Introduction aux calculs parallèles

Exercices avec Python

(CPE 19-20)

(Version élèves)

28 novembre 2019

17:58

I Plan

- Introduction :
- Calcul concurrent
- Section Critique & *Race Condition*
- Intérêt d'un verrou pour la protection d'une *section critique* (de calculs)

  ➛ Illustration via des exemples

Exercices

1. Calcul parallèle de la somme des éléments d'un (grand) vecteur / liste de nombres.
2. Calcul parallèle de PI (via un cercle unitaire)

   ➛ Il y a différentes autres techniques de calcul de PI
3. Merge-sort par de multiples Processus
4. Quick-sort par de multiples Processus
5. Une version multi-processus de la méthode Newton
6. ...

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

II Section critique et Exclusion mutuelle

• Pour constater l'intérêt d'une exclusion mutuelle, on observe dans un premier temps comment deux processus exécutés en parallèle peuvent incrémenter une variable partagée.

• Soit deux processus qui incrément chacun une même variable partagée (et non protégée) appelée ici *variable_partagee*. Pour qu'elle soit partagée, cette variable est déclarée du type entier initialisée à 0 via **variable_partagee = mp.Value('i',0)**.

```
import multiprocessing as mp

# Incrémentation sans protéger la variable partagée
def count1_on_se_marche_sur_les_pieds(nb_iterations):
    """ Chacun écrit à son rythme (non protégée)"""
    global variable_partagee
    for i in range(nb_iterations):
        variable_partagee.value += 1

#------------ PARTIE principale (le point d'entrée de cet exemple -------
if __name__ == '__main__' :
    nb_iterations = 5000
    # La variable partagée
    variable_partagee = mp.Value('i',0)  # ce sera un entier initialisé à 0
    print("la valeur de variable_partagee AVANT les incrémentations : ", variable_partagee.value)
    # On crée 2 process
    pid1=mp.Process(target=count1_on_se_marche_sur_les_pieds, args=(nb_iterations,)); pid1.start()
    pid2=mp.Process(target=count1_on_se_marche_sur_les_pieds, args=(nb_iterations,)); pid2.start()
    pid1.join(); pid2.join()

    print("la valeur de variable_partagee APRES les incrémentations %d (attendu %d) "% (variable_partagee.value,nb_iterations*2))
```

• Trace : on teste deux / trois fois ce code pour constater le non-déterminisme des résultats.

```
la valeur de variable_partagee AVANT les incrémentations :  0
la valeur de variable_partagee APRES les incrémentations 5251 (attendu 10000)

la valeur de variable_partagee AVANT les incrémentations :  0
la valeur de variable_partagee APRES les incrémentations 51280 (attendu 10000)

la valeur de variable_partagee AVANT les incrémentations :  0

la valeur de variable_partagee APRES les incrémentations 5389 (attendu 10000)
...
```

• On constate que dans de nombreux cas, l'incrémentation effectuée par l'un des processus est interrompue par l'autre.

☞ Expliquer **mp.Value** et l'accès à la valeur de la variable par **value**.

Pourquoi les incrémentations n'ont pas toutes lieu ?

La raison est que certaines incrémentations sont interrompues [1] pendant l'incrémentation.

Pour comprendre cela, il faut observer les endroits (instructions machines) où cette incrémentation peut être interrompue.

Pour comprendre ce phénomène, définissons une simple fonction d'incrémentation d'une variable x puis demandons à Python la traduction (en byte-code ou pseudo-assembleur) de cette simple fonction d'incrémentation effectuée par $x = x + 1$.

1. la tâche qui incrémente se voit retirée le processeur par l'ordonnanceur du système

```
def incremeter(x) :
    x = x + 1
```

Puis

```
import dis      # pour voir les détails

dis.dis(incremeter)
```

On obtient :

```
0      LOAD_FAST             0 (x)        # x est à un décalage de 0 p/r au début de la zone des variables
3      LOAD_CONST            1 (1)          # charger la constate 1
6      BINARY_ADD                        # additionner
7      STORE_FAST            0 (x)          # stocker le résultat dans x


# Epilogue : préparer en renvoyer None (Une fonction Pythone renvoie tjs qq chose !)
10     LOAD_CONST            0 (None)
13     RETURN_VALUE
```

Le processus qui exécute ces instructions peut être interrompu à la fin de chacune.

