Posteriors avec Monte Carlo Hamiltonien

petite introduction

J.E Campagne

2 octobre 2021

Table des matières

1 Introduction

Dans la présente note, je vais donner une illustration d'échantillonnage Monte Carlo par la méthode Hamiltonienne. Dans un premier temps, je vais passer en revue quelques méthodes classiques de tirage Monte Carlo: Importance Sampling, Metropolis et Metropolis-Hasting. Autant que possible je recode ces méthodes afin de les discuter. Une section est dédiée à l'usage de la librairie `NumPyro` construite sur la base de `JAX` décrite dans la note *`JAX`: un Numpy augmenté* (12 Sept. 2021). Ainsi, un notebook (`demoMCSampling.ipynb`) associé à cette note est disponible à l'adresse https://github.com/jecampagne/jax-demo.

Faisons un petit rappel bayésien avant de coder. Mettons que l'on ait un jeu de paramètres z inconnus (on peut étendre aux variables cachées, dites *latent variables*) ainsi qu'une connaissance *a priori* de leur distribution de probabilité $p(z)$ (le *prior*). Si, l'on dispose des données x et que l'on spécifie également la dépendance de x vis-à-vis de z via le *likelihood* $p(x|z)$, alors en appliquant le théorème de Bayes, il vient que la distribution *a posteriori* des paramètres z étant donné les données x suit la relation

$$p(z|x) = \frac{p(z)p(x|z)}{\int p(z)p(x|z)dz} \tag{1}$$

Cependant, comme en Mécanique Statistique, le calcule de l'intégrale au dénominateur est en générale inaccessible et l'on doit recourir à des approximations. On peut citer les méthodes variationnelles bayésiennes (BVI) et les Monte Carlo de générations de chaînes de Markov (MCMC).

Dans le cas de la BVI, on introduit une paramétrisation de la *posterior*, notée $p_\theta(z|x)$, et l'on ajuste les paramètres θ selon une fonction de coût. En fait, on utilise comme en Mécanique Statistique [1] la minimisation de la distance de Kullback–Leibler

$$D_{KL}(q_\theta(z|x)\|p(z|x)) = E_{z\sim q_\theta(z|x)}[\log q_\theta(z|x) - \log p(z|x)] \geq 0 \tag{2}$$

La fonction de coût $\ell(\theta)$ que l'on va chercher à maximiser est alors définie par

$$\ell(\theta) = \log p(x) - D_{KL}(q_\theta(z|x)\|p(z|x)) = E_{z\sim q_\theta(z|x)}[\log p(x,z) - \log q_\theta(z|x)] \leq \log p(x) \tag{3}$$

On appelle également $-\ell(\theta)$ *l'énergie libre variationnelle*. Bien entendu, le choix de $q_\theta(z|x)$ doit être judicieux pour rendre la maximisation de $\ell(\theta)$ possible, je ne m'étend pas.

De leur coté, les méthodes de MCMC tentent de générer un lot d'échantillons (ie. *la chaîne*) $\{z_i\}_{i<N}$ qui asymptotiquement reflète un tirage selon la loi $p(z|x)$. Pour cela, comme pour BVI, on se donne une distribution (*a priori*) $p(z)$ avec laquelle on tire un premier échantillon z_0. Ensuite, par la donnée d'une probabilité de transition $q(z_i|z_{i-1}, x)$, on génère successivement tous les éléments de la chaîne. Ainsi, si on attend suffisamment longtemps on peut en principe approcher la distribution $p(z|x)$ aussi fidèlement que l'on veut. Le "suffisamment longtemps" est bien un point des plus délicats et la méthode de Monte Carlo Hamiltonien a été mise au point pour éviter la marche aléatoire de la chaîne.

Si on veut savoir quand utiliser l'une ou l'autre méthode, au delà des arguments théoriques (ie. convergence assurée ou pas), on peut dans un premier temps avoir à l'esprit l'argument suivant: si l'on dispose de petits lots de données et que l'on est confiant en son modèle alors MCMC est adapté, par contre si l'on a beaucoup de données et que l'on veuille tester rapidement beaucoup de modèles, alors BVI est plus adapté. Ceci étant dit passons à la pratique.

1. Voir Sec. II.4.1 de la note *Relations entre Réseaux de Neurones, Physique Statistique et Théorie de l'Information* (JEC, 2021).

2 Calcul d'une intégrale 1D

Considérons le calcul d'une intégrale 1D [2]

$$I = \int \phi(x)p(x)dx$$

$$\text{avec} \qquad \phi(x) = \frac{x^2}{8}, \qquad p(x) = \frac{1}{Z}\left(\frac{w_1}{\sigma_1}e^{-(x/\sigma_1)^2/2} + \frac{w_2}{\sigma_2}e^{-((x-\mu_2)/\sigma_2)^2/2}\right) \equiv \frac{\tilde{p}(x)}{Z_p} \qquad (4)$$

(avec $(w_1, w_2) = (8, 2)$, $(\sigma_1, \sigma_2) = (0.5, 0.1)$ et $\mu_2 = 3/2$). Le facteur Z est là pour satisfaire $\int p(x)dx = 1$. Dans ce cadre $\tilde{p}(x)$ est la forme non-normalisée de $p(x)$. On se rend compte que I n'est autre que la valeur moyenne de $\phi(x)$ selon la loi de densité de probabilité $p(x)dx$ qui dans l'exemple est une superposition de 2 gaussiennes. Le graphe des différentes fonctions est donné sur la figure 1. Il est assez clair que l'on peut utiliser

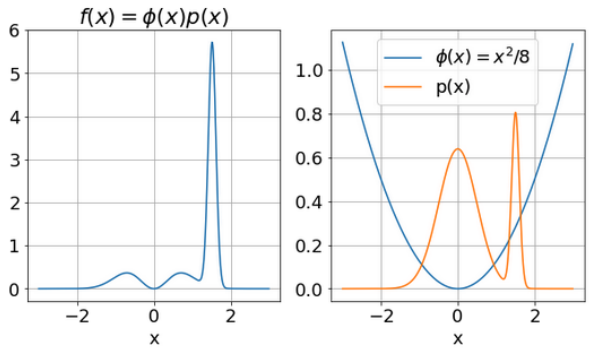

FIGURE 1 – Graphes des fonctions impliquées dans le calcule de l'intégrale I (Eq. 4).

une méthode [3] d'intégration numérique classique (ex. par quadrature Simpson, Gauss-Legendre, Clenshaw-Curtis,...) pour estimer I, mais l'idée ici est de procéder par tirage Monte Carlo (MC).

Une idée qui vient à l'esprit immédiatement quand on réalise comme Leibnitz que l'intégrale est l'aire sous la courbe $f(x)$ (ie. *l'hypographe*), c'est de procéder à la méthode

2. En pratique, on limite les bornes d'intégration à l'intervalle $[-3, 3]$.
3. Au passage dans le notebook, je fais appel à `scipy.integrate` bien que $p(x)$ soit codé en `JAX`. Il n'y a pas de base un équivalent `jax.scipy.integrate` mais je donne une implémentation simple de la quadrature de Clenshaw-Curtis.

du "toucher-couler". Elle revient à tirer au hasard (loi uniforme) un couple (x, y) dans une aire de surface connue qui englobe le graphe de $f(x)$ (ex. un bête rectangle de base égal au support de $f(x)$ et dont la hauteur est donnée par un majorant du maximum de f) et de procéder au comptage de la fraction des couples se trouvant sous la courbe $f(x)$. Je n'ai pas codé cette méthode, elle fonctionne bien si f est relativement constant sur le support d'intégration, ce qui n'est manifestement pas la cas dans notre example.

On peut alors se tourner vers la méthode suivante: procéder à un tirage aléatoire uniforme d'un lot d'abscisses $\{x_i\}_{i<N}$ dans le support de l'intégrale (uniformément car on se réfère à la masse dx), puis on procède à l'évaluation de l'intégrale selon la moyenne pondérée suivante [4]:

$$\hat{I} = \frac{\sum_{i=1}^{N} \phi(x_i)\tilde{p}(x_i)}{\sum_{i=1}^{N} \tilde{p}(x_i)} \tag{5}$$

Je donne dans le notebook une implémentation basique sans chercher à optimiser le code, vous y verrez en passant la manipulation des clés pour la génération de nombres aléatoires. Si donc on obtient une estimation de l'intégrale, sa précision évolue typiquement en $1/\sqrt{N}$ (Fig. 2) ce qui rend cette technique assez peu compétitive par rapport à des méthodes par quadrature. Par exemple, il nous faut dépasser 10^6 d'échantillons pour obtenir une erreur relative inférieure à 10^{-4}, alors qu'avec une quadrature adaptée (ex. 160pts Clenshaw-Curtis), on obtient aisément une erreur relative bien inférieure à 10^{-10}. Cependant, l'idée *in fine* n'est pas d'évaluer des intégrales en 1D, mais de générer des événements selon une loi de probabilité dans des dimensions qui peuvent être largement supérieures [5].

On remarque que le problème de cette méthode vient de ce que la génération uniforme des abscisses ne prend pas en compte les lieux où la distribution $\tilde{p}$ est la plus importante. Ainsi, on génère inutilement des échantillons x_i qui n'ont qu'une faible importance dans le calcul. Autrement dit, la probabilité uniforme dx ne rend pas compte de la concentration de $p(x)$ sur un support restreint. Or, ce phénomène de concentration tant à s'accentuer quand on considère $x \in \mathbb{R}^d$ avec d "grand" (cf. Th. des Grandes Déviations [6]), donc il faut trouver d'autres méthodes.

4. nb. mettons que l'on ne connaisse pas la normalisation Z, on peut alors utiliser $\tilde{p}$ ce qui est un avantage.

