RAPPORT DE PROJET - IMDS4A

Année 2021-2022

Les méthodes Monte-Carlo par Chaîne de Markov (MCMC)

Projet réalisé par

Simon AMBLARD
Alexandre CANCEL

Projet encadré par

Pierre DRUILHET

POLYTECH
CLERMONT-FERRAND

REMERCIEMENTS

Tout d'abord, nous tenons à remercier tout particulièrement et à témoigner toute notre reconnaissance à notre responsable de projet, M. Pierre DRUILHET, pour son dévouement, son soutien dans la concrétisation de ce projet, pour ses conseils éclairés, sa patience et sa disponibilité.

SOMMAIRE

TABLE DES FIGURES

TABLE DES TABLEAUX

1 Introduction

Dans le cadre de notre quatrième année d'école au sein de Polytech Clermont en Ingénierie Mathématique et Data Science, nous avons eu un projet attribué. Ce projet a donc duré toute l'année scolaire et a été réalisé en binôme. L'intitulé de notre projet est : Simulation de loi par méthode MCMC. Dans la plupart des cas, les lois de probabilité ne sont connues qu'à une constante près. Pour simuler celle-ci, on peut utiliser les méthodes MCMC (Monte-Carlo par Chaîne de Markov) pour résoudre ce problème. Dans un premier temps, nous avons alors étudié ces simulations de loi et créé des algorithmes utilisant les méthodes MCMC. Ensuite, au fil du temps, nous nous sommes rendu compte que ces algorithmes pouvaient servir au décryptage de message codé. C'est pourquoi nous nous sommes intéressés à ce point des méthodes MCMC et nous avons implémenté un algorithme de décryptage à l'aide de l'algorithme de Metropolis-Hasting. Pour la réalisation de ce projet, nous avons utilisé le langage de programmation Python puisque c'est l'un des plus accessibles pour utiliser les statistiques. Enfin, nous avons pu avoir l'aide de notre responsable de projet que nous rencontrions une fois toutes les deux semaines pour poser nos questions et faire en sorte que notre projet soit le plus abouti possible.

2 Simulation d'une loi par méthode MCMC

2.1 Introduction aux méthodes MCMC

La méthode MCMC est une méthode très utilisée pour simuler des lois de probabilité. MCMC veut dire Markov-Chain Monte Carlo. Un algorithme MCMC est un algorithme stochastique qui permet de simuler une distribution à l'aide d'une chaîne de Markov.

On peut regrouper les méthodes de Monte Carlo en deux grandes familles.
Premièrement les méthodes par acceptation-rejet, mais aussi les méthodes par pondération.

Avec la formule de Bayes, on a la loi à postériori définie par :

$$\pi(\theta \mid x) = \frac{f(x \mid \theta)\,\pi(\theta)}{f_X(x)},$$

Mais, en Bayésien, $f_X(x) = \int f(x \mid \theta)\,\pi(\theta)\,d\theta$ est inaccessible dans la plupart des cas, on connaît la loi $\pi(\theta \mid x)$ à postériori seulement à une constante près.

On utilise alors les méthodes MCMC pour simuler les lois à posteriori. Comme son nom l'indique, ces méthodes sont composées de deux termes mathématiques importants : les méthodes de Monte-Carlo et les chaînes de Markov. Il nécessite alors une chaîne de Markov pour mettre en place cette estimation de densité. Une chaîne de Markov est une suite de

variables aléatoires (X_n avec $n \in N$) qui permet de modéliser l'évolution dynamique d'un système aléatoire : X_n représente l'état du système à l'instant *n*.

On appelle méthode de Monte-Carlo toute méthode visant à calculer une valeur numérique et utilisant des procédés aléatoires, c'est-à-dire des techniques probabilistes. Le nom de ces méthodes fait allusion aux jeux de hasard pratiqués à Monte-Carlo.

Les méthodes de Monte-Carlo sont particulièrement utilisées pour calculer des intégrales de dimensions supérieures à 1 (en particulier, pour calculer des surfaces, des volumes, etc.)

Soit *Y* une variable aléatoire de densité *f(y)* et *h* une fonction. On cherche à calculer :

$$E\big(h(Y)\big) = \int h(y)f(y)dy$$

On va maintenant mettre en place une approximation par Monte-Carlo. On simule *N* réalisations indépendantes de $Y \sim (y_1, \ldots, y_n)$. Grâce à la loi forte des grands nombres, on obtient alors l'approximation suivante :

$$\frac{1}{N}\sum_{i=1}^{N} h(y_i) \simeq E\big(h(Y)\big)$$

Or, on ne peut pas utiliser directement cette méthode puisqu'on ne connaît pas la densité de la loi à posteriori qu'à une constante près. C'est là qu'interviennent les chaînes de Markov. On va alors générer $\big(\theta^{(1)}, \ldots, \theta^{(N)}\big)$ selon la loi approchée de $\pi(\theta \mid x)$ pour une fonction h donnée, on a alors :

$$E(h(\theta) \mid x) = \int h(\theta)\pi(\theta \mid x)d\theta \;\approx \frac{1}{N}\sum_{i=1}^{N} h(\theta_i))$$

Il existe deux algorithmes importants mettant ces méthodes en place : l'algorithme de Metropolis-Hasting et l'échantillonnage de Gibbs.

