NB : "OU" car si un des centres ne se modifie plus, ses points ne changent plus et donc l'autre centre ne bougera plus
```

1 • Réaliser d'abord la version séquentielle de l'algorithme K-Means.

Rappel : pour simplifier, les données sont des points générés aléatoirement dans un plan 2D.

Coder l'algorithme de K-Means.

Pour obtenir un bonus, ajouter le code pour un affichage (plot) visuel des points de chaque cluster.

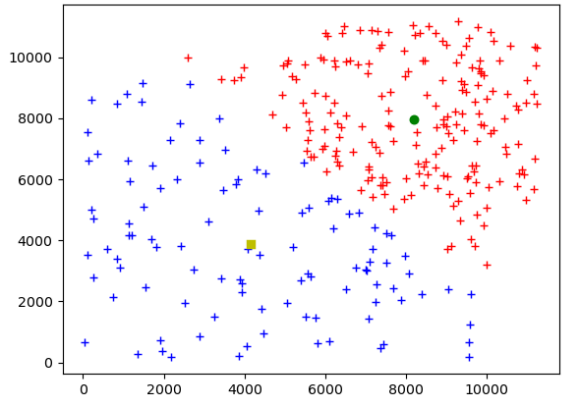


FIGURE 1 – Résultats d'un exemple de clustering 2D avec k=2 (les 2 points : les centres des 2 clusters)

2 • Modifier votre code pour obtenir $k > 2$ (par exemple 4) clusters.

3 • A l'aide de la version précédente, réaliser une **version parallèle** pour obtenir k clusters avec les mêmes possibilités de visualisation.

A titre d'indication, voici le pseudo-code des fonctions principales. La fonction "main" est la fonction principale que l'on applique aux données X étant donné le nombre k de clusters.

☞ Vous aurez probablement (?) besoin d'utiliser un Pool de processus (voir cours 2 pour des exemples).

```
def centroid_init(X = Points, k)  :
   Désignez k médoides : m1..mk (choisir mj (j dans 2..k les plus éloignes de mi, i=1..j−1)

def assigner_points_aux_clusters(X, les_k_centres) :
   liste_de_points_des_clusters = Attribuer un cluster à chaque point (dans X) selon la distance Euclidienne aux centres
   Pour cela, pour chaque point p , calculer la distance à chaque les_k_centres[num_cluster] et noter cette distance
   Parmi ces distances prendre le minumum :  num_cluster sera le cluster du point p

   De cette manière, on dispatche les points et chaque point a maintenant un cluster proche

def main(X = les points, k):
      cluster_centers = centroid_init(X, k)
      for i in range(max_iter):
         splitted_X = Scinder les points dans k groupes selon leur proximité aux cluster_centers

         # En Parallèle, attribuer chaque points de X à un cluster
         p=Pool()
         result=p.map(assigner_points_aux_clusters, liste de couples (splitted_X,  cluster_centers))
         p.close()
         # Fusionner les resultats
         p.join()