5. En passant, l'article de Vahid Keshavarzzadeh et al. propose un algorithme de calcul de quadrature en multi-dimensions avec une expérimentation en $d = 100$, https://arxiv.org/pdf/1804.06501.pdf.

6. Voir J.E.C, *Relations entre Réseaux de Neurones, Physique Statistique et Théorie de l'Information*, Juin 2021.

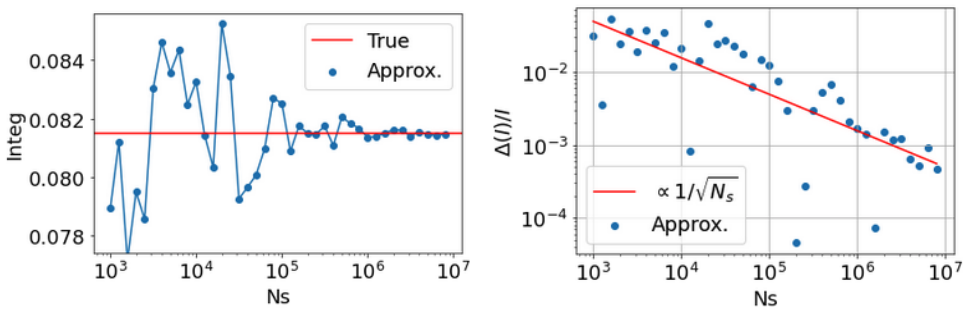

FIGURE 2 – Évaluation de l'intégrale 1D (Eq. 5) selon le nombre d'échantillons utilisés. A droite il s'agit de l'erreur relative, où la référence est un calcul effectué via `scipy.integrate.quad`.

Ce constat est le même si on se pose le problème non pas de calculer une intégrale, mais d'une manière générale de générer des échantillons selon une loi de distribution, par exemple pour estimer les paramètres d'un modèle et leurs *intervalles de confiance*.

3 Importance Sampling

Une première approche pour tenter de palier le problème soulevé dans la section précédente consiste à utiliser une distribution intermédiaire q, dont on sait tirer des échantillons (ie. selon la loi $q(x)dx$). En reprenant, l'exemple de la distribution $p(x)$ Eq. 4, il semble en effet que l'on peut tenter de prendre soit une simple distribution gaussienne pour $q(x)$ (notée $q_0(x)$), soit une somme de deux gaussiennes (notées $q(x)$) pour rendre compte des 2 composantes de $p(x)$, comme sur la figure 3. Dans cet exercice, on fait "semblant" de connaitre un peu les caractéristiques de $p(x)$ pour comparer les résultats obtenus avec les deux distributions q_0 et q. Dans la suite, sauf cas contraire ce qui vaut pour $q(x)$ vaut pour $q_0(x)$. Notons, que pour être néanmoins un peu générale, je considère le cas où les deux distributions $q(x)$ et $q_0(x)$ se mettent sous la forme générique $\tilde{q}(x)/Z_q$, c'est-à-dire que les distributions ne sont pas forcément normalisées.

Ainsi, les échantillons $\{x_i\}_{i<N}$ tirés selon $q(x)dx$, vont tantôt *sur* représenter ou *sous* représenter des valeurs de x si ces dernières avaient été tirées de la loi originale $p(x)dx$. Il

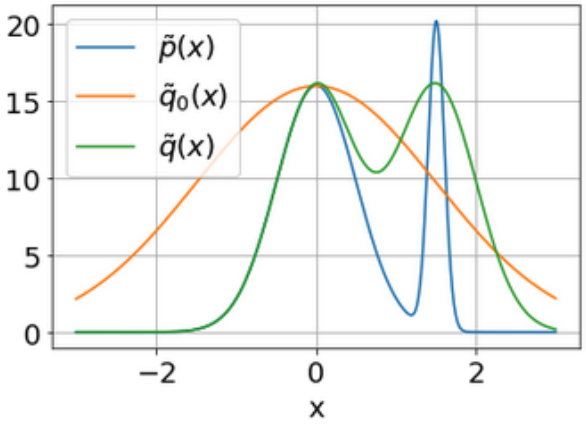

FIGURE 3 – Graphes des distributions non normalisée $\tilde{p}(x)$ et $\tilde{q}(x)$, $\tilde{q}_0(x)$.

convient donc de pondérer chaque échantillon du poids:

$$w_i = \frac{\tilde{p}(x_i)}{\tilde{q}(x_i)} \tag{6}$$

Ainsi, si l'objectif est de calculer l'intégrale Eq. 4, alors la formule 5 est simplement modifiée selon

$$\hat{I} = \frac{\sum_{i=1}^{N} \phi(x_i) w_i}{\sum_{i=1}^{N} w_i} \tag{7}$$

En quelque sorte, on a réécrit Eq. 4 selon

$$I = \int \phi(x)\ (p(x)/q(x))\ q(x)dx = \int \phi(x) w(x)\ q(x)dx \tag{8}$$

et on s'est placé dans le cas de la section précédente mais avec l'avantage de tirer x selon $q(x)dx$ et non uniformément (ie. selon dx).

Une implémentation est donnée dans le notebook et les résultats sont présentés sur la figure 4. On perçoit une amélioration avec cette méthode d'"Importance Sampling" par rapport à la méthode du paragraphe précédent en utilisant l'approximation $q(x)$ alors que numériquement on ne constate pas d'améliorations pour les résultats obtenus avec $q_0(x)$ (ce point se constate particulièrement quand on effectue les moyennes par décade du nombre d'échantillons, voir le notebook). Ainsi, on peut tirer un premier enseignement: trouver la "bonne" loi d'approximation $q(x)$ demande à connaitre assez finement la

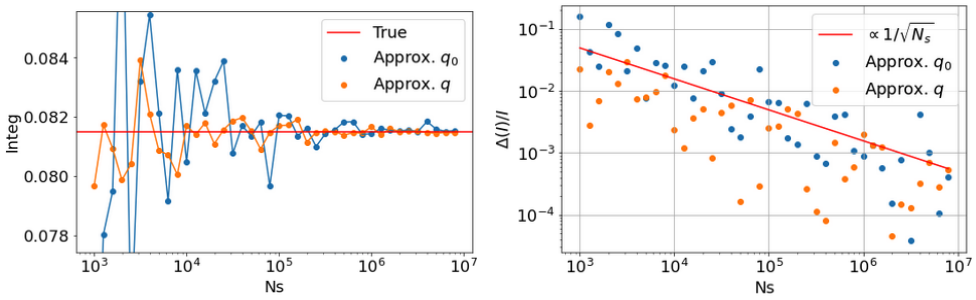

FIGURE 4 – Évaluation de l'intégrale 1D (Eq. 7) selon le nombre d'échantillons utilisés. A comparer avec la figure 2, et dans ce cadre la courbe qui sert de "guide" est la même dans les 2 figures.

probabilité $p(x)$, en particulier les lieux géométriques qui comptent (ie. ses divers modes par exemples) comme par exemple le pic localisé à $x = 1.5$. Il nous faut en particulier une fonction $q(x)$ avec des queues suffisantes pour "accrocher" tous les pics de la distribution $p(x)$. Mais se faisant, trop de queue comme la gaussienne simple $q_0(x)$ engendre mécaniquement des inefficacités déjà rencontrées avec la loi uniforme, et pire cela génère également des poids très importants.

La distribution de poids $\{w_i\}_{i<N}$ est très importante, et je vais en donner une illustration en dimension d quelconque à travers un exemple qui semble anodin. Soit la distribution uniforme $p(x)dx$ sur une boule de rayon R:

$$p(x) = \frac{1}{V_d(R)} \begin{cases} 1 & \|x\|_2 = \sqrt{\sum_{k=1}^{d} x_k^2} = r < R \\ 0 & \text{sinon} \end{cases} \tag{9}$$

avec $V_d(R)$ le volume de la boule de rayon R en dimension d, et l'on veut estimer l'intégrale

$$I(d) = \int r^2 p(x)\ d^d x \tag{10}$$

en utilisant la distribution normale comme "approximation"

$$q(x) = \mathcal{N}(0, \sigma^2; d) = \frac{e^{-x^2/(2\sigma^2)}}{(2\pi\sigma^2)^{d/2}} \tag{11}$$

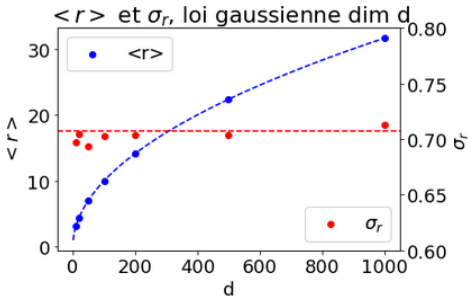

FIGURE 5 – Calcul de la moyenne de la distance à l'origine r et de son écart-type, pour des points tirés selon une loi gaussienne, en fonction de la dimension du problème. J'ai pris $\sigma = 1$.

Notez au passage que la valeur exacte de $I(d)$ est par ailleurs simple à calculer et vaut $I(d) = R^2 \times d/(d+2)$.