➛ Expliquer l'anatomie d'une incrémentation.

Si nous ne voulons pas être interrompu[2] pendant cette incrémentation, on devrait considérer l'incrémentation ($x = x + 1$) comme une action critique (sensible)[3].

• On appellera une **section critique** (SC) la zone du code qui doit être accédée en exclusion mutuelle. Dans notre exemple, la SC est clairement `variable_partagee.value += 1`.

II-*A* *Protection de la section critique*

Comment faire ? :

on protège la SC qui est la zone où il y a un risque de *perturbation* (d'interruption). Les modifications à apporter sont en gras et de plus grande taille.

```
import multiprocessing as mp

# Incrémentation avec protection de la variable partagée
def count2_on_protege_la_section_critique(nb_iterations):
    """ Chacun incrémente dans la section protégée """

    global variable_partagee
    global verrou

    for i in range(nb_iterations):
        verrou.acquire()
        variable_partagee.value += 1
        verrou.release()
```

2. Plus exactement, si nous ne voulons pas qu'une autre processus que celui qui a commencé l'incrémentation puisse incrémenter la même variable
3. On dira également que l'on se prémunit d'une **race condition**.

Le reste du code :

```
#------------ PARTIE principale (le point d'entrée de cet exemple -------
if __name__ == '__main__' :
    nb_iterations = 5000
    # La variable partagée
    variable_partagee = mp.Value('i',0)  # ce sera un entier

    # On recommence avec la version protégée par un verrou
    verrou=mp.Lock()

    print("la valeur de variable_partagee AVANT les incrémentations : ", variable_partagee.value)

    # On crée 2 process
    pid1=mp.Process(target=count2_on_protege_la_section_critique, args=(nb_iterations,)); pid1.start()
    pid2=mp.Process(target=count2_on_protege_la_section_critique, args=(nb_iterations,)); pid2.start()
    pid1.join(); pid2.join()

    print("la valeur de variable_partagee APRES les incrémentations %d (attendu %d): " %
(variable_partagee.value,nb_iterations*2))
```

- Trace : on teste plusieurs fois !

```
la valeur de variable_partagee AVANT les incrémentations :  0
la valeur de variable_partagee APRES les incrémentations 10000 (attendu 10000):
la valeur de variable_partagee AVANT les incrémentations :  0
la valeur de variable_partagee APRES les incrémentations 10000 (attendu 10000):
...
```

➛ Expliquer le code et *Lock* (verrou équivalent à un sémaphore binaire)

➛ On peut utiliser également RLock() (un Lock Ré-entrant = Récursif)

With-Statement-Context Manager : il est possible de simplifier la fonction ci-dessus

```
def count2_on_protege_la_section_critique(nb_iterations):
    """ Chacun écrit dans la section protégée """
    global variable_partagee;  global verrou
    for i in range(nb_iterations):
        verrou.acquire()
        variable_partagee.value += 1
        verrou.release()
```

Par ce qui suite (appelé *With-Statement-Context Manage* : gestionnaire des expressions **with xxx**).

```
def count2_on_protege_la_section_critique(nb_iterations):
    """ Chacun écrit dans la section protégée """
    global variable_partagee;  global verrou
    for i in range(nb_iterations):
        with verrou :
            variable_partagee.value += 1
```

L'intérêt est d'écrire moins de choses et de ne pas risquer d'oublier *release()* (danger de DeadLock).

II-*B* *Solution avec un sémaphore*

Un *Lock* est en fait un **sémaphore** avec un seul jeton (que l'on appelle sémaphore *binaire*).

Les sémaphores généralisent donc la notion de verrou en nous permettant de fixer le nombre de jetons à une valeur quelconque, y compris 0 (que l'on appelle sémaphore *privé*). [4]

Les primitives d'accès aux sémaphores (*release()* et *acquire()*) sont donc les mêmes que pour un verrou *Lock*.