2.2 Algorithme Metropolis-Hasting pour simuler une loi

Initialement développé en 1953 pour le traitement de problèmes de physique par Metropolis, Rosenbluth et Teller, cet algorithme a ensuite été beaucoup utilisé en physique statistique pour simuler des systèmes complexes. Actuellement, dans la littérature statistique, cet algorithme est présenté comme une méthode permettant de fabriquer une chaîne de Markov dont on s'est donné la loi stationnaire π. Sa mise en œuvre présente l'avantage de ne nécessiter la définition de π qu'à une constante près. C'est l'algorithme MCMC le plus général puisqu'il impose moins de conditions sur la densité cible. Pour approximer la loi Π on va créer une chaîne de Markov $(x_1, \ldots, x_n)$ avec les étapes suivantes :

1. Générer $y_{t+1} \sim q(x_t, \cdot)$
2. Calculer la *probabilité d'acceptation*

$$\alpha(x_t, y_{t+1}) = \min\left[\frac{\pi(y_{t+1})q(y_{t+1}, x_t)}{\pi(x_t)q(x_t, y_{t+1})}, 1\right]$$

3. Prendre $x_{t+1} = \begin{cases} y_{t+1} & \text{avec probabilité } \alpha \\ x_t & \text{avec probabilité } 1-\alpha \end{cases}$

Intuitivement, cet algorithme va tenter de se déplacer dans l'espace et ce déplacement va être accepté ou rejeté à chaque itération. Cette acceptation va se faire selon l'état actuel de la chaîne et la distribution Π. Grâce à cet algorithme, on va pouvoir avoir une idée des densités de certaines lois qui sont impossibles à déterminer directement.

Nous avons donc implémenté cet algorithme et avons fait plusieurs tests dessus. Nous avons principalement travaillé avec la loi à posteriori suivante :
$\pi(x) = \exp(-b * |x|) * |x|^{(a-1)}$. Nous avons choisi cette loi puisque celle-ci est évidemment une loi de densité et elle est traçable directement, on va donc pouvoir observer si notre approximation par méthode MCMC est correcte ou non. Nous avons d'abord tracé la chaîne de Markov associée à cette loi. Ensuite, pour vérifier que notre approximation est plutôt correcte, nous avons tracé l'histogramme de celle-ci et la loi de densité de notre loi pour comparer. Nous observons que cette approximation est correcte et que plus notre nombre d'itération est grand plus notre approximation est juste.

Ensuite, nous avons implémenté le corrélogramme associé à la chaîne de Markov. On observe, que dans notre cas, il existe quelques corrélations. Nous avons alors décidé de calculer l'Effective Sample Size qui permet de se rendre compte s'il y a de fortes ou faibles corrélations dans notre chaîne de Markov. Il est calculé avec la formule suivante : $ESS = \frac{N}{1+2\sum_{i=1}^{N} p_i}$
avec *N* le nombre d'itérations, p_i la corrélation d'ordre *i*

En fait, l'ESS permet de calculer la non-redondance et la traduit en termes de taille de l'échantillon.

Nous avons ensuite calculé le nombre de rejets lors de l'algorithme de Metropolis, il est important que le nombre de rejets ne soit pas trop haut.

Ainsi, avec ces mesures, nous avons pu jouer sur nos paramètres afin d'obtenir l'approximation la plus optimisée de notre loi. Nous avons notamment travaillé sur les paramètres comme le pas de notre loi de proposition qui est une loi normale, le thinning, le burn-in et notre valeur initiale.

Le thinning est un entier qui permet de réduire l'autocorrélation. Étant donné que l'autocorrélation a tendance à diminuer à mesure que le décalage augmente, le thinning réduira l'autocorrélation finale dans l'échantillon tout en réduisant également le nombre total d'échantillons enregistrés. En raison de l'autocorrélation, la réduction du nombre d'échantillons

efficaces sera souvent inférieure au nombre d'échantillons retirés lors de l'éclaircissement. Le thinning diminue toujours la taille effective de l'échantillon.

Le burn-in est un terme qui décrit la pratique consistant à jeter certaines itérations au début d'une exécution MCMC. La notion de burn-in dit que vous commencez quelque part, disons à x, puis vous exécutez la chaîne de Markov pendant *n* étapes, à partir de laquelle vous jetez toutes les données (pas de sortie). C'est la période de rodage. Après le rodage, vous exécutez normalement, en utilisant chaque itération dans vos calculs MCMC. Ce paramètre permet d'éviter la période où l'algorithme de Metropolis est loin de l'équilibre.

Il peut être aussi intéressant de réaliser une première chaîne de Markov où l'on calcule l'espérance et la variance de notre chaîne de Markov. Avec ces résultats d'observation obtenus, on peut modifier notre loi de proposition en conséquence pour avoir de meilleurs résultats. C'est-à-dire sur notre deuxième chaîne, on va avoir x_0 égal à notre première espérance et la variance de la loi normale égale à la variance calculée précédemment. En fait, on va modifier le pas et la valeur initiale de notre chaîne de Markov.

Notre objectif, ici, est de trouver le juste milieu pour avoir le moins d'autocorrélation et le moins rejets possible.