Une première chose à réaliser [7], c'est que si $x \sim \mathcal{N}(0, \sigma^2; d)$ alors la valeur moyenne de r et son écart-standard sont égaux à $(d \gg 1)$:

$$\langle r \rangle \approx \sigma\sqrt{d} \qquad\qquad \sigma_r \approx \frac{\sigma}{\sqrt{2}} \tag{12}$$

Une illustration numérique de ces lois est donnée sur la figure 5 dont la production peut être retrouvée dans le notebook. En conséquence, les points $x \sim \mathcal{N}(0, \sigma^2; d)$ se "concentrent" dans une coquille sphérique à une distante de l'origine égale à $\langle r \rangle$ et dont l'épaisseur σ_r est constante, le rapport $\sigma_r/\langle r \rangle$ tendant vers 0 au fur et à mesure que la dimensionalité du problème augmente. Ainsi, pour que le rapport $\tilde{p}(x_i)/q(x_i)$ ne soit pas majoritairement nul, il faut que $\langle r \rangle \leq R$ ou bien $\sigma \leq R/\sqrt{d}$. J'en tiens compte dans la fonction `test(dim)` du notebook dont je reproduis ici quelques éléments:

```
@partial(jit , static_argnums=(1,2))
def importance_sampling_nD(key,N,dim=2,R=1,s=1):
    key, subkey = jax.random.split(key)
    x_Qrnd = jax.random.multivariate_normal(subkey,
                                            np.zeros(dim),
```

7. Pour le démontrer, on utilise les résultats suivants: le volume de la boule unité en dimension d est $V_d = S_{d-1}/d$, avec la surface de la sphère donnée par $S_{d-1} = 2\pi^{d/2}/\Gamma(d/2)$, et $\int_0^\infty u^d e^{-d^2/2} du = 2^{(d-1)/2}\Gamma((d+1)/2)$.

```
                                                    s**2*np.identity(dim),
                                                    (N,))

    w = prob_nD(x_Qrnd,R)/qapp_nD(x_Qrnd,s)
    norm = jnp.sum(w)
    w /= norm
    phi_samples=phi_nD(x_Qrnd)
    integ = jnp.dot(w,phi_samples)
    return key,integ,w

def test(dim):
    key = jax.random.PRNGKey(0)
    Ns = 10_000
    R = 1
    sigma = R/jnp.sqrt(dim)
    _,integ,w = importance_sampling_nD(key,Ns,dim=dim,R=R,s=sigma)
    w_max = jnp.max(w)
    w_med = jnp.median(w)
    return integ,w_max/w_med
```

En conséquence, si $r \simeq R \pm R/\sqrt{2d}$, alors une estimation du rapport entre la valeur maximale de w et la valeur médiane donne une loi d'échelle de type

$$\frac{w_{max}}{w_{med}} \approx e^{\alpha\sqrt{d}} \tag{13}$$

ce qui est confirmé par l'expérimentation numérique illustrée sur la figure 6 où $\alpha = \sqrt{2}$. Qu'est-ce à dire? Et bien, en grande dimension la dispersion des poids w est telle que la somme Eq. 7 est dominée par quelques tirages avec des valeurs extrêmement larges de w. Ceci est très critique.

On en vient à conclure, par ces deux exemples, que la seule condition pour utiliser cette méthode d'Importance Sampling, est de pouvoir disposer d'une approximation $q(x)$ si proche de celle de $p(x)$, qu'on conviendrait de signifier qu'il s'agit d'un *fine tuning*. C'est quasi-impossible à tenir sauf cas particulier en "basse dimension". Il va donc falloir se tourner vers d'autres méthodes.

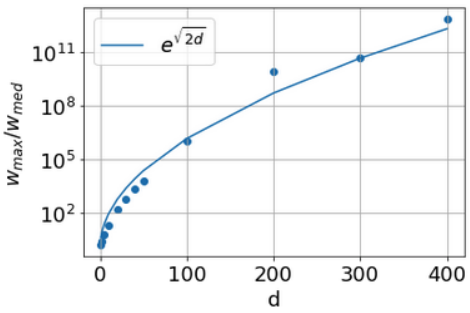

FIGURE 6 – Illustration de la grande dispersion des poids w en grande dimension en utilisant le rapport entre la valeur maximale et la valeur médiane (plus stable que la moyenne empirique).

4 Metropolis-Hastings

Les méthodes précédentes, notamment l'Importance Sampling, utilisent une distribution $q(x)$ (ou sa version non normalisée $\tilde{q}(x)$) définie une fois pour toute, afin de calculer les poids $w_i = \tilde{p}(x_i)/\tilde{q}(x_i)$ (Eq. 6) qui pondèrent les échantillons générés selon $q(x)$ afin de les faire correspondre à la loi $p(x)$. La méthode développée d'abord en 1949 par Nicholas C. Metropolis (1915-99) et Stanisław Ulam (1909-84) fut reprise plus en détails en 1953 par Metropolis et collaborateurs puis étendue en 1970 par Wilfred Hastings (1930-2016). On la baptise l'algorithme de Metropolis-Hastings (MH) même si plusieurs auteurs y ont contribué (dont des mathématiciennes). Il fait partie des algorithmes de Monte Carlo de génération de chaines de Markov (MCMC) constituées de séries d'échantillons aléatoires tirés en adaptant la distribution $q(x)$ à l'échantillon courant. Plus précisément, on génère un nouvel échantillon x_i à l'étape i selon la démarche suivante:

1. on tire un échantillon $\hat{x}$ selon $q(\hat{x}; x_{i-1})$, se peut être une distribution gaussienne de moyenne x_{i-1} par exemple;
2. puis on forme le rapport

$$r = \frac{\tilde{p}(\hat{x})}{\tilde{p}(x_{i-1})} \times \frac{q(x_{i-1}; \hat{x})}{q(\hat{x}; x_{i-1})} \qquad \text{(Metropolis} - \text{Hastings)} \qquad (14)$$

3. enfin on décide de la valeur de x_i selon:

$$\text{Soit } u \sim \mathcal{U}(0,1), \quad \text{si } r > u \Rightarrow x_i = \hat{x}, \text{ sinon } x_i = x_{i-1} \tag{15}$$

L'initialisation du processus se fait par la génération aléatoire de x_0. La distribution $q(x;y)$ est dite *probabilité de transition* de y vers x. Si la distribution $q(x;y)$ est invariante par échange des variables x et y (ex. une gaussienne) alors le rapport r se simplifie

$$r = \frac{\tilde{p}(\hat{x})}{\tilde{p}(x_{i-1})} \qquad \text{(Metropolis)} \tag{16}$$

Même si cette version du rapport r est plus particulièrement dédiée à Metropolis, bien souvent on utilise la terminologie Metropolis-Hastings dans tous les cas.

Remarquons que comme pour l'Importance Sampling, nous n'avons pas besoin de connaitre la normalisation de la distribution $p(x)$, ce qui est particulièrement adapté au cas de figure rencontrés en Mécanique Statistique (MS). Ce fût d'ailleurs la première utilisation de la méthode. L'algorithme de MH est généralement adapté pour des cas en grande dimension autres que ceux de la Mécanique Statistique grâce à son extension due à Hastings. On note qu'avec cette méthode, l'étape i fournit toujours un échantillon soit un nouveau soit l'ancien. Donc, pour N_s tirages il n'y a pas forcément N_s valeurs différentes. La séquence $\{x_i\}_{i<N_s}$ constitue une chaîne de Markov (Andreï A. Markov, 1856-1922) à la manière d'une marche aléatoire dont les échantillons ne sont pas indépendants les uns des autres. Cet inter-dépendance entre échantillons fait qu'il faut en générer beaucoup afin de produire un set indépendant et identiquement distribués (*iid*) selon la distribution cible $p(x)$.

La procédure de rejet/acceptation du nouvel échantillon, indique que l'on va fournir un échantillon identique au précédent surtout dans les régions les plus probables, et fournir un déplacement quand on se trouve dans les régions les moins probables. Ainsi, les zones les plus probables sont les plus visitées ce qui était l'objectif initial.

Une première illustration est donnée dans le notebook à travers une implémentation en `Numpy` permettant de générer plusieurs chaînes simultanément, avec comme probabilité de transition pour générer $\hat{x}$ (*proposal*) une simple densité uniforme centrée sur l'échantillon x_i (*current*) et dont la largeur est $2 \times scale$:

```
def MHSampling_v2p(proba,Ns=100_000,Nchain=10,scale=0.1, forget=25_000):
```

```
    # Ns nbre de samples per chain
    samples = []
    # forget a fixed number of samples for each chain
    Ns = Ns + forget
    current = scs.uniform.rvs(loc=-3,scale=6, size=(Nchain,))
    for i in range(Ns):
        samples.append(current.copy())
        #proposal in [current-scale,current+scale]
        proposal = scs.uniform.rvs(loc=current-scale,scale=2*scale,
            size=(Nchain,))
        # proposal distrib is symetric: q(x_c,x_p)=q(x_c-x_p)=q(x_p,x_c)
        r = proba(proposal)/proba(current)
        u = scs.uniform.rvs(size=(Nchain,))
        mask = r>u #accept proposal
        current[mask] = proposal[mask]

    #return all chains with the first "forget" samples rejected
    samples = np.array(samples).T
    return samples[:,forget:]
```

Les résultats sont présentés sur la figure 7. Sur la figure du haut on remarque le caractère "marche aléatoire": même si le début des 2 chaînes présentées sont assez similaires, il y a des périodes où les 2 chaînes sont franchement différentes. Si l'on compare les distributions individuelles de chaque chaîne (figure du bas), on remarque une certaine volatilité, et ce n'est qu'en combinant les 100 chaînes que l'on réalise la meilleure approximation de $p(x)$. Il s'en suit d'ailleurs que l'erreur relative sur l'approximation de la moyenne de l'intégrale $\int x^2 p(x)dx$ est de l'ordre de 10^{-5} avec 10^7 échantillons au total ce qui est bien meilleur que les estimations présentées sur la figure 4.