Ci-dessous, nous avons le code du même exemple d'incrémentation avec un sémaphore.

```
import multiprocessing as mp

variable_partagee = mp.Value('i', 0)  # ce sera un entier initialisé à 0
verrou = mp.Semaphore()  # Val init=1


def count2_SC_sem(nb_iterations):
    """ Chacun écrit à son rythme (non protégée)"""
    global variable_partagee
    for i in range(nb_iterations):
        with verrou :
            variable_partagee.value += 1

def test_SC_protege_par_Sem():
    # if __name__ == '__main__' :
    nb_iterations = 5000

    # La variable partagée : placée hors cette fonction (sinon, la passr en param)
    # variable_partagee = mp.Value('i',0)  # ce sera un entier initialisé à 0

    print("la valeur de variable_partagee AVANT les incrémentations : ",
          variable_partagee.value)
    # On crée 2 process
    pid1 = mp.Process(target=count2_SC_sem, args=(nb_iterations,))
    pid1.start()
    pid2 = mp.Process(target=count2_SC_sem, args=(nb_iterations,))
    pid2.start()
    pid1.join()
    pid2.join()

    print("la valeur de variable_partagee APRES les incrémentations %d (attendu %d) " % (
        variable_partagee.value, nb_iterations * 2))

test_SC_protege_par_Sem()
```

• Trace : on teste plusieurs fois ! Tout va bien cette fois.

```
la valeur de variable_partagee AVANT les incrémentations :  0
la valeur de variable_partagee APRES les incrémentations 10000 (attendu 10000):
la valeur de variable_partagee AVANT les incrémentations :  0
la valeur de variable_partagee APRES les incrémentations 10000 (attendu 10000):
...
```

Rappel : l'expression et son équivalent (qui suit)

```
    with Sem :   # Sem est du type      mp.Smephore
            variable_partagee.value += 1
```

```
    Sem.acquire()
    try: variable_partagee.value += 1
    finally: Sem.release()
```

4. Dans la mesure où pour un tel sémaphore déclaré par **S=multiprocessing.Semaphore(0)**, la tâche qui exécute *S.release()* n'est en général pas celle qui exécute *S.acquire()*.

mettra en place, pour nous, les appels *acquire()* et *release()* sur le sémaphore *verrou*.

Cependant, dans certains cas, on a besoin d'appeler *acquire()* sans devoir appeler *release()* dans la même fonction. C'est notamment le cas dans l'utilisation des *sémaphores privés* utilisés dans le TP précédent sur les sémaphores : la situation où un processus attend sur *Sem.acquire()* et attend qu'un autre processus appelle *Sem.release()* pour le libérer.

II-C *Solution alternative*

• Parfois il est possible d'éviter une SC. Quand cela est possible, on peut faire en sorte que les différents Processus (on en a N) n'accèdent que des données dont ils sont seuls possesseurs.

➞ Dans l'exemple ci-dessus, on peut décider que pour procéder à nos incrémentations, chaque processus i accède à sa propre variable T_i où T est un tableau / liste de taille N.

• La solution alternative suivante associe chaque processus à sa propre variable (une zone partagée). A la fin des calculs, le processus principal se chargera de ramasser les résultats.

Pour cela, Python propose le type **Array** (notez le 'A' majuscule).