         Somme_dans_chaque_cluster = Dans chaque cluster, calculer la somme des points (dimension−wise)
         new_cluster_centers = Dans chaque cluster, calculer le nouveau centre avec Somme_dans_chaque_cluster/nb_points_du_cluster
         Arrêter les itérations si new_cluster_centers == cluster_centers
         Sinon  : cluster_centers = new_cluster_centers
```

III-*G* *Exercice : schéma Lecteurs/Rédacteurs*

Difficulté : ** , **(5 points)**

Un groupe d'élèves en projet a réalisé un site qui permet de manipuler (éditer, lire, imprimer, ...) des documents.

Pour simplifier, on se donne un seul document.

Chacun des élèves du groupe peut décider de lire ce document ou de le modifier.

Supposons que pour une simulation, on a n rédacteurs (qui modifient le document) et de m lecteurs de ce document. Le groupe est donc composé de $n + m$ élèves. Par exemple, $n = 2$ et $m = 4$.

Un document étant déposé, on ne peut avoir plus d'une rédaction à un moment donné mais il est possible d'avoir plusieurs lecteurs simultanées (en même temps). Bien entendu, la rédaction exclue toute lecture et inversement.

Par contre, la priorité est donnée aux rédacteurs. Dès qu'en rédacteur souhaite modifier le document (il manifeste d'abord ce souhait en ajoutant p. ex. +1 sur la variable *nb_demande_de_redaction*), plus aucun lecteur supplémentaire ne peut lire le document jusqu'à la fin des rédactions (jusque *nb_demande_de_redaction==0*).

Le fonctionnement de chaque "rédacteur" est le suivant :

• **Rédacteur**

Répéter tant que pas fini :

- Ajouter +1 au `nbr_demandes_de_rédaction` [manifeste son souhait de modification]
- N'accepter plus aucune lecture supplémentaire
- Attendre la fin des lectures en cours
- Attendre la fin d'une rédaction éventuelle en cours (d'un autre rédacteur) [à l'aide de *nb_demande_de_redaction*]
- Procéder à la rédaction (simuler par une attente quelques secondes et l'affichage d'un message)
- Signaler la fin de la rédaction en faisant -1 sur `nbr_demandes_de_rédaction`

➛ Ce qui a pour effet de réveiller un rédacteur en attente éventuelle

➛ Si pas de demande de rédaction (`nbr_demandes_de_rédaction=0`), de réveiller des lecteurs éventuels en attente

Le fonctionnement de chaque "rédacteur" est le suivant :

• **Lecteur**

Répéter tant que pas fini :

- Tant que `nbr_demandes_de_rédaction > 0` : attendre
- Procéder à la lecture (attendre quelques secondes et afficher un message)

➛ Rappelons que les lectures parallèles sont possibles

➛ Mais toutes les tentatives de lecture seront bloquées si un rédacteur s'est manifesté pendant nos lectures (ce rédacteur a fait +1 sur `nbr_demandes_de_rédaction`)

- Signaler la fin de la lecture

➛ Ce qui a pour effet de réveiller un autre rédacteur en attente éventuelle si plus aucune lecture en cours.

☞ Si vous le souhaitez, vous pouvez gérer le nombre de lectures en parallèle (p. ex. `nbr_lectures_en_cours`).

Exemple de scénario avec $n = 2$ rédacteurs et $m = 4$ lecteurs : 6 processus créés et lancés.

- Une demande de rédaction (R_1) arrive (R_1 fait +1 sur `nbr_demandes_de_rédaction`).
- Elle a pour effet de bloquer toutes les demandes de lecture.
- Puisque pas de rédaction en cours, R_1 entre en rédaction
- Deux demandes de lecture (L_1, L_2) arrivent. Elles sont bloquées jusqu'à la fin de R_1 à condition qu'aucune autre demande de rédaction ne survienne (qui aura pour effet d'augmenter `nbr_demandes_de_rédaction`).
- R_1 termine
- L_1 entre en lecture ; L_2 entre en lecture
- Une demande de rédaction (R_2) arrive. Elle est mise en attente car des lectures sont en cours.
- Deux demandes de lecture (L_3, L_4) arrivent mais ces lecteurs ne peuvent pas entrer car R_2 a fait une demande.
- Fin de L_1 et de L_2 (faire varier tous les délais de lecture/rédaction par un sleep de valeur différente)
- R_2 entre en rédaction
- ....

IV Annexes : 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.

IV-*A* *PI par la méthode arc-tangente*

☞ Vu en cours.

Méthode arc-tangente :

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

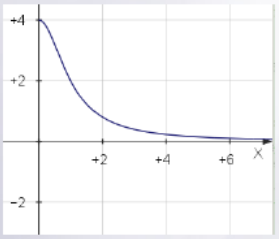
$$\pi \approx \int_0^1 \frac{4}{1+x^2}\,dx \qquad \text{ou la version discrète} \qquad \pi \approx 1/n \sum_1^n \frac{4}{1+x_i^2}$$

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 :

$$\sum_1^n \frac{4}{1+x_i^2} \approx \sum_{i=1}^{n} \frac{4}{1+(\frac{i-0.5}{n})^2} = \sum_{i=0}^{n-1} \frac{4}{1+(\frac{i+0.5}{n})^2}$$

➛ $(\frac{i-0.5}{n})$ 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
```

IV-*B* *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

IV-*C* *Par la loi Normale*

Difficulté : */*****

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

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

☞ On peut utiliser une variable centrée réduite ($\sigma = 1$, $\mu = 0$) pour simplifier les calculs avec $f(x) = \dfrac{1}{\sqrt{2\pi}} e^{-\dfrac{x^2}{2}}$ d'où $\displaystyle\int_0^{\infty} f(x)dx = \frac{1}{2}$

IV-*D* *Approximation de PI 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.