Par l'algorithme que nous avons implémenté nous allons voir que ce change ces paramètres à nos résultats.
Voici nos résultats pour notre loi gamma de paramètres a = 4 et b = 7. Pour notre premier test de notre méthode MCMC, nous n'avons utilisé aucun des méthodes décrites précédemment. Les paramètres de notre méthode MCMC sont les suivants :
Nb_itération = 10000
burn-in = 0
thinning = 1
variance = 5
espérance = 0.1

On obtient alors la chaîne de Markov suivante :

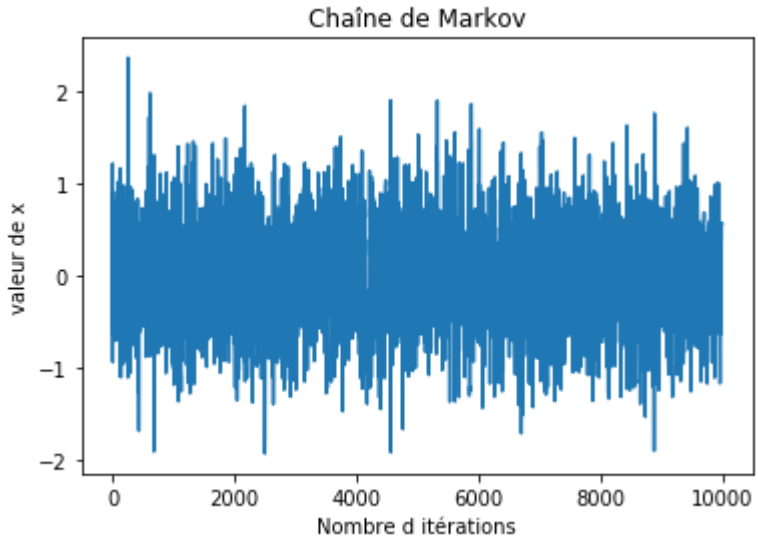

Figure 1 : Chaîne de Markov

Grâce à des calculs simples, on obtient les valeurs suivantes :

Var = 0.3983862849220144
E = -0.015673486502632398
rejet = 7296

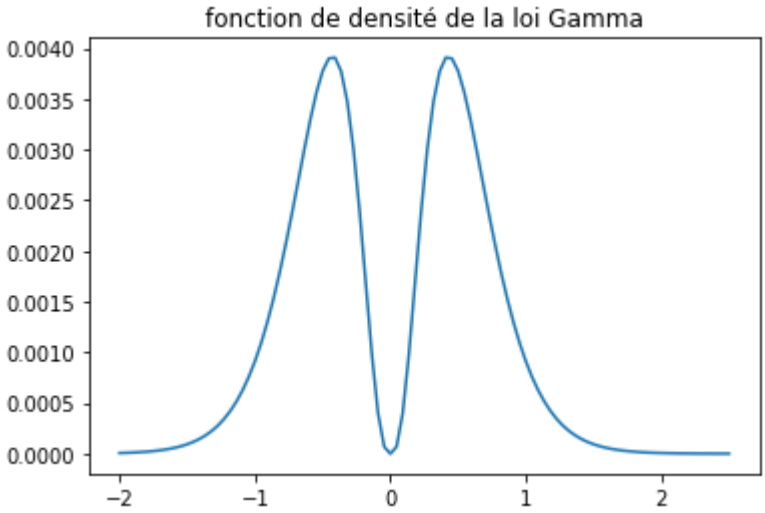

Figure 2 : Fonction densité de la loi Gamma

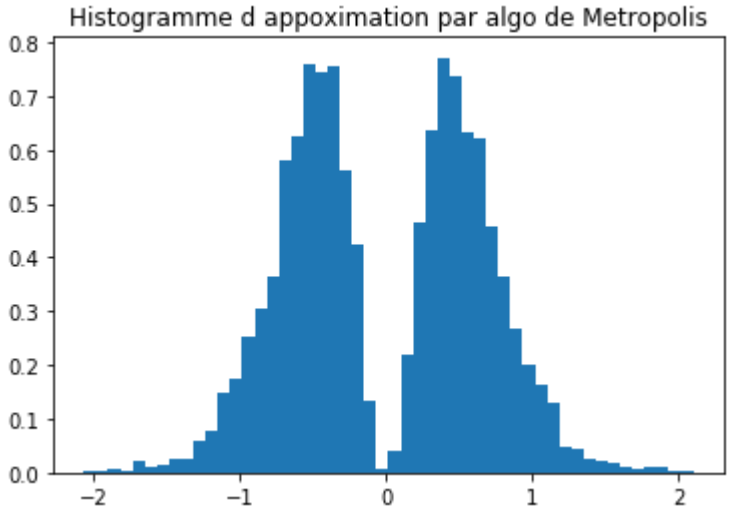

Figure 3 : Histogramme de l'approximation de la loi de densité

On remarque que notre histogramme se rapproche bien de notre fonction de densité.

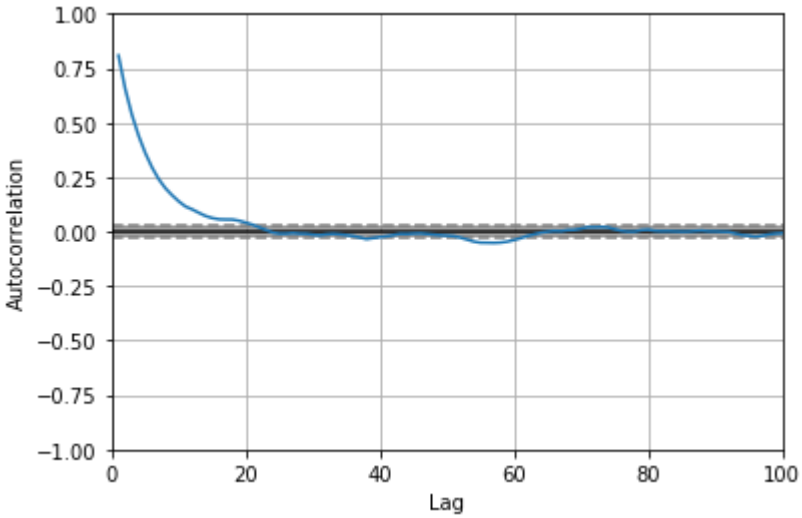

Figure 4 : Corrélogramme

Avant la 20ème itération les x_i sont fortement corrélé puis, après il n'y a plus de corrélation.