Deux difficultés de cette méthode apparaissent à première vue:

— la première concerne le choix de la probabilité de transition, et en particulier sa "largeur" typique: trop petite on risque d'attendre longtemps avant d'explorer les zones importantes de $p(x)$, trop grande on risque au contraire d'en rater. Ce point peut être attaqué par la production de plusieurs chaînes dont chacune peut être affectée de biais mais dont la réunion permet d'obtenir un comportement moyen plus conforme à la probabilité cible. L'exemple présenté en donne une illustration. En dimension d, par un argument similaire à celui développé à la section précédente, on peut formuler une contrainte sur la variance de la probabilité de transition Σ_q

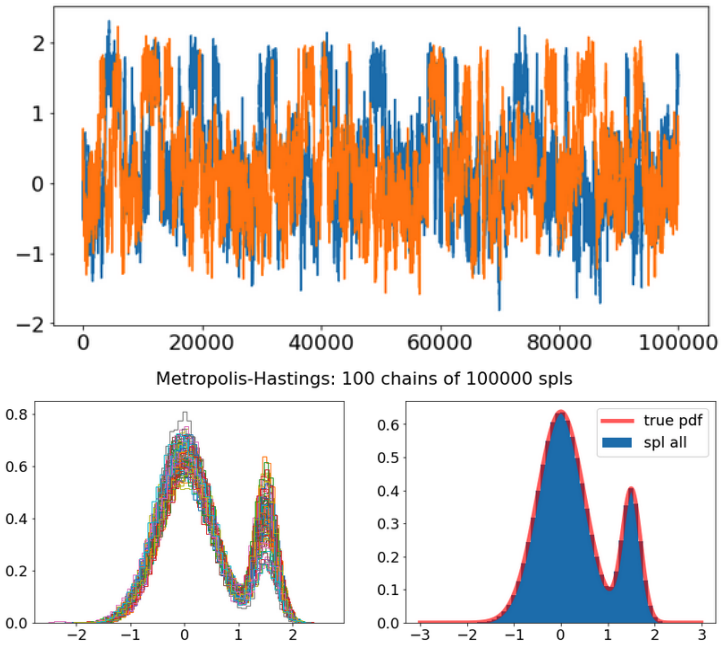

FIGURE 7 – Génération de 100 chaînes de Markov par la méthode Metropolis-Hastings: (figure du haut) évolutions de 2 chaînes au cours de leurs générations; (figure du bas) distributions associées à chaque chaîne (à gauche) et leur combinaison (à droite).

en fonction de la variance de la probabilité cible Σ_p:

$$\Sigma_q \approx \frac{\Sigma_p}{d} \tag{17}$$

Mais comme la variance Σ_p n'est pas connue *a priori*, en fait on procède en général par une procédure itérative de plus haut niveau: après chaque production de N_c chaînes, on en déduit une estimation de $\Sigma_p^{(n)}$ pour adapter $\Sigma_q^{(n)}$ qui sert à la production d'une nouvelle série de chaînes, et ainsi de suite.

— la seconde difficulté et de savoir si les chaînes sont suffisamment longues pour avoir converger afin de fournir des échantillonnages indépendants de $p(x)$. Il s'agit de l'étude mathématique de la vitesse de convergence de l'algorithme. Notons que l'on supprime en général une partie initiale de la chaîne afin de s'affranchir de l'initialisation aléatoire de x_0.

Une littérature abondante traite de ces problèmes et ce n'est pas le lieu ici d'en faire une synthèse. Je vous renvoie vers l'article récent de G. Wang "*Exact Convergence Rate Analysis of the Independent Metropolis-Hastings Algorithms*" [8].

Dans le notebook, je donne une version écrite en `JAX` afin de donner une illustration de l'usage de la compilation `JIT` permet de booster les performances du code et le portage sur GPU est immédiat. Au passage, j'ai procédé à la réécriture de l'algorithme en termes de logarithme des probabilités (ie. *logpdf*).

Pour sortir de la 1D, j'illustre l'usage de l'algorithme MH à travers le problème simple suivant: on dispose d'un nuage de points $\{x_i, y_i\}_{i<n}$ et l'on cherche à trouver une représentation de type:

$$y_i = b + s \times x_i + \varepsilon \equiv f(x_i; b, s) + \varepsilon \tag{18}$$

où (b, s) sont les paramètres classique d'une droite, et ε un bruit blanc gaussien (ie. $\varepsilon \sim \mathcal{N}(0, \sigma)$). Les paramètres que l'on va tenter d'estimer avec des chaînes de Markov sont (b, s, σ), on est donc ici dans un problème en dimension 3. Et l'on voit ainsi, à travers ces types de problèmes, comment la dimension peut largement dépasser le cadre 1D.

Bien entendu, on peut utiliser une minimisation du log-likelihood (plus précisément

8. https://arxiv.org/abs/2008.02455v6

il s'agit schématiquement de $-\log p(\theta|Data))$ suivant:

$$\mathcal{L}(\theta=(\sigma,b,s)|\{x_i,y_i\}_{i<n})) = \frac{n}{2}\log\Big(2\pi\sigma^2\Big) + \frac{\sum_{i=1}^{n}(f(x_i;b,s)-y_i)^2}{2\sigma^2} \tag{19}$$

par une méthode classique de type DFGS [9] (méthode itérative d'optimisation de problème non contraint). Pour mémoire, le notebook donne une illustration de cette méthode à base de la fonction `minimize` de la librairie `scipy`. Cependant, l'idée ici est de se placer dans le cadre de génération de chaînes de Markov via l'algorithme MH pour en tirer les probabilités marginales sur les paramètres individuellement ainsi que les lois conjointes 2-à-2. Et je vais en profiter pour utiliser la librairie `JAX`.

Dans un premier temps, le log-likelihood est complété par les log-pdf des *priors* sur les 3 paramètres (rappel: nous sommes dans un cadre bayésien dès lors que l'on introduit un modèle de "bruit" sur les données):

```
def jax_my_logpdf(par,xi,yi):
    # priors: a=par[0], b=par[1], sigma=par[2]
    logpdf_a = jax.scipy.stats.norm.logpdf(x=par[0],loc=0.5,scale=1.)
    logpdf_b = jax.scipy.stats.norm.logpdf(x=par[1],loc=2.5,scale=1.)
    logpdf_s = jax.scipy.stats.gamma.logpdf(x=par[2],a=3,scale=1.)

    val = xi*par[1]+par[0]
    tmp = jax.scipy.stats.norm.logpdf(x=val,loc=yi,scale=par[2])
    log_likeh= jnp.sum(tmp)

    return log_likeh + logpdf_a + logpdf_b + logpdf_s
```

Notons au passage que nous n'avons pas besoin de connaitre la normalisation de la distribution de probabilité conjointe (paramètre, data) pour utiliser l'algorithme MH. Ensuite, la loi de transition $q(x)$ est choisi comme étant une loi normale (en 3D) de $\sigma = 0.1$. Voici le code qui génère un nouvel état (ou position) de la chaîne (θ_i) connaissant l'état antérieur (θ_{i-1}):

```
def jax_metropolis_kernel(rng_key, logpdf, position, log_prob):
    key, subkey = jax.random.split(rng_key)
    """Moves the chain by one step using the Random Walk Metropolis
       algorithm.
    """
```

9. Pour Broyden-Fletcher-Goldfarb-Shanno

```
    move_proposals = jax.random.normal(key, shape=position.shape) * 0.1

    proposal = position + move_proposals
    proposal_log_prob = logpdf(proposal)

    log_uniform = jnp.log(jax.random.uniform(subkey))
    do_accept = log_uniform < proposal_log_prob - log_prob

    position = jnp.where(do_accept, proposal, position)
    log_prob = jnp.where(do_accept, proposal_log_prob, log_prob)
    return position, log_prob
```

Le moteur principal qui part d'une position initiale et génère petit-à-petit le nombre d'états/échantillons des chaînes se présente comme suit:

```
def jax_metropolis_sampler(rng_key, n_samples, logpdf, initial_position):
    """Generate samples using the Random Walk Metropolis algorithm.
    """
    def mh_update(i, state):
        key, positions, log_prob = state
        _, key = jax.random.split(key)

        new_position, new_log_prob = jax_metropolis_kernel(key,
                                                           logpdf,
                                                           positions[i-1],
                                                           log_prob)


        positions=positions.at[i].set(new_position)
        return (key, positions, new_log_prob)

    #Initialisation
    all_positions = jnp.zeros((n_samples,initial_position.shape[0]))
    all_positions=all_positions.at[0,0].set(scs.norm.rvs(loc=1,scale=1))
    all_positions=all_positions.at[0,1].set(scs.norm.rvs(loc=2,scale=1))
    all_positions=all_positions.at[0,2].set(scs.uniform.rvs(loc=1,scale=2))

    logp = logpdf(all_positions[0])

    initial_state = (rng_key,all_positions, logp)
    rng_key, all_positions, log_prob = jax.lax.fori_loop(1, n_samples,
```

```
                                                    mh_update,
                                                    initial_state)

    return all_positions
```

Enfin, la vectorisation de cette dernière méthode permet de processer plusieurs chaînes simultanément:

```
run_mcmc = jax.vmap(jax_metropolis_sampler,
                    in_axes=(0, None, None, 1),    # see comment above
                    out_axes=0)                    # output axis 0 hold the
                        vectorization over n_chains
                                                    # => (n_chains,
                                                        n_samples, n_dims)


all_positions = run_mcmc(rng_keys, n_samples,
                         lambda par: jax_my_logpdf(par,xi,yi),
                         initial_position)
```

Sur CPU, la génération de 100 chaînes contenant 10^5 échantillons chacune devrait prendre 1 minute environ, peut-être moins. Ensuite, dans le notebook je donne plusieurs façons d'obtenir des informations sur les chaînes ainsi générées, et notamment la figure 8 donne les distributions des marginales [10]. On obtient des rapports $\sigma_{\theta_i}/\theta_i \approx 1\%$ avec l'ensemble des 10^7 échantillons. Si l'on change la taille de la probabilité de transition en la multipliant par 10 (en passant de 0.1 à 1), comme on peut le constater en modifiant et rejouant la cellule du notebook qui contient

```
move_proposals = jax.random.normal(key, shape=position.shape) * 0.1
```

alors même si les valeurs moyennes ne sont pas si différentes, on apprécie néanmoins l'effet du changement à travers les écarts-standards et autres statistiques.