➞ Le verrou disparaît (plus besoin dans ce cas précis). A aucun moment la *race-condition* survient.

```
import multiprocessing as mp

# Ici, chaque process incrémente la valeur de SA case
 # ATTENTION : l'écriture ci-dessus n'est pas efficace (mais là n'est pas le but !). Cependant on peutécrire écrire :
 # def count3_on_travaille_dans_un_array_VERSION_PLUS_RAPIDE(nb_iterations) :
def count3_on_travaille_dans_un_array(nb_iterations):
    global tableau_partage
    for i in range(nb_iterations):
        mon_indice = mp.current_process().pid % 2  # donnera 0 / 1 selon le process
        tableau_partage[mon_indice]+=1


    global tableau_partage
    var_local_a_moi_tout_seul=0
    for i in range(nb_iterations): var_local_a_moi_tout_seul+=1
     # Et on écrit UNE SEULE FOIS :
    mon_indice = mp.current_process().pid % 2
    tableau_partage[mon_indice]+=1
#-------------------------- Avec Array ------------------------------
tableau_partage = mp.Array('i', 2)  # tableau de 2 entiers

# Initialisation des array :
tableau_partage[0]=0; tableau_partage[1]=0;   # Initialisation de l'arra
# Ou via
tableau_partage[:]= [0 for _ in range(2) ] # IL FAUT les [:] sinon, ne marche pas (et tableau_partage devient une liste !)

# ATTENTION : NE PAS INITIALISER comme ceci : tableau_partage= [0 for _ in range(2) ]
# Cette écriture redéfinira  notre Array comme une liste ! (principe de la prog. fonctionnelle)

# Egalement, sans [:], print dennera le type de l'Array, pas son contenu
print("le contenu du tableau_partage AVANT les incrémentations : ", tableau_partage[:])

# On crée 2 process
nb_iterations = 5000
pid1=mp.Process(target=count3_on_travaille_dans_un_array, args=(nb_iterations,)); pid1.start()
pid2=mp.Process(target=count3_on_travaille_dans_un_array, args=(nb_iterations,)); pid2.start()
pid1.join(); pid2.join()

print(tableau_partage[0], " et " , tableau_partage[1])
print("la somme du tableau partage APRES les incrémentations : %d (doit etre %d)"\
      %(sum(tableau_partage),nb_iterations*2) )
```

• La trace :

```
"""
TRACE
le contenu du tableau_partage AVANT les incrémentations :  [0, 0]
5000  et  5000
la somme du tableau partage APRES les incrémentations : 10000 (doit etre 10000)
"""
```

Ici, chaque processus a travaillé avec sa case et *main* fait la somme.

☞ Question : a-t-on besoin de protéger la somme faite par *main*(via la fonction *sum*) ?

III Éléments utile pour la suite

III-A *Différentes mesures du temps*

Voir aussi la documentation : https ://docs.python.org/3/library/time.html

III-B *time.time()*

- Avec time.time() : en secondes (ou plus récemment millisecondes ?)

```
>>> import time
>>> time.time()       #return seconds from epoch
1261367718.971009
```

III-C *time.time_ns()*

- Avec time_ns()

```
>>> import time
>>> time.time_ns()
1530228533161016309

>>> time.time_ns() / (10 ** 9) # convert to floating-point seconds
1530228544.0792289
```

III-D Échange de valeurs entre processus

• Ci-dessous un processus avec pipe / queue pour les cas simples (pour éviter le renvoie de valeur / SHM)

III-E Echange avec Queue (get / put)

Un exemple simple :

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())
    p.join()

# # prints "[42, None, 'hello']"
```

III-F Echange avec Pipe (send / recv)

Caractérisé avec send/recv

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())
    p.join()

# prints "[42, None, 'hello']"
```

III-F-1 Echange de messages (avec Pipe et send/recv)

Remarquer l'échange d'un tableau entier.

```
>>> from multiprocessing import Pipe
>>> a, b = Pipe()
>>> a.send([1, 'hello', None])
>>> b.recv()
[1, 'hello', None]
>>> b.send_bytes(b'thank you')
>>> a.recv_bytes()
b'thank you'
>>> import array
>>> arr1 = array.array('i', range(5))
>>> arr2 = array.array('i', [0] * 10)
>>> a.send_bytes(arr1)
>>> count = b.recv_bytes_into(arr2)
>>> assert count == len(arr1) * arr1.itemsize
>>> arr2
array('i', [0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

III-G *Échange à l'aide de 'value' (et lock dans sa déclaration)*

```
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double

class Point(Structure):
    _fields_ = [('x', c_double), ('y', c_double)]

def modify(n, x, s, A):
    n.value **= 2
    x.value **= 2
    s.value = s.value.upper()
    for a in A:
        a.x **= 2
        a.y **= 2

if __name__ == '__main__':
    lock = Lock()

    n = Value('i', 7)
    x = Value(c_double, 1.0/3.0, lock=False)
    s = Array('c', b'hello world', lock=lock)
    A = Array(Point, [(1.875,-6.25), (-5.75,2.0), (2.375,9.5)], lock=lock)

    p = Process(target=modify, args=(n, x, s, A))
    p.start()
    p.join()

    print(n.value)
    print(x.value)
    print(s.value)
    print([(a.x, a.y) for a in A])