ESS = 2259.003641554309

Dans un deuxième temps, on va utiliser les techniques énoncées précédemment afin d'avoir de meilleurs résultats.

Dans notre fonction de l'algorithme de Metropolis-Hasting, on va désormais prendre en entrée un thinning de 10, un burn-in de 500 et on va exécuter notre fonction une première fois avec un pas et une valeur initiale aléatoire et ensuite, grâce aux résultats de celle-ci, on va adapter notre pas et notre valeur initiale. On a trouvé, dans notre première chaîne, une moyenne de -0.02 et une variance de 0.4. On va alors utiliser comme pas 0.4 et comme valeur initiale -0.02. Voici les résultats obtenus après la modification de ces paramètres.

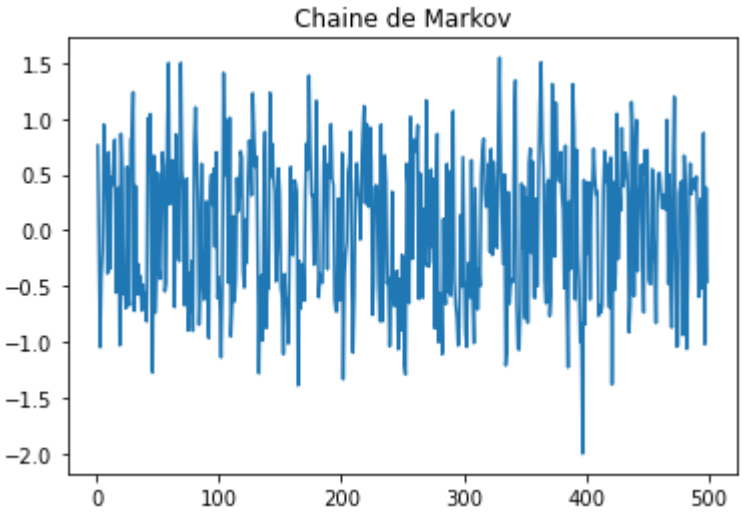

Figure 5 : Chaîne de Markov avec paramètres

On peut remarquer qu'il y a plus de plateau sur cette chaîne de Markov :