On met ainsi le doigt sur un point intrinsèquement délicat de cette méthode: quelle taille σ_q typique doit avoir la probabilité de transition (ie. la distribution $q(x)$)? Il est à remarquer que l'ensemble de la chaîne constituée des différents états x_i, est une sorte de marche aléatoire dans un espace à d-dimensions. Or, dans le cadre des "Random Walk", un résultat est que le nombre de steps N_s de taille ε qu'il faut pour se trouver à une

10. Les fonctions simples utilisées ne donnent pas accès aux contours traditionnels à $n\sigma$ ou bien à x%CL, à la section 5.2, j'utilise la librairie `corner` à cette fin.

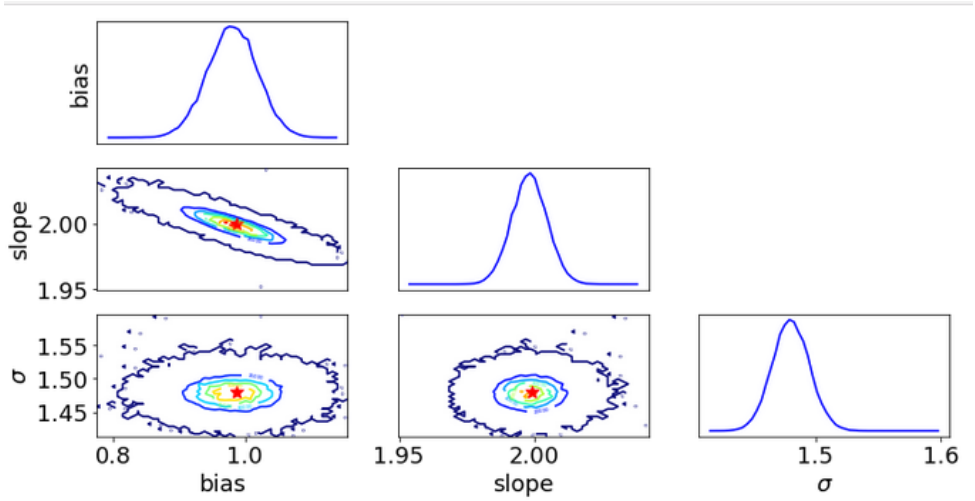

FIGURE 8 – A partir des chaînes de Markov, on peut obtenir les distributions marginales sur les paramètres et leurs corrélations.

distance L du point de départ, est donné par la loi $N_s = (L/\varepsilon)^2$. Appliqué au cas qui nous occupe, alors N_s suit une loi quadratique telle que $N_s = (\sigma_p^{max}/\sigma_q)^2$ avec σ_p^{max} est la composante maximale de la matrice de covariance de la distribution $p(x)$. D'un certain coté, l'indépendance vis-à-vis de la dimensionalité contraste avec le résultat de la section précédente. Mais en pratique, on ne connait pas *a priori* σ_p^{max} et si $\sigma_q \gg \sigma_p^{min}$ alors on rate les features de $p(x)$ à la taille de σ_p^{min}. Donc, la méthode ne se suffit pas en elle-même, puisque l'utilisateur doit en partie tâtonner pour trouver le bon set d'hyper-paramètres (distribution de transition, sa taille, l'initialisation des chaînes, leur nombre, leurs tailles individuelles...). Et *in fine*, on peut devoir recourir à la production de longues chaînes pour obtenir une bonne représentation de la distribution $p(x)$ avec l'algorithme de Metropolis-Hastings.

5 Monte Carlo Hamiltonien: HMC et NUTS

Dans cette section, je vais introduire une méthode Monte Carlo qui se veut être une réponse à la question: peut-on réduire le temps nécessaire pour qu'une chaîne de Markov

puisse donner un lot d'échantillons indépendants issues d'une distribution cible [11] $p_c(x)$. Dans la suite je ne fait qu'effleurer quelques notions et je renvois par exemple vers l'article de Michael Betancourt [12] (2018) pour plus de détails à ce sujet.

5.1 Premiers pas à la main

Une première remarque est que dans la plus part des cas, on peut concevoir que la distribution $p_c(x)$ peut se mettre sous la forme générique suivante:

$$p_c(x) = \frac{e^{-V(x)}}{Z} \Leftrightarrow -\log(p_c(x)) = V(x) + Cte \tag{20}$$

Par exemple, en statistique bayésienne, la distribution *a posteriori* des paramètres d'un modèle (nb. ici les valeurs de x sont celles des dits paramètres comme à la section précédente), $V(x) = -\log(\pi(x)\mathcal{L}(x|\mathcal{D}))$, avec $\pi(x)$ le *prior* et $\mathcal{L}$ le likelihood compte tenu du data set $\mathcal{D}$. De plus, on suppose par la suite que l'on sait *calculer le gradient* de $V(x)$.

Dans la technique HMC, on ajoute une variable conjuguée à x, que l'on note opportunément p et l'on forme la fonction (hamiltonien)

$$H(x,p) = V(x) + E_c(p) \tag{21}$$

avec $E_c(p) = \|p\|_2^2/2$ représentant une sorte d'énergie cinétique du système quand $V(x)$ en représente l'énergie potentielle. Les équations dynamiques du système sont alors identiques à celle de la Mécanique classique de Hamilton:

$$\dot{x} = \frac{\partial H}{\partial p} = \frac{\partial E_c}{\partial p} = p \qquad \dot{p} = -\frac{\partial H}{\partial x} = -\frac{\partial V}{\partial x} = \frac{\partial \log p_c(x)}{\partial x} \tag{22}$$

L'algorithme HMC se décline autour de la version simplifiée suivante pour discuter des ingrédients. Partant d'un état x_i d'une chaîne pour obtenir l'état x_{i+1}, on procède ainsi

1. tirage aléatoire de $p_i \sim \mathcal{N}(0,1)$; puis l'on initialise $p_{new} = p_i$ et symétriquement $x_{new} = x_i$;

11. Le changement de notation de $p(x)$ à $p_c(x)$ pour la distribution cible est motivée par les équations classiques de Mécanique Hamiltonienne faisant intervenir la variable "moment" p.

12. Michael Betancourt, *A Conceptual Introduction to Hamiltonian Monte Carlo*, https://arxiv.org/pdf/1701.02434.pdf

2. itération de n_{steps} étapes d'intégration des équations du mouvement suivant la méthode dite de l'algorithme *leapfrog* en anglais ou *saute-mouton* en français:

$$\begin{cases} p_{new} & = p_{new} - \varepsilon \times \frac{1}{2} \frac{\partial V(x)}{\partial x}|_{x=x_{new}} \\ x_{new} & = x_{new} + \varepsilon \times p_{new} \\ p_{new} & = p_{new} - \varepsilon \times \frac{1}{2} \frac{\partial V(x)}{\partial x}|_{x=x_{new}} \end{cases} \tag{23}$$

   (notons la mise à jour du gradient en cours de route);

3. puis, on renverse le moment $p_{new} = -p_{new}$;
4. enfin, on procède comme pour l'algorithme de Metropolis selon la probabilité d'acceptation du nouvel état (x_{new}, p_{new}):

$$p_{acc} = \min \left\{ 1, r = \frac{\tilde{P}(x_{new}, p_{new})}{\tilde{P}(x_i, p_i)} \right\} \tag{24}$$

   avec

$$\tilde{P}(x, y) = \exp\{-H(x, p)\} = \exp\{-V(x)\} \exp\{-E_c(p)\} \tag{25}$$

   Cela se fait par tirage uniforme d'un nombre u dans l'intervalle $[0, 1]$. Si $u < r$ on accepte x_{new} comme valeur pour x_{i+1}, sinon on prend x_i. Remarquons le découplage qu'il y a entre les deux variables x et p. Notons aussi, que l'on ne garde pas p_{new} par la suite.

Pourquoi à première vue utiliser des équations d'Hamilton? Les arguments sont les suivants: la *réversibilité* temporelle de la dynamique hamiltonienne est importante pour la réversibilité de la chaîne de Markov (voir après), ensuite la *conservation* de l'Hamiltonien au cours du temps fait qu'en principe $r = 1$ et donc favorise l'acceptation d'un nouvel état de la chaîne (mais en pratique la conservation de H n'est pas exacte ne serait-ce que par les imperfection de l'algorithme d'intégration); la proposition du nouvel état est guidée par p_{new}, or ce dernier pointe dans *la direction de plus forte densité* de probabilité, ce qui va dans le bon sens. Enfin, la dynamique hamiltonienne *conserve les volumes dans l'espace des phases*, c'est le théorème de Liouville. Cette propriété est cruciale pour pouvoir utiliser la méthode de Metropolis (étape 4) afin d'accepter ou non un nouvel état de la chaîne. Maintenant, deux questions: pourquoi l'algorithme (HMC) converge vers $p_c(x)$, et pourquoi a-t'on cette structure étrange de l'algorithme de saute-mouton, notamment le renversement du moment à la fin?