"""
49
0.1111111111111111
HELLO WORLD
[(3.515625, 39.0625), (33.0625, 4.0), (5.640625, 90.25)]
"""
```

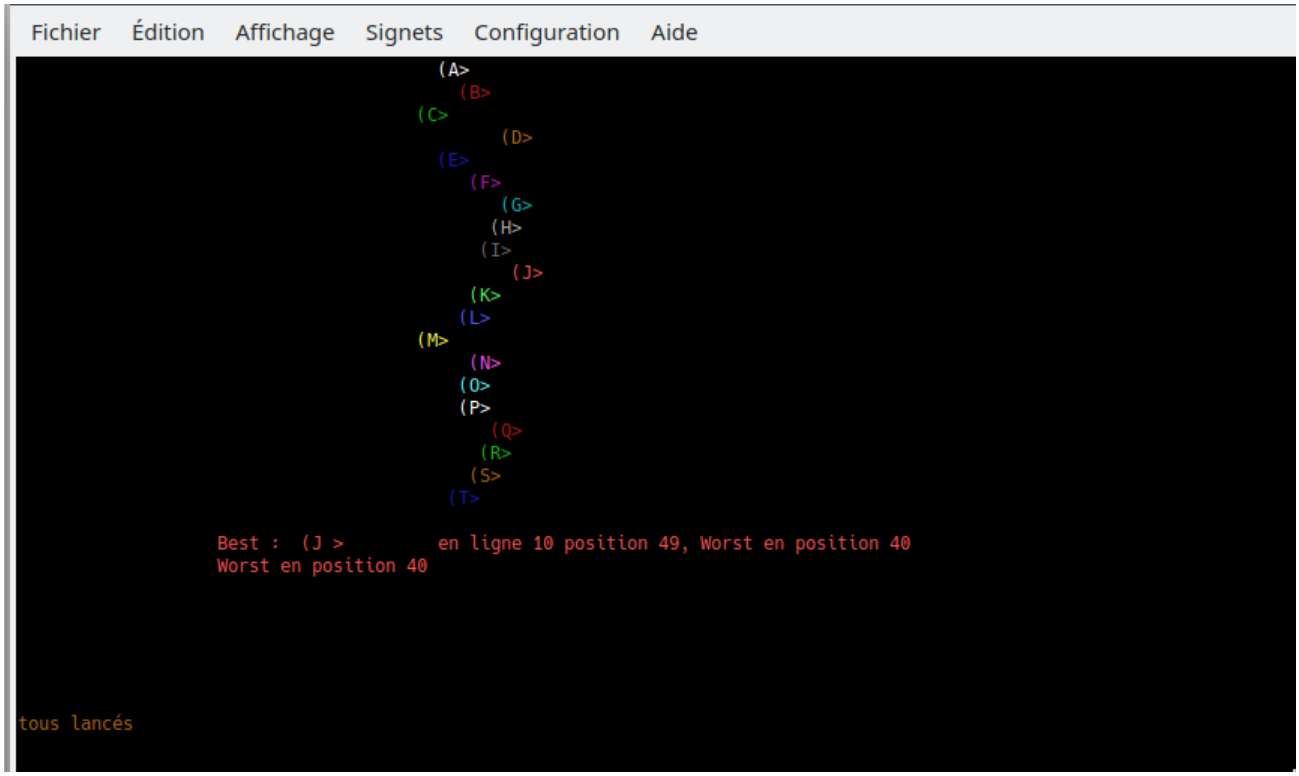
☞ Voir aussi https://docs.python.org/3/library/multiprocessing.html (p.ex. BAseMAnager, Listeners and Clients)

☞ Lire également en bas de cette même page "Programming guidelines"

IV Exercices

IV-A *Exercice : Course Hippique*

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 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).

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'écran VT100 ou similaire).

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

```
# Juin 2019
# Cours hippique
# Version très basique, sans mutex sur l'écran, sans arbitre, sans annoncer le gagant, ... ...