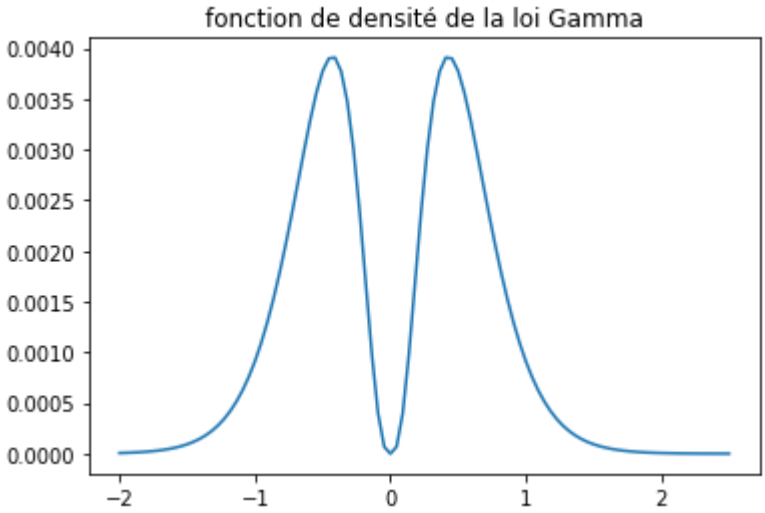

Figure 6 : Fonction de densité de la loi à posteriori

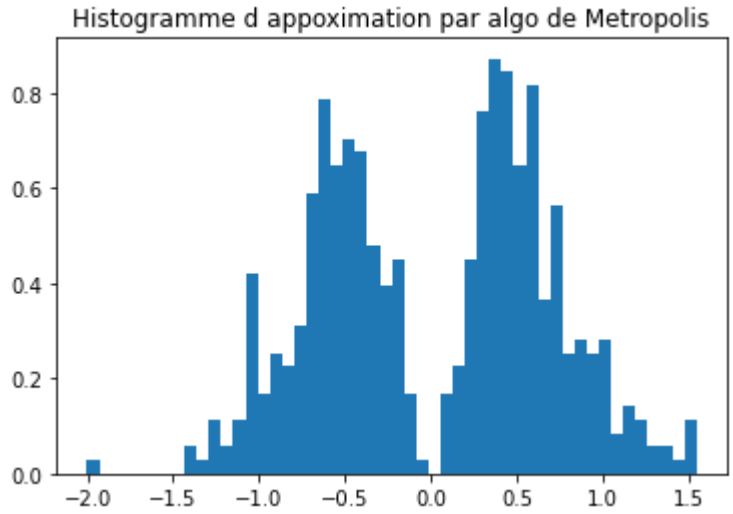

Figure 7 : Histogramme d'approximation par algo de Metropolis

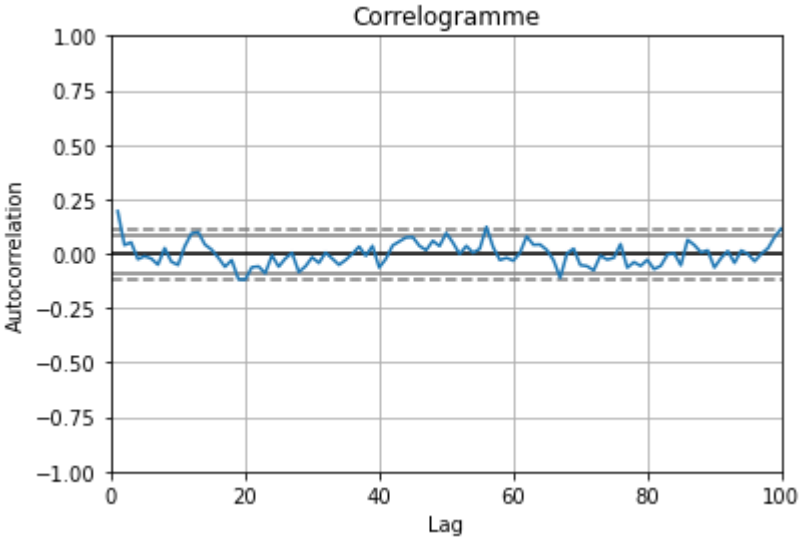

Figure 8 : Corrélogramme

rejet = 4454
ESS = 4885.467907443195

On perd des itérations de l'échantillon, mais on a un meilleur corrélogramme puisque dès la première itération, la corrélation est très faible et le résultat d'ESS est bien meilleur. Ensuite, grâce à l'utilisation d'une seconde chaîne de Markov, on arrive à faire baisser considérablement le nombre de rejets. En revanche, on remarque que l'histogramme est moins précis dû au fait que l'on perd de l'information avec le thinning et le burn-in.

2.3 Echantillonnage de Gibbs

Le deuxième algorithme très utilisé dans le domaine des méthodes MCMC est l'échantillonnage de Gibbs. Soit un problème multidimensionnel dans lequel on cherche à simuler la distribution d'un paramètre $\Theta = (\theta_1, \ldots, \theta_n)$ étant donné des observations.
On va supposer qu'on ne sait pas simuler la loi $P(\Theta|X)$, on note Θ_{-i} le vecteur :

$$\Theta = (\theta_1, \ldots, \theta_{i-1}, \theta_{i+1}, \ldots, \theta_n)$$

On suppose alors qu'on est capable de simuler, pour tout i, la loi $P(\theta_i|X, \Theta_{-1})$.
On considère le vecteur $\Theta^{(k)}$ obtenu à la $k^{ème}$ itération. On en déduit alors le vecteur $\Theta^{(k+1)}$ qui suit.
-> on simule $\theta_1^{(k+1)} \sim P\left(\theta_1 \middle| X, \theta_2 = \theta_2^{(k)}, \ldots, \theta_n = \theta_n^{(k)}\right)$
-> on simule $\theta_2^{(k+1)} \sim P\left(\theta_2 \middle| X, \theta_1 = \theta_1^{(k+1)}, \theta_3 = \theta_3^{(k)}, \ldots, \theta_n = \theta_n^{(k)}\right)$
-> ….
-> on simule $\theta_n^{(k+1)} \sim P\left(\theta_n \middle| X, \theta_1 = \theta_1^{(k+1)}, \ldots, \theta_{n-1} = \theta_{n-1}^{(k)}\right)$

Les itérations successives de cet algorithme génèrent successivement les états d'une chaîne de Markov.

L'échantillonneur de Gibbs peut être vu comme un analogue de l'algorithme de Metropolis-Hastings. La proposition concerne à tour de rôle chacune des dimensions de Θ. La probabilité d'acceptation est systématiquement de 1. Il faut cependant être capable de déterminer (ou simuler suivant) les lois $P(\theta_i|X, \Theta_{-1})$.

3 Cryptographie grâce aux méthodes MCMC

3.1 Introduction à la cryptographie

La cryptographie est l'ensemble des techniques qui permettent de chiffrer et de déchiffrer un message, dont le contenu ne doit être connu que de son expéditeur et de son destinataire. Son déchiffrement par un tiers n'est pourtant pas impossible. Il nécessite la connaissance d'un certain nombre de données fondamentales. Au cours des siècles, de nombreux systèmes cryptographiques ont été mis au point, de plus en plus perfectionnés, de plus en plus astucieux. De grands chercheurs associés à la naissance de l'informatique étaient aussi des spécialistes de

cryptographie : Charles Babbage (1894) et Alan Turing (il s'est illustré pendant la Seconde Guerre Mondiale, en décodant les messages que la marine allemande chiffrait avec la machine Enigma, dont un exemplaire a été envoyé en Angleterre par des résistants).

Une des applications dans la vraie vie des méthodes MCMC est le décryptage de messages codés. Cette méthode de décryptage a été inventée à la suite de messages cryptés d'un tueur en série nommé le Zodiaque. Ses messages étaient indécryptables. En 2001, un psychiatre a trouvé un message crypté similaire à celui du Zodiaque dans la prison où il travaillait. Il décida alors de le confier au mathématicien Persi Diaconis qui travaillait en tant que professeur de statistiques à l'Université de Stanford. Ce professeur a alors élaboré, avec l'aide de certains étudiants, une technique de décryptage très efficace mettant en œuvre les méthodes MCMC.