La première question réfère à la notion de *detailed balance*[13] introduite par James Clerk Maxwell et Ludwig Boltzmann en Mécanique Statistique et stipule dans notre cas que:

$$p_c(x)p_t(x \to y) = p_c(y)p_t(y \to x) \tag{26}$$

où p_c est la probabilité cible et $p_t(x \to y)$ la probabilité de transition de la position x vers la position y. Que vaut cette probabilité de transition dans le cas de HMC? Voyons cela: si l'on part d'un état (x_0, p_0) au bout de T étapes (à la manière de discrétisation du temps) on aboutit à l'état (x_T, y_T). Cependant, la dynamique hamiltonienne est déterministe[14], donc il existe deux fonctions f_1 et f_2 telles que

$$x_T = f_1(x_0, p_0) \qquad p_T = f_2(x_0, p_0) \tag{27}$$

Or, la *réversibilité* des équations dynamiques nous dit que si l'on part de l'état $(x_T, -p_T)$ alors en T étapes on retombe sur l'état initial (x_0, p_0). Cela impose alors que

$$x_0 = f_1(x_T, -p_T) \qquad p_0 = f_2(x_T, -p_T) \tag{28}$$

ce qui confère alors une correspondance bi-univoque entre (x_0, p_0) et (x_T, p_T) d'une part et $(x_T, -p_T)$ d'autre part. Supposons à présent qu'il n'y ait qu'un unique moment $p^{(x,y)}$ tel que l'on ait entre les positions x et y le mapping suivant $y = f_1(x, p^{(x,y)})$, et en même temps notons $p_y = f_2(x, p^{(x,y)})$. La réversibilité nous dit alors que $x = f_1(y, -p_y)$ et $p^{(x,y)} = f_2(y, -p_y)$. Ainsi, il n'y a qu'un moment $p^{(y,x)}$ qui relie $(x, p^{(x,y)})$ et $(y, p^{(y,x)})$ c'est $p^{(y,x)} = -p_y$. Donc, $p_t(x \to y) = \pi(p^{(x,y)})$ ($\pi(p) \propto e^{-E_c(p)}$) car seule la donnée du moment fait bouger la valeur de la position, et d'une manière similaire on identifie

13. principe du *bilan détaillé* qui compare les probabilités de processus inverses $A \to B$ et $B \to A$. Notons qu'il existe des versions de HMC qui ne satisfont pas ce critère car il n'est pas nécessaire pour obtenir une distribution stable vers laquelle l'algorithme converge: voir par exemple https://arxiv.org/pdf/1409.5191.pdf. Au passage, on peut démontrer que l'algorithme de Metropolis-Hastings satisfait cette propriété de *detailed balance*.

14. nb. ce qui n'exclut pas la chaos, mais c'est une autre histoire.

$p_t(y \to x) = \pi(-p_y)$ [15]. Maintenant, on peut démontrer la relation 26:

$$\begin{aligned}
p_c(x)p_t(x \to y) &= p_c(x)\pi(p^{(x,y)}) \\
&= \frac{1}{Z}e^{-V(x)-E_c(p^{(x,y)})} = \frac{1}{Z}e^{-H(x,p^{(x,y)})} \\
&= \frac{1}{Z}e^{-H(y,p_y)} && \text{(H = Cte)} \\
&= \frac{1}{Z}e^{-V(y)-E_c(p_y)} \\
&= p_c(y)\pi(p_y) \\
&= p_c(y)\pi(-p_y) && \text{(E_c(p) sym. p} \leftrightarrow -\text{p))} \\
&= p_c(y)p_t(y \to x) && (29)
\end{aligned}$$

Ayant cette propriété et sachant que $\int p_t(y \to x)dx = 1$ à savoir que somme des transitions possibles de la position y vers les autres états accessibles vaut naturellement 1, alors

$$\begin{aligned}
\int p_c(x)p_t(x \to y)dx &= \int p_c(y)p_t(y \to x)dx \\
&= p_c(y)\int p_t(y \to x)dx = p_c(y) && (30)
\end{aligned}$$

Donc, p_c est point fixe (stationnaire) de l'équation à noyau dont le kernel est la probabilité de transition. Par définition, la chaîne de Markov des positions successives engendrées par l'application de la probabilité de transition p_t est réversible par rapport à la probabilité cible p_c.

Maintenant concernant la seconde question, inspectons la partie de l'algorithme qui génère la proposition (x_{new}, p_{new}). D'abord à quoi correspond l'étape 1? En fait, elle fait référence à un tirage de p_{new} selon la probabilité la probabilité $e^{-E_c(p)}$ qui n'est autre que la première étape d'un échantillonnage de Gibbs. En effet, cette méthode que je n'ai pas mise en œuvre permet d'échantillonner une loi de probabilité $P(\mathbf{x} = (x_1, x_2, \ldots, x_K))$

15. J'en conviens cela beaucoup de *p*...

selon un processus itératif où l'on tire une variable à la fois

$$\begin{aligned}
x_1^{new} &\sim P(x_1|x_2^i,\ldots,x_K^i)\\
x_2^{new} &\sim P(x_2|x_1^{new},x_3^i,\ldots,x_K^i)\\
&\vdots\\
x_k^{new} &\sim P(x_k|x_1^{new},x_2^{new},\ldots,x_{k-1}^{new},x_{k+1}^i,\ldots,x_K^i)\\
&\vdots
\end{aligned} \tag{31}$$

L'échantillonnage de Gibbs est un exemple d'algorithme de Metropolis où l'on accepte toujours le nouvel échantillon, et la chaîne de Markov générée converge vers la distribution $P(\mathbf{x})$. Or, dans le cas de HMC, les deux variables x et p sont indépendantes (Eq. 25), d'où la génération de p selon $e^{-E_c(p)}$. Un point anecdotique est que HMC est aussi le sigle pour Hybrid Monte Carlo car comme on vient de le voir on utilise une étape de Gibbs pour tirer p et ensuite la dynamique hamiltonienne pour mettre à jour x et p simultanément. L'ensemble des étapes 2 est une forme discrétisée des équations du mouvement, elles rappellent l'algorithme d'Euler par exemple. Le plus intrigant est l'étape 3, à quoi sert-elle? Reprenons les étapes d'une itération dans le sens + de la progression de (x_0, p_0) vers un nouvel état (l'étape intermédiaire est noté avec 1/2):

$$\begin{aligned}
\hat{p}_{1/2}^+ &= p_0^+ - \varepsilon/2\ \nabla V(x_0^+)\\
\hat{x}_1^+ &= x_0^+ + \varepsilon\ \hat{p}_{1/2}^+\\
\hat{p}_1^+ &= \hat{p}_{1/2}^+ - \varepsilon/2\ \nabla V(x_1^+)
\end{aligned} \tag{32}$$

et finalement on passe donc de (x_0^+, p_0^+) à $(\hat{x}_1^+, -\hat{p}_1^+)$. Si l'on reprend la même démarche mais en partant de $(x_0^-, p_0^-) = (\hat{x}_1^+, -\hat{p}_1^+)$ dans le sens inverse alors il vient pour le premier demi-step du moment:

$$\begin{aligned}
\hat{p}_{1/2}^- &= p_0^- - \varepsilon/2\ \nabla V(x_0^-)\\
&= -\hat{p}_1^+ - \varepsilon/2\ \nabla V(\hat{x}_1^+)\\
&= -\left(\hat{p}_{1/2}^+ - \varepsilon/2\ \nabla V(x_1^+)\right) - \varepsilon/2\ \nabla V(x_1^+)\\
&= -\hat{p}_{1/2}^+
\end{aligned} \tag{33}$$

On va bien dans le sens inverse... Concernant la position

$$\begin{aligned}\hat{x}_1^- &= x_0^- + \varepsilon\ \hat{p}_{1/2}^- \\ &= \hat{x}_1^+ - \varepsilon\ \hat{p}_{1/2}^+ \\ &= x_0^+ + \varepsilon\ \hat{p}_{1/2}^+ - \varepsilon\ \hat{p}_{1/2}^+ \\ &= x_0^+ \end{aligned} \tag{34}$$

donc on revient à la position originale. Finalement, en effectuant le second demi-step du moment, il vient

$$\begin{aligned}\hat{p}_1^- &= p_{1/2}^- - \varepsilon/2\ \nabla V(x_1^-) \\ &= -\hat{p}_{1/2}^+ - \varepsilon/2\ \nabla V(x_0^+) \\ &= -p_0^+ + \varepsilon/2\ \nabla V(x_0^+) - \varepsilon/2\ \nabla V(\hat{x}_0^+) \\ &= -p_0^+ \end{aligned} \tag{35}$$

Ainsi, après l'étape du renversement du moment, on retrouve également la valeur originale p_0^+. Donc, l'étape de renversement du moment permet de rendre compte de la réversibilité des lois de la dynamique hamiltonienne qui est à la base de la propriété de convergence de l'algorithme. Cependant, en pratique l'expression de l'énergie cinétique étant quadratique en p, changer le signe n'a aucune incidence sur la probabilité d'acceptation de la nouvelle position.

Ainsi, on peut comprendre les différentes étapes de l'algorithme HMC (dans sa version la plus simple). Un point cependant concerne le cœur de l'algorithme saute-mouton: il nous faut connaître le gradient $\nabla V(x)$, ce qui peut donner lieu de nouveau à une approximation si jamais on a recours à une méthode par différence finie. C'est là où la *différentiation automatique* comme avec `JAX` va venir en aide. Dans un premier temps, je donne dans le notebook une implémentation simple de l'algorithme HMC, avec une application à la génération d'échantillons selon une simple loi normale. On peut se faire la main en jouant sur les paramètres `step_size` (aka ε) et `n_steps`, pour se rendre compte que la distribution des échantillons est des plus versatiles. Il faut donc développer une stratégie pour stabiliser les résultats. Un troisième paramètre que l'on peut également introduire concerne la "masse équivalente" qui entre dans la distribution normale du mo-

ment p, en effet on peut tout aussi bien considérer une expression de l'énergie cinétique comme $E_c = \|p\|^2/(2m)$ ce qui revient à dire que $p \sim \mathcal{N}(0, \sigma_p = \sqrt{m})$. Là aussi, on peut constater la sensibilité des résultats.