# 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

#------------------------------------------------------------------

from multiprocessing import Process
import os, time,math, random, sys

keep_running=True  # Fin de la course ?

# 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 !


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

    while col < LONGEUR_COURSE and keep_running :
        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))
```

```
#-------------------------------------------------------------
# La partie principale :
if __name__ == "__main__" :
    Nb_process=20
    mes_process = [0 for i in range(Nb_process)]

    LONGEUR_COURSE = 100
    effacer_ecran()
    curseur_invisible()

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

    move_to(Nb_process+10, 1)
    print("tous lancés")



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

    move_to(24, 1)
    curseur_visible()
    print("Fini")
```

IV-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 mutex.
- 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"
- 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 !) :

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

IV-*B* *Exercice : Calcul parallèle de la somme de nombres*

• On se donne une (grande) liste (ou *array* pour plus d'efficacité) initialisé à 1.

• On calculera la somme de cette liste d'abord par un seul processus puis par plusieurs. On comparera les temps de calculs respectifs.

Exercice-1 : Écrire une version sans création de processus qui définit et remplit un tableau (*array*, *liste*, etc), calcule la somme des éléments de ce tableau et affiche ce résultat avec le temps de calcul nécessaire (à l'aide du module *time* et l'appel de la fonction *time.time()*).

Exercice-2 : Écrire la version multi-processus de la *somme*. Faites en sorte que l'on puisse passer le nombre de processus à créer en paramètre (*sys.argv*). Conserver le même tableau pour pouvoir comparer les temps d'exécution.

Indications :

- Pour représenter le tableau des valeurs, vous pouvez utiliser une liste, un 'array', une variable partagée 'Array' ou une mémoire partagée (SharedArray). La classe *Manager* propose également des listes ou dictionnaires partagés.

- Pour faire la somme en parallèle, le tableau sera découpé en autant de tranches que de Processus créés. Chaque processus cumule la somme de sa "tranche" dans une variable locale puis ajoute cette somme partielle à une variable partagée.

- La variable globale qui contiendra la somme totale des éléments du tableau doit être protégée par un verrou (*mutex*) dans une section critique.

- Pour régler le problème de visibilités des variable globales, certaines d'entre elles peuvent être passées en paramètres.

IV-C *Estimation de PI*

☞ Voir l'exercice ci-après.

• 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).

IV-C-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.

IV-C-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).

IV-*D* *Exercice : Estimation parallèle de Pi*

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.

IV-E *Exercice : Calcul parallèle du Merge Sort*

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.

IV-E-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))
```

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
```

IV-E-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])
```