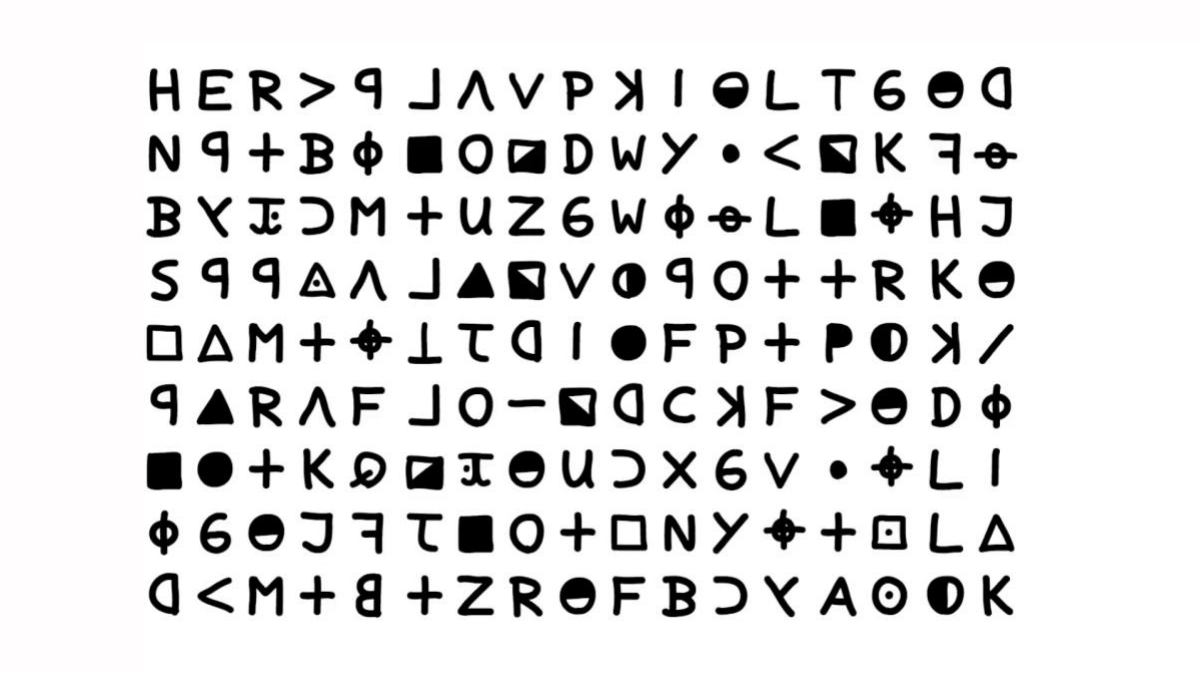

Figure 9 : Message codé du Zodiaque

3.2 Une solution avec les méthodes MCMC

On imagine que l'ordre des lettres de l'alphabet a été mélangé de manière totalement aléatoire. Par exemple voici les nouvelles correspondances des lettres :

Tableau 1 : Alphabet mélangé aléatoirement

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
K	I	J	H	B	P	O	W	U	R	D	E	X	M	L	G	N	Y	T	C	V	S	Q	Z	F	A

Ici, le 'A' correspond à un 'K' et le 'B' correspond à un 'I'. Les lettres de l'alphabet crypté peuvent évidemment être des symboles.

On sait que les lettres de l'alphabet n'ont pas toutes la même probabilité d'intervenir dans un mot. Il est simple d'entrer un fichier en .txt dans un algorithme afin de calculer la proportion

de chaque lettre du fichier. Avec un fichier long, on obtiendra les proportions de chaque lettre. Ensuite, on sait qu'il y a aussi des enchaînements de lettres plus probables que d'autres dans la langue française. Par exemple, un 'Q' est le plus souvent suivi d'un 'U'.

Avec ces deux statistiques, on va pouvoir obtenir un score de plausibilité. Il s'agit d'un réel qui permet de calculer la plausibilité qu'une phrase soit bien traduite.

Afin d'obtenir le meilleur score de plausibilité, on va utiliser les méthodes MCMC et notamment l'algorithme de Metropolis afin que le score converge vers le meilleur résultat possible.

À l'aide de nos recherches sur le domaine et de nos nouvelles connaissances en méthode MCMC, nous avons donc décidé d'implémenter cet algorithme de décryptage.

On sait qu'il faut utiliser un livre qui sera considéré comme une chaîne de Markov : il y a des probabilités de transition entre chaque lettre. Or, pour obtenir un déchiffrage, on va devoir enlever tous les caractères parasites du livre. Ici, nous avons choisi de passer toutes les lettres en majuscule et de passer les lettres avec accent en majuscule sans accent. Par exemple, on va passer un 'à' en 'A'. De ce fait, on aura passé notre livre en chaîne de caractères avec uniquement les lettres de l'alphabet en majuscule et les espaces.

Ensuite, nous avons calculé la fréquence de chaque lettre dans la langue française à l'aide d'un livre plutôt long et moderne puisque nous supposons que nous devons déchiffrer un message crypté définit récemment. S'il s'agit de déchiffrer un code des années 1600 il serait préférable de choisir un livre de cette époque-là. Pour ce faire, nous avons d'abord récupéré le livre "L'appel de l'ange" de Guillaume Musso en .pdf et nous l'avons converti en .txt. Ensuite, ce fichier sera récupéré par notre programme et nous passerons celui-ci en chaîne de caractères avec uniquement les lettres de l'alphabet en majuscule et les espaces grâce aux fonctions implémentées précédemment.

