Distribution de l'échantillonneur de Gibbs (MCMC)

I) Echantillonneur de GIBBS

L'échantillonneur de Gibbs est un algorithme de la famille MCMC (Markov Chain Monte Carlo).
Selon une distribution de probabilité π sur un univers Ω, cet algorithme crée une chaîne de Markov dont la distribution stationnaire est π. Il permet de générer des éléments de Ω selon la loi π sans utiliser directement son expression car difficile à manipuler mais en utilisant les densités conditionnelles. Nous sommes partis d'un cas spécifique pour faciliter l'implémentation

Ci-dessous se trouve un code réalisé sur python de ce fameux échantillonneur de Gibbs.

```
def gibbs(n):
    x=0
    y=0
    thining=500
    abscisse = []
    ordo = []
    for i in range(n):
        for j in range(thining):
            x=np.random.gamma(4,1.2/(y*y+5))
            y=np.random.normal(1.2/(x+2),1.2/np.sqrt(x+2))
        abscisse.append(x)
        ordo.append(y)
    return abscisse, ordo
t_init = time.time()
sample_results = gibbs(20000)


############################ Affichage du temps d'exécution ###################################


print("Le temps d'exécution est de  %s secondes"%(time.time()-t_init))
```

```
Le temps d'exécution est de  76.19361114501953 secondes
```

Avant même l'exécution du code, nous pouvons nous rendre compte que le temps d'exécution peut être assez élevé, en effet, nous pouvons voir deux boucles imbriquées ainsi que deux appels à des fonctions non triviales de python (np.random.gamma & np.random.normal qui sont donc de la librairie Numpy).

Et comme prévu nous pouvons voir en dessous le temps d'exécution qui est assez élevé, en effet il est d'environ 76 secondes, ce qui disons-le pour ce simple algorithme est assez long.

Le coût de cet algorithme est O(n) car en doublant les entrées, on double le temps d'exécution.

C'est suite à ce constat (surtout car c'était le but du projet) que nous avons voulu donc paralléliser une partie du code pour drastiquement diminuer le temps d'exécution.

II) Distribution de l'algorithme

a) Multithreading vs multiprocessing

Plusieurs méthodes nous permettaient d'y parvenir mais nous avons choisi de nous focaliser sur le multithreading et le multiprocessing.

Cependant étant sur python, le multithreading n'était pas une bonne option à cause du verrou GIL (Global Interpreter Lock) de python.
Python protège les listes et les dictionnaires via ce verrou qui est le même pour toutes les listes pour pouvoir efficacement gérer le garbage collector. Ce qui fait que dans les faits, le langage python n'est presque pas multithreading alors qu'il est supposé l'être par design.

Pour cette raison donc le multithreading empêche notre système d'être au maximum de ses capacités.
Le multiprocessing en revanche permet de créer des programmes qui s'exécutent en concurrence.
La librairie multiprocessing attribue à chaque processus son propre interpréteur et à chacun son propre GIL.

b) Choix du multiprocessing

Comme on peut le voir sur le code l'algorithme non distribué ci-dessus, nous simulons 20000 observations sur un seul processeur donc.

Avec le multiprocessing nous est donc venue l'idée de distribuer ces simulations sur plusieurs processeurs. Au lieu de simuler 20000 observations sur un seul processeur, nous avons décidé d'en simuler (20000 / nombre de processeurs) par processuer, par exemple pour 5 processeurs nous en simulerions que 4000 que nous regrouperions à la fin de l'exécution afin d'obtenir les 20000 observations voulues.

Voici ci-dessous l'algorithme obtenu :

```
final_samples = [], []
def gibbs2(n):
    x=0
    y=0
    thining=500
    abscisse = []
    ordo = []
    for i in range(n):
        for j in range(thining):
            x=np.random.gamma(4,1.2/(y*y+5))
            y=np.random.normal(1.2/(x+2),1.2/np.sqrt(x+2))
        abscisse.append(x)
        ordo.append(y)
    return abscisse, ordo

############################ Début de la parallélisation ###################################
def gibbs_distribue(obs, n_process):

    n = round(obs/n_process)
    with concurrent.futures.ProcessPoolExecutor() as executor:
        results =[executor.submit(gibbs2,n) for _ in range(n_process)]
        for f in concurrent.futures.as_completed(results):
            final_samples[0].append(f.result()[0])
            final_samples[1].append(f.result()[1])


############################ Affichage du temps d'exécution ###################################
t_init = time.time()
gibbs_distribue(20000,5)
print("Le temps d'exécution est de  %s secondes"%(time.time()-t_init))
```

```
Le temps d'exécution est de  22.28390407562256 secondes
```