La littérature présente des méthodes adaptatives pour contrôler ε au fur et à mesure de l'évolution de la chaîne. On montre que l'erreur sur le rapport $|H(x_{new}, p_{new})/H(x_i, p_i)|$ qui intervient à l'étape 4 est en ε^3 si $n_{steps} = 1$ et en ε^2 pour $n_{steps} \gg 1$, donc il faudrait privilégier des ε petit. Mais le nombre de steps (`n_steps`) reste un problème: trop petit la chaîne a une évolution de type marche aléatoire (trop de stochasticité), trop grand on utilise des ressources inutilement, mais surtout cela peut faire en sorte que la trajectoire de la chaîne boucle sur elle-même ce qui génère des échantillons "proche" les uns des autres comme si la chaîne stagnait, voire même on peut faire perdre tout simplement les propriétés de convergence de la chaîne (non-ergodicité). Donc la détermination de n_{steps} est délicate et demande des simulations préparatoires et beaucoup de pratique. Ce point dur rendait la pratique de la méthode HMC difficile, enfin avant la mise au point de l'algorithme NUTS [16] en 2011.

NUTS, pour No-U-Turn Sampler, construit une procédure qui assure la distance maximale entre la position x_i et la potentielle nouvelle position x_{new}. On réalise que si l'on part de x_i, la dynamique hamiltonienne nous donne que la variation de $\|x_i - x(t)\|^2$ satisfait l'équation

$$C(t) = \frac{1}{2}\frac{d}{dt}\|x(t) - x_i\|^2 = (x(t) - x_i).\frac{dx(t)}{dt} = (x(t) - x_i).p(t) \tag{36}$$

Ainsi, on peut se rendre compte au fur et à mesure de la progression dans l'algorithme saute-mouton si le terme de droite devient négatif, car alors cela indiquerait un point de rebroussement de la trajectoire. Cependant, si on ne simule que la partie où $C(t) > 0$ on a un problème avec la réversibilité. Pour palier ce problème, l'algorithme NUTS introduit une variable auxiliaire et construit un arbre binaire de mouvements forward (direction + utilisé plus haut) et backward ($-$) afin de donner sur une profondeur 2^n des propositions de mouvement. Mais l'algorithme complet est complexe donc je vous renvoie à l'article des auteurs. Pour un utilisateur, c'est là où la librairie `NumPyro` [17]

16. Matthew D. Hoffman et Andrew Gelman (2011), *The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo*, arXiv:1111.4246, https://arxiv.org/abs/1111.4246

17. Site web: http://num.pyro.ai; Du Phan, Neeraj Pradhan and Martin Jankowiak, *Composable Effects*

va être utile, et la section suivante donne deux exemples d'utilisation. Notons avant de passer à la pratique que `NumPyro` n'est pas la seule librairie sur le marcher pour traiter de la génération de chaînes de Markov par la méthode HMC/NUTS. L'usage d'autres librairies est par exemple discuté dans ce post datant de 2018: https://mattpitkin.github.io/samplers-demo/pages/samplers-samplers-everywhere/.

5.2 Librairie `NumPyro`

La librairie `NumPyro` est la version `JAX` de la librairie `Pyro` de calcul probabiliste [18] dont le moteur était `Pytorch`. Dans la suite, je ne vais utiliser que les méthodes d'échantillonnage de type HMC et NUTS mais sachez qu'il y en a d'autres à disposition.

Pour un premier cas, je vais utiliser la classe suivante:

```
class MixtureModel_jax():
    def __init__(self, locs, scales, weights, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.loc = jnp.array([locs]).T
        self.scale = jnp.array([scales]).T
        self.weights = jnp.array([weights]).T
        norm = jnp.sum(self.weights)
        self.weights = self.weights/norm

        self.num_distr = len(locs)

    def logpdf(self, x):
        log_probs = jax.scipy.stats.norm.logpdf(x,loc=self.loc,
            scale=self.scale)
        return jax.scipy.special.logsumexp(jnp.log(self.weights) +
            log_probs, axis=0)[0]

    def pdf(self, x):
        probs = jax.scipy.stats.norm.pdf(x,loc=self.loc, scale=self.scale)
        return jnp.dot(self.weights.T,probs).squeeze()
```

d'abord dans le cas d'une simple distribution normale 1D:

for Flexible and Accelerated Probabilistic Programming in NumPyro https://arxiv.org/pdf/1912.11554.pdf

18. https://pyro.ai/

```
single_gaussian_model = MixtureModel_jax([0],[1],[1])
```

L'usage de la méthode HMC pour l'échantillonnage se fait comme suit

```
rng_key = jax.random.PRNGKey(42)
_, rng_key = jax.random.split(rng_key)

kernel = HMC(potential_fn=lambda x: -single_gaussian_model.logpdf(x))  #
    negative log
num_samples = 1_000_000
n_chains    = 1
mcmc = MCMC(kernel, num_warmup=2_000, num_samples=num_samples,
            num_chains=n_chains,
            progress_bar=False)
mcmc.run(rng_key, init_params=jnp.zeros(n_chains))
mcmc.print_summary()
samples_1 = mcmc.get_samples()
```

(nb. pour 10^6 échantillons il vaut mieux ne pas produire de barre de progression, mais avec nettement moins de d'échantillons c'est tout à fait praticable), notons que l'usage de NUTS est tout aussi simple car l'API est la même. En l'occurrence avec cet exemple très simple, il n'y a pas de différences entre les deux méthodes. On obtient une bonne distribution des échantillons (voir notebook), et l'on obtient une erreur relative sur $\langle x^2 \rangle$ de l'ordre de 10^{-3}.

Prenons à présent, le mélange de deux gaussiennes que nous avons déjà rencontré:

```
mixture_gaussian_model = MixtureModel_jax([0,1.5],[0.5,0.1],[8,2])
```

La génération de 10^6 échantillons se fait de la même façon que précédemment à la différence qu'il faut changer le modèle dans `kernel` (nb. rien n'empêche d'avoir un code plus générique). La distribution des échantillons semble tout à fait satisfaisante *a priori*.

Cependant, on commence à avoir un doute quand on calcule $\langle x^2 \rangle$. Si on génère 1 chaîne de 10^5 échantillons, on obtient une erreur relative de 4 10^{-2} ou 3 10^{-2} en utilisant respectivement HMC ou NUTS. Or, ces niveaux sont plutôt attendus pour des lots de $O(5\ 10^3)$ échantillons. Il faut faire attention coté utilisateur à rendre aléatoire la position initiale, et donc à utiliser plusieurs chaînes. Si par exemple, on utilise 100 chaînes dont les valeurs initiales sont aléatoires (uniforme) dans l'intervalle $[-3, 3]$ alors avec 10^4 échantillons pour chaque chaîne, on obtient une erreur relative de 2 10^{-2} avec HMC et

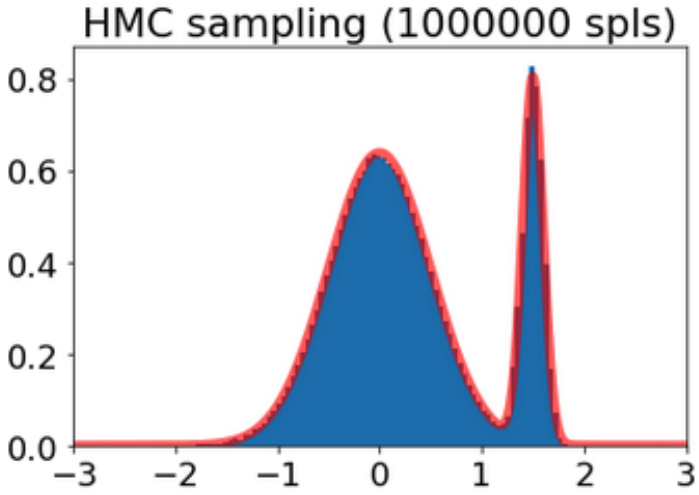

FIGURE 9 – Génération de 10^6 échantillons d'une densité de probabilité ayant 2 modes, selon la méthode HMC. Avec NUTS nous ne pouvons pas voir de différences à ce stade.

5 10^{-4} pour NUTS. Bon, si l'on commence à comprendre la nécessité de processer plusieurs chaînes avec des positions initiales différentes, d'où vient la différence entre HMC et NUTS?