Tableau 2 : Nombre de fois qu'une lettre intervient dans le livre

Code Ascii	Lettre	Fréquence
32	' '	93046
65	A	39397
66	B	3727
67	C	13474
68	D	15430
69	E	68057
70	F	4579
71	G	4136
72	H	4264

Code Ascii	Lettre	Fréquence
73	I	30683
74	J	2504
75	K	432
76	L	25612
77	M	11248
78	N	28848
79	O	21621
80	P	11732
81	Q	4343
82	R	27248
83	S	30281
84	T	29620
5	U	24875
86	V	6948
87	W	258
88	X	1486
89	Y	1406
90	Z	509

Ensuite, nous devons compter la fréquence qu'une lettre suive une autre. Par exemple, combien de fois on obtient un 'U' après un 'Q'. Tous ces résultats seront donc stockés dans une matrice avec en abscisse la seconde lettre et en ordonnée la première. Pour mieux visionner cette matrice, on va retourner une image qui va décrire la matrice. Plus le carré est clair, plus une lettre tombe après une autre. Par exemple, lorsqu'on a un « U » en abscisse et un « Q » en ordonnée, on obtient logiquement un carré très clair.

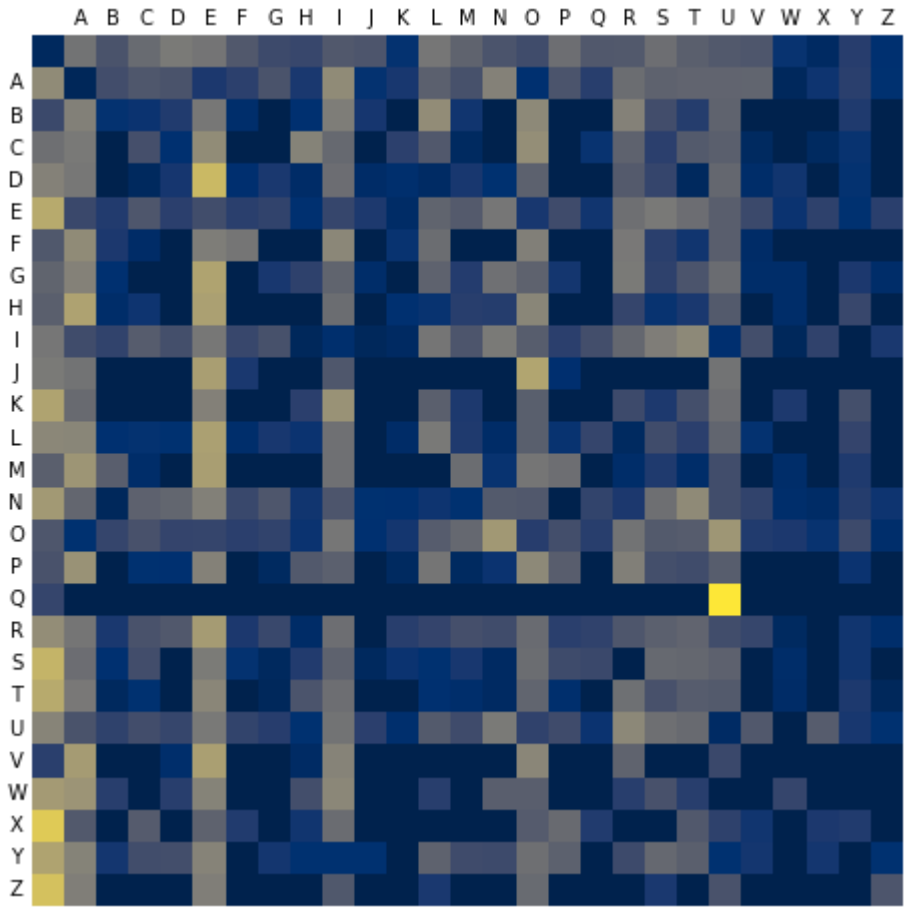

Figure 10 : Table des probabilités de chaque bigramme

Par exemple, sur la ligne 17 de notre matrice qui correspond à la lettre Q nous obtenons la première ligne du tableau ci-dessous.

Tableau 3 : Fréquence des lettres suivant un "Q"

		A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Nb de lettre suivant un Q	33	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4310	0	0	0	0	0

À partir de cette matrice, nous allons ainsi calculer les probabilités d'obtenir une lettre après une autre.

Nous obtiendrons donc la première ligne du tableau ci-dessous pour la lettre Q.

Tableau 4 : Probabilité d'obtenir une lettre après un "Q"

		A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Probabilité d'obtenir une certaine lettre suivant un Q	0.007	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.993	0	0	0	0	0

Après avoir obtenu cette matrice de probabilités, nous allons prendre le logarithme de la probabilité multipliée afin de ne pas avoir de nombre trop petit auquel on va ajouter un epsilon très petit choisi au préalable afin de ne pas avoir de 0 lorsque nous allons faire notre logarithme. Ensuite, nous sommons toutes les valeurs de la matrice obtenue après le passage au logarithme en fonction des suites de lettres présentes dans notre chaîne de caractères puis nous divisons cette somme par le nombre total de caractères présent au sein de notre phrase.
$Score1(x) = \frac{1}{N}\sum_i \log(p_i + \in)$ avec p_i élément du tableau et N est le nombre de caractères et $\in$ une constante proche de 0.
Le score de plausibilité varie entre -2 pour les meilleures traductions à -7 pour les pires. Ce score va permettre de savoir si la suite de caractères est possible ou non dans la langue française.