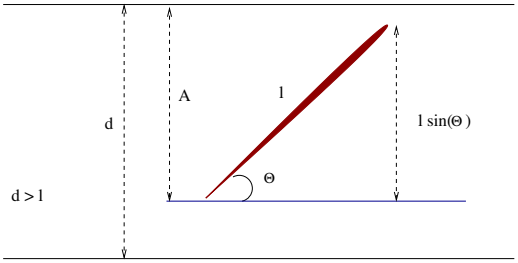


FIGURE 2 – Problème des Aiguilles de Buffon

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

$$\boxed{A < l\ sin(\theta)}$$

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

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

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

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

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

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

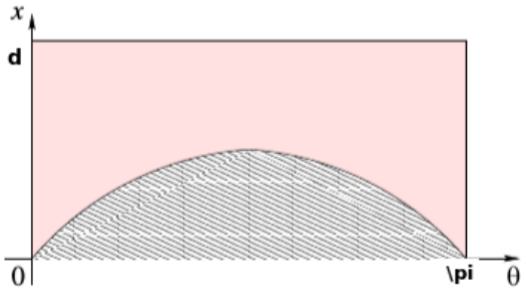
$$p = \int_0^{\pi} \int_0^{lsin(\theta)} \frac{1}{d.\pi}\, dA\, d\theta = \frac{2l}{d.\pi} \qquad [1]$$

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 $Pr\_croisement$:

$$Pr\_croisement = \frac{aire(V)}{aire(U)} = \frac{1}{\pi.d} \int_0^{\pi} l.sin(\theta)\, d\theta = \frac{2l}{d.\pi}$$

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

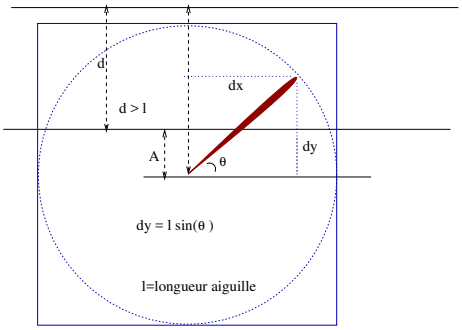
IV-*D-1 Travail à réaliser*

Difficulté : **/*****

1. 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 $Pr\_croisement = \dfrac{aire(V)}{aire(U)}$ 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)
2. Estimer $\pi = \dfrac{2.n}{Pr\_croisement} . \dfrac{\ell}{d}$

☞ **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 !



3. 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 carrée $\ell \times \ell$ 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 $\overrightarrow{h}$ et $\overrightarrow{\ell}$ sont confondus.
   On peut alors calculer $sin(\theta) = \dfrac{dy}{h}$ que l'on notera sin_theta.
4. 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).
5. Si on a $A < \ell\ .\ sin\_theta$, on un *hit* de plus.
6. 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.

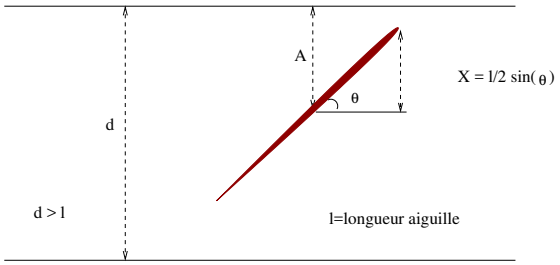
```
def calcul_PI_OK_selon_mes_slides_on_evite_use_of_PI_version_sequentielle() :
    L_lg_needle=10 # cm
    D_dist_parquet= 10 # distance entre 2 lattes du parquet
    Nb_iteration=10**6

    def tirage_dans_un_cercle_unitaire_et_sinus_evite_PI() :
        def tirage_un_point_dans_cercle_unitaire_et_calcul_sinus_theta() :
            while True :
                dx = random.uniform(0,1)
                dy = random.uniform(0,1)
                if dx**2 + dy**2 <= 1 : break
            sinus_theta = dy/(math.sqrt(dx*dx+dy*dy))
            return sinus_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 sinuus(theta), on se passe de theta et on caclcule sinus(theta) à
            # l'ancienne = (cote_opposé / hypothenuse)
            sinus_theta = tirage_un_point_dans_cercle_unitaire_et_calcul_sinus_theta()
            A=random.uniform(0,D_dist_parquet)
            if A <  L_lg_needle * sinus_theta :
                nb_hits+=1

        return nb_hits

    nb_hits=tirage_dans_un_cercle_unitaire_et_sinus_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
```

IV-*D-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 $X = \dfrac{l.sin(\theta)}{2}$ par rapport à une des 2 lignes horizontales limitrophes.