IV-F *Exercice : tri-rapide*

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)
```

IV-*G* *Exercice : calcul parallèle des nombres premiers*

Exercice-7 : On se fixe un entier N (p. ex. 10000) et on souhaite calculer les nombres premiers dans l'intervalle $2..N$ à l'aide de plusieurs processus Python.

☞ Ne pas dépasser 8 processus (Intel $I7$) sous peine de surcharge votre machine.

Réaliser cette application.

Rappel : on dira que l'entier P est premier s'il n'est pas divisible par un entier dans l'intervalle $2..\lceil\sqrt{P}\rceil$

V Autres méthodes de calcul de PI

Pour indication et complément : il existe de multiples méthodes de calcul de PI. En voici quelques unes.

V-A *PI par la méthode arc-tangente*

Méthode arc-tangente :

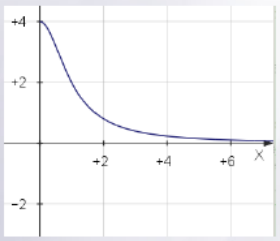
- On peut calculer une valeur approchée de PI par la méthode suivante :

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

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)

V-B *Par la méthode d'espérance*

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

V-C *Par la loi Normale*

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

V-D *Par une méthode probabiliste*

☞ Ce calcul n'apporte pas de grande efficacité dans sa version parallèle (par rapport à lma méthode séquentielle). Cependant, il reste un "bon" exercice de parallélisme !

La méthode : la probabilité qu'il y ait le même nombre de piles et de faces dans n (n grand) lancers d'une pièce est $\frac{1}{\sqrt{\pi.n}}$ (en fait, cette probabilité est $\frac{\binom{2n}{n}}{2^{2n}} \equiv \frac{1}{\sqrt{\pi.n}}$ par la formule de *Stirling*).

➙ On calcule donc la partie gauche qu'on pose égale à $\frac{1}{\sqrt{\pi.n}}$

☞ Ici, il faut calculer la factorielle en parallèle :

Définissez un tableau *tab* ou *tab[i]* contiendra $i!$.

Remplissez ce tableau avec un calcul parallèle (et PrD).

Notez que ce code de calcul parallèle n'est pas plus efficace que la solution séquentielle à cause du fait d'attendre que (i-i) ! soit calculer avant de calculer $i!$ avec l'aide des pipes.

VI 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.

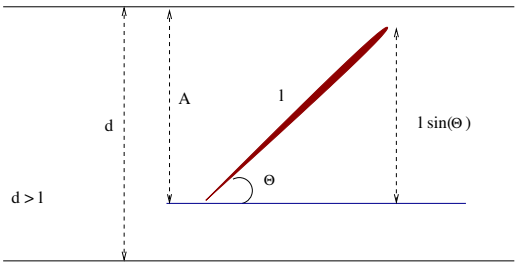


FIGURE 1 – Problème des Aiguilles de Buffon

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 :

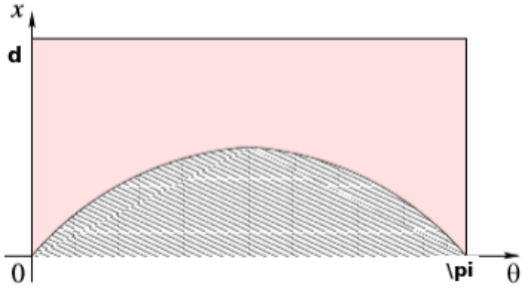
$$p = \int_0^{\pi} \int_0^{lsin(\theta)} \frac{1}{d.\pi}\, dA\, d\theta = \frac{2l}{d.\pi} \qquad [1]$$

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) !

VI-*A* *Travail à réaliser*

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

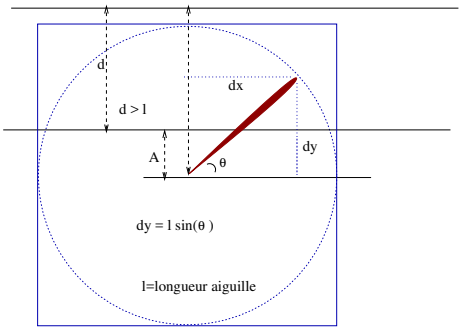
   $A \in [0, d]$ uniformément réparti

   $\theta \in [0, \pi]$ 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.

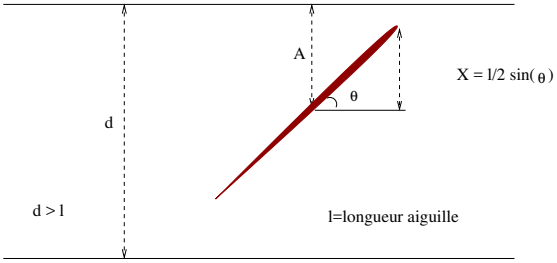
```
def calcul_PI_OK_selon_mes_slides_on_evite_use_of_PI_version_sequentielle() :
    L_lg_needle=10 # cm
    D_dist_parquet= 10 # distance entre 2 lattes du parquet
    Nb_iteration=10**6

    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))
```

VI-*B* *Addendum aux aiguilles de Buffon*

VI-B-1 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).

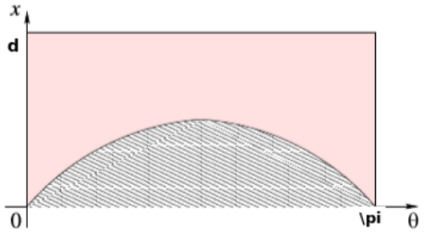
On a alors la formulation de probabilité "totale" : $P(E) = \displaystyle\int_0^{\pi} P(E|\Theta = \theta)\ f_{\Theta}(\theta)\ d\theta$

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

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 $a = d$) :



VI-B-2 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**.

Table des matières