Une des façons de voir le problème est de construire l'auto-correlation en fonctions d'un décalage (`lag`) "temporel". La figure 10 montre que les lots d'échantillons HMC ou NUTS présentent un spectre d'auto-corrélation ayant une valeur > 0.5 respectivement pour $lag < 10$ (NUTS) et $lag < 30$ (HMC). Les échantillons sont donc corrélés comme on s'y attend pour des chaînes de Markov, mais l'échantillon obtenu par NUTS l'est sensiblement moins ce qui tend à donner un meilleur résultat. Il y a un paramètre qui rend compte en quelque sorte de cette corrélation entre échantillons (voir `mcmc.print_summary()`), c'est `n_eff` le *nombre effectif* d'échantillons. En l'occurrence, on trouve également un facteur 3 plus grand entre `n_eff` obtenu avec NUTS comparativement à celui obtenu avec HMC. Il faut donc avoir un œil sur ce paramètre. Un autre facteur qui teste *la convergence* de(s) la(les) chaîne(s) est celui de Gelman-Rubin ($\hat{R}$) [19]: il calcule les variances entre chaînes et au sein de chaque chaîne pour en tirer un score qui doit être proche de 1 (nb. les exemples HMC/NUTS avec 1 ou 100 chaînes précédent ont un score $\hat{R} \approx 1$ conforme à un bon critère de convergence). Pour une discussion plus détaillée sur les outils de diagnostic,

19. Voir https://bookdown.org/rdpeng/advstatcomp/monitoring-convergence.html

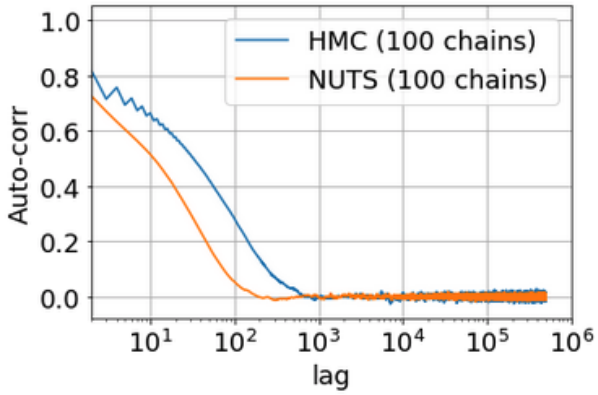

FIGURE 10 – Graphes des auto-corrélations des chaînes de type HMC ou NUTS en fonction du décalage (`lag`. Dans cette représentation, un signal constitué d'échantillons indépendant les uns des autres n'a qu'un pic pour $lag = 0$ d'intensité égale à 1.

je vous renvoie vers le livre de A. Gelman et al. mis à jour en 2021 et accessible en ligne [20]. Maintenant, le problème que l'on essaye de résoudre ici fait parti de la série des problèmes où la distribution comporte plusieurs "modes" (*multi-modal distribution*) pour lesquels il est délicat d'utiliser des MCMCs tels quels, il faudrait un autre note pour traiter de ce cas un peu plus en détails.

Un autre exemple pour illustrer l'usage de `NumPyro` est celui de la détermination de paramètres comme à la section 4. Je l'étend juste au cas de 5 paramètres (voir détails dans le notebook):

```
yi = 1.0 + 0.2 * xi**2 + 0.5*xi**3 +  e
```

(pour mémoire e modélise un bruit blanc gaussien de variance σ_e^2). Pour utiliser `NumPyro`, on construit un modèle où 4 paramètres (θ_i) ont un *prior* gaussien de sigma assez large, et le cinquième (σ_e) un *prior* uniforme également assez large. Quant au likelihood, il est modélisé selon

$$p(\theta|\mathcal{D}) \propto \exp\left\{-\frac{\sum_i \mu(\theta, x_i)^2 - y_i}{2\sigma_e^2}\right\} \tag{37}$$

avec $\mu(\theta, x) = \sum_{k=0}^{3} \theta_k x^k$. Tout cela se code comme suit:

20. Andrew Gelman, John B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, et Donald B. Rubin *Bayesian Data Analysis Third edition*, http://www.stat.columbia.edu/~gelman/book/BDA3.pdf

```
def my_model(Xspls, Yspls):
    a0 = numpyro.sample('a0', dist.Normal(0.,10))
    a1 = numpyro.sample('a1', dist.Normal(0.,10))
    a2 = numpyro.sample('a2', dist.Normal(0.,10))
    a3 = numpyro.sample('a3', dist.Normal(0.,10))
    sigma = numpyro.sample('sigma', dist.Uniform(low=0.,high=10.))
    mu = a0 + a1*Xspls + a2*Xspls**2 + a3*Xspls**3
    numpyro.sample('obs', dist.Normal(mu, sigma), obs=Yspls)
```

On y reconnait les *priors* et le *likelihood*. Ensuite, pour tirer 5000 échantillons selon la distribution postérieure, on procède par une méthode similaire à celle utilisée pour l'exercice de la distribution gaussienne, seulement on doit fournir le dataset $\mathcal{D} = \{x_i, y_i\}_{i<N}$ avec dans l'exemple $N = 5000$:

```
rng_key = jax.random.PRNGKey(0)
_, rng_key = jax.random.split(rng_key)

# Run NUTS.
kernel = NUTS(my_model, init_strategy=init_to_sample())
num_samples = 5_000
mcmc = MCMC(kernel, num_warmup=1000, num_samples=num_samples,
    progress_bar=False)
mcmc.run(rng_key, Xspls=xi, Yspls=yi)
mcmc.print_summary()
samples_1 = mcmc.get_samples()
```

En passant, vous noterez que j'ai du spécifier la "stratégie" d'initialisation des paramètres: celle par défaut est en fait un tirage aléatoire uniforme dans une boite dont le "range" est de 2, ce qui n'est pas ce que l'on attend en général. La "stratégie" `init_to_sample` utilise la formulation des *priors* du modèle.

Maintenant, en regardant les paramètres de diagnostic, on vérifie bien que la chaîne a convergé, et la figure 11 présente les résultats des *posteriors* avec les contours (θ_i, θ_j) $(i \neq j)$ à 68% et 95% ainsi que les quantiles correspondants pour les distributions 1D de chaque θ_i. Incidemment, j'ai utilisé la librairie `corner`[21]. Si on augmente le nombre d'échantillons de la chaîne cela affine les contours à 68% et 95% mais ne change pas fondamentalement leur allure, tandis que leur taille dépend du nombre de points N du dataset.

21. https://corner.readthedocs.io

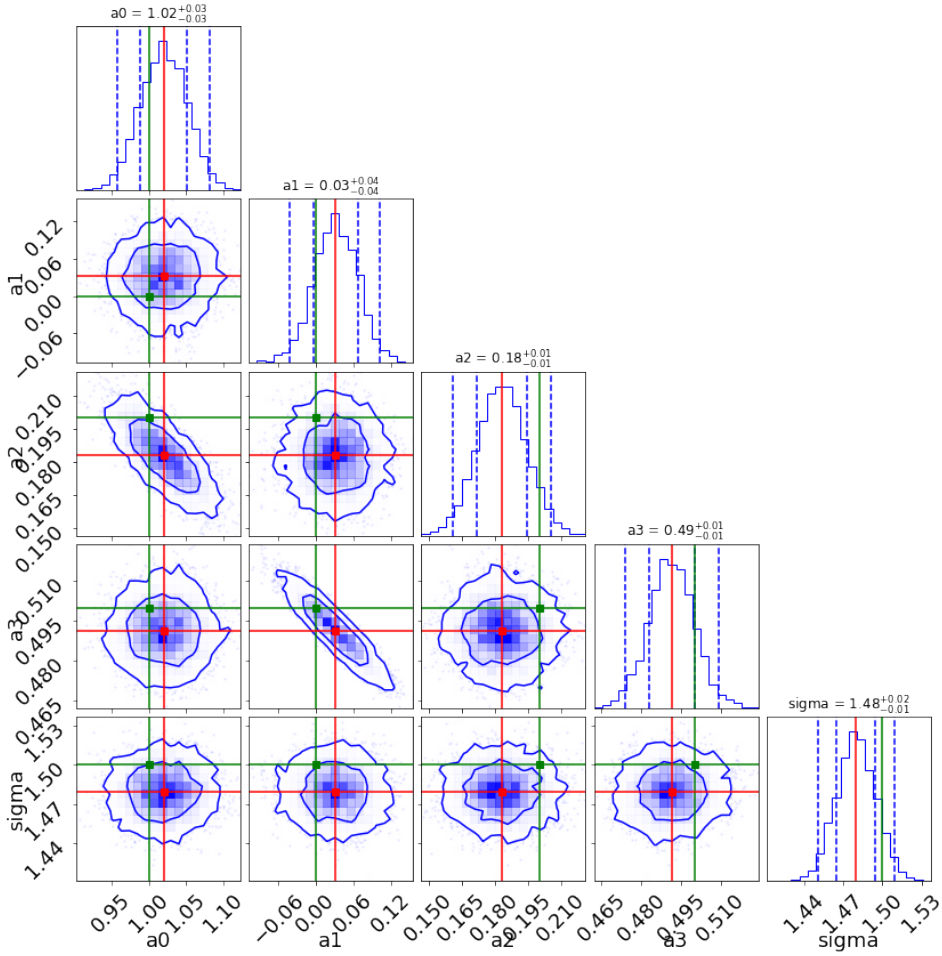

FIGURE 11 – Distributions des *posteriors* avec les contours/lignes verticales à 68% et 95% de niveau de confiance. Les points et lignes rouges sont les moyennes calculées sur les échantillons de la chaîne, tandis que les points verts et lignes correspondantes sont les "vraies" valeurs des paramètres.

6 Conclusion

Dans cette note, j'ai présenté quelques méthodes classiques de tirages Monte Carlo et de production de chaînes de Markov (MCMC), à des fins de calculs d'intégrales et d'estimations de paramètres d'un modèle. L'usage des Monte Carlo basés sur la dynamique hamiltonienne (HMC) qui résout le problème de la marche aléatoire de la méthode Metropolis-Hastings, utilise particulièrement la différentiation automatique pour calculer le gradient de l'énergie potentielle. Une version particulière des HMC, notée NUTS, permet d'optimiser les paramètres de l'intégration numérique des équations du mouvement. Une illustration de ces méthodes est présentée avec la librairie `NumPyro` basée sur `JAX`. Ceci étant les problèmes inhérents à l'utilisation des MCMCs ne sont pas gommés par l'usage de telle ou telle librairie et selon le cas de figure la maitrise du résultat peut être délicate. Si les MCMCs échantillonnent la distribution posterior directement, l'alternative de la méthode variationnelle bayésienne qui paramètrise la distribution posterior obtient quant à elle une approximation locale. Notons pour finir qu'il y a des méthodes qui incorpore des éléments de MCMCs dans l'inférence variationnelle bayésienne.