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

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

➙ Chaque $\dfrac{l.sin(\theta)}{2}$ est la proba pour une des 2 lignes horizontales,

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

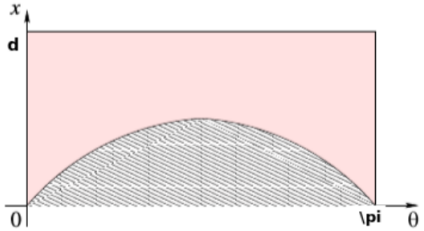
On a alors la formulation de probabilité "totale" : $P(E) = \displaystyle\int_0^{\pi} P(E|\Theta = \theta)\ f_{\Theta}(\theta)\ d\theta$

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).

On a $\quad P(E) = \displaystyle\int_0^{\pi} \frac{l.sin(\theta)}{d} \frac{1}{\pi}\, d\theta \ = \frac{1}{\pi d} \int_0^{\pi} sin(\theta)\, d\theta = \frac{2.l}{\pi d}$

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 $a = d$) :



IV-*D-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'espérence de M) :

→ Probabilité de croiser une ligne = $\dfrac{E(M)}{n}$ [2]

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

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

Estimation de π :

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

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

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**.

V Annexes : Exercices optionnels et en Bonus

V-A *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 tableaux 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)
```

V-B *Exercice : Calcul parallèle du Merge Sort*

Difficulté : *** **(5 points)**

Le principe du tri fusion : pour trier un tableaux 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 trié et leur fusion tient compte de cela.

V-B-1 *Version séquentielle de base*

• Pour commencer, voyons la version de base du tri-fusion

```
import math, random
from array import array

def merge(left, right):
    tableau = array('i', [])  # tableau vide qui reçoit les résultats
    while len(left) > 0 and len(right) > 0:
        if left[0] < right[0]: tableau.append(left.pop(0))
        else: tableau.append(right.pop(0))

    tableau += left + right
    return tableau

def merge_sort(Tableau):
    length_Tableau = len(Tableau)
    if length_Tableau <= 1: return Tableau
    mid = length_Tableau // 2
    tab_left = Tableau[0:mid]
    tab_right = Tableau[mid:]
    tab_left = merge_sort(tab_left)
    tab_right = merge_sort(tab_right)
    return merge(tab_left, tab_right)

def version_de_base(N):
    Tab = array('i', [random.randint(0, 2 * N) for _ in range(N)])
    print("Avant : ", Tab)
    start=time.time()
    Tab = merge_sort(Tab)
    end=time.time()
    print("Après : ", Tab)
    print("Le temps avec 1 seul Process = %f pour un tableau de %d eles " % ((end-start)*1000, N))

    print("Vérifions que le tri est correct --> ", end="")
    try :
        assert(all([(Tab[i] <= Tab[i+1]) for i in range(N-1)]))
        print("Le tri est OK !")
    except : print(" Le tri n'a pas marché !")
```

Les résultats pour un tableau de 1000 éléments :

```
N = 1000
version_de_base(N)

# Le temps avec 1 seul Process = 10.593414 pour un tableau de 1000 eles
# Le tableau puis
# Vérifions que le tri est correct --> Le tri est OK !
```

V-B-2 *Exercice*

Exercice-4 : transformer cette version de base en une version de tri sur-place :

➛ Ne pas découper le tableau à trier en sous tableaux mais travailler avec des 'tranches' de ce dernier. Par conséquent, un sous tableau de la version de base ci-dessus sera repéré par deux indices début - fin (= une zone du tableau global).

Exercice-5 : Écrire une version avec des Processus de cette méthode de tri : 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])
```

Table des matières