On voit que le temps d'exécution a considérablement diminué, en effet, il a été divisé par environ 3.42, on pourrait s'attendre à ce que le temps d'exécution soit divisé par 5 mais cela peut être dû au fait qu'il y a un verrou sur chacune les listes, en effet, pour enregistrer les résultats dans les listes final_samples, la fonction concurrent.futures.as_completed() permet de nous donner les résultats dans l'ordre de complétion et tant que la tâche à exécuter en dessous n'est pas terminée, grâce à un verrou, les autres processeurs ne peuvent incrémenter la liste.
Si le processeur 3 termine son exécution en premier alors les échantillons issus de celui-ci seront enregistrés dans les listes et seulement à la fin le verrou sera levé pour permettre au prochain processeur qui a terminé d'y ajouter ses données et ainsi de suite jusqu'à la fin de l'exécution du programme. Les résultats s'affichent donc par ordre d'arrivée.

Ceci pourrait expliquer pourquoi le temps d'exécution n'est pas divisé par le nombre de processeurs.
Le coût de l'algorithme est O(n) car de nouveau en doublant les entrées, on double le temps d'exécution.

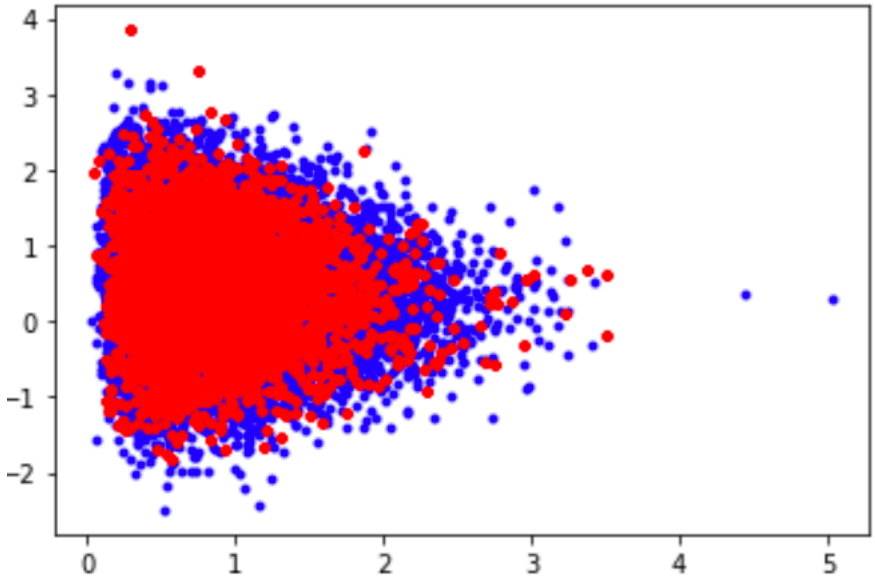
En allant dans le moniteur d'activité de l'ordinateur on peut voir que lors de l'exécution du programme plusieurs processeurs pythons sont créés et que 5 d'entre eux s'exécutent presque à 100%

Nom du processus	% process...	Temps de traitement
revisiond	0,0	0,13
ReportCrash	0,0	0,01
remoted	0,0	0,14
remindd	0,0	0,54
rapportd	0,0	0,39
QuickLookUIService (PID 415)	0,0	0,08
QuickLookSatellite	0,0	0,45
python3.7	0,1	1,93
python3.7	0,0	1,17
python3.7	99,3	6,55
python3.7	99,4	6,55
python3.7	0,0	0,00
python3.7	99,4	6,55
python3.7	0,0	0,00
python3.7	99,5	6,55
python3.7	0,0	0,00
python3.7	99,3	6,55
python	0,0	0,06

Il y en a autant d'inactifs car nous avons testé à plusieurs reprises pour différents nombres de processeurs les résultats.

III) Vérifications

Afin de vérifier que les deux versions de l'algorithme étaient bien en accord, nous avons décidé de faire un graphique des 2 échantillons finaux et de les comparer, voici ci-dessous les résultats :



Échantillonneur issu de l'échantillonneur de Gibbs (bleu) vs l'échantillonneur de Gibbs distribué (rouge)

On voit que les 2 ont la même distribution et sont très similaires ce qui est normales vu qu'ils sont issus des mêmes distributions.