3.3 L'algorithme de Metropolis-Hasting appliqué au score de plausibilité

```
while (k < N) and (best_score < scoreParfait):

    # Construire un mélange provisoire
    i = np.random.randint(1,27)
    j = np.random.randint(1,27)
    tt_code = permutationCode(code, i, j)

    tt_trad = changerMotAvecNouvCode(text_cripter, tt_code)
    tt_score = scorePlausibilite(tt_trad)

    # Tester si le déplacement doit être accepté
    x = np.random.rand()
    p = np.exp(ALPHA*(tt_score - score) * len(text_cripter))

    if(x < p):
        code = tt_code.copy()
        trad = tt_trad
        score = tt_score

        if(score > best_score):
            best_code = code.copy()
            best_score = score
            best_trad = trad
            print(best_trad + "    [k=" + str(k) + " L={0:.2f}]".format(best_score))
    else :
        rejet = rejet + 1
```

Figure 11 : Algorithme de Metropolis-Hasting appliqué à la cryptographie

L'algorithme de Metropolis-Hasting permet ici de trouver le minimum global de la fonction du score de plausibilité. A chaque itération on échange deux lettres aléatoirement et calcule le nouveau score de plausibilité. En fait, lorsque le score de l'itération *i* est meilleur que celui de l'itération *i-1* on va l'accepter et si le score est moins bon on va l'accepter avec une certaine probabilité qui varie selon la différence entre les deux scores. Plus la différence entre les deux scores est grande, moins il y a de chance que le déplacement soit accepté.

Ici, nous avons donc un algorithme de Metropolis-Hasting appliqué au score de plausibilité. À chaque nouveau mélange de lettres, on calcule le nouveau score de plausibilité puis on teste si le déplacement est accepté. Si oui, on regarde s'il est meilleur que le meilleur score en mémoire pour éviter de surcharger l'affichage et de ne retenir que le meilleur score qui a été vu pour le moment. Le déplacement est accepté si $p_i = \exp(\alpha(score_i - oldscore) \times l)$ (Avec α constante, l la longueur de la chaîne et $score_i$ le nouveau score et oldscore le score en mémoire) est supérieur à un nombre aléatoire entre 0 et 1.

On fait *N* itérations sauf si le meilleur score est supérieur à -2,05 qui correspond à un score considéré comme quasiment parfait.

3.4 Résultat et analyse du décryptage

Pour tester notre programme, nous avons mis en place un système de chiffrement aléatoire d'une phrase. Par exemple, ici, nous avons entré la phrase suivante : "Cette année, nous avons un projet important pour valider notre quatrième année d'école d'ingénieur. Nous avons un rapport et un oral pour noter ce projet. Nous avons eu un tuteur de projet spécialiste en statistique pour nous aider. Nous avons choisi d'implémenter un code permettant de déchiffrer un message codé. Nous sommes fiers de comprendre la plupart des messages". Grâce à notre système de chiffrement, les lettres sont changées aléatoirement. À chaque fois que le code compile, le chiffrement change. Par exemple, nous avons ce nouvel alphabet :

Tableau 5 : Cryptage du message

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
M	W	E	F	V	B	Z	O	R	K	G	P	C	S	N	I	L	H	U	Y	X	Q	D	T	J	A

Et notre phrase devient la suivante :
"EVYYV MSSVV SNXU MQNSU XS IHNKVY RCINHYMSY INXH QMPRFVH SNYHV LXMYHRVCV MSSVV F VENPV F RSZVSRVXH SNXU MQNSU XS HMIINHY VY XS NHMP INXH SNYVH EV IHNKVY SNXU MQNSU VX XS YXYVXH FV IHNKVY UIVERMPRUYV VS UYMYRUYRLXV INXH SNXU MRFVH SNXU MQNSU EONRUR F RCIPVCVSYVH XS ENFV IVHCVYYMSY FV FVEORBBHVH XS CVUUMZV ENFVH SNXU UNCCVU BRVHV FV ENCIHVSFHV PM IPXIMHY FVU CVUUMZVU". On a calculé le score de plausibilité de ce texte et on obtient un score de -6.28.

Ensuite, nous avons testé notre programme pour ce chiffrement. Comme notre texte est plutôt long, il converge plutôt rapidement vers la solution voulue. Juste en remplaçant les lettres par leurs fréquences dans le livre précédent, on obtient la traduction suivante :

“PENNE LAAEE AIRT LVIAT RA USIHEN OCUISNLAN UIRS VLMODES AINSE GRLNSOECE LAAEE D EPIME D OAQEAOERS AIRT LVIAT RA SLUUISN EN RA ISLM UIRS AINES PE USIHEN AIRT LVIAT ER RA NRNERS DE USIHEN TUEPOLMOTNE EA TNLNOTNOGRE UIRS AIRT LODES AIRT LVIAT PBIOTO D OCUMECEANES RA PIDE UESCENNLAN DE DEPBOFFSES RA CETTLQE PIDES AIRT TICCET FOESE DE PICUSEADSE ML UMRULSN DET CETTLQET”.
Ici, on a un score de -3.21 donc on améliore relativement ce résultat.

Puis, à l’aide de notre algorithme de Metropolis-Hasting, on obtient la traduction suivante au bout de seulement 2801 itérations :
“CETTE ANNEE NOUS AVONS UN PROYET IMPORTANT POUR VALIDER NOTRE QUATRIEME ANNEE D ECOLE D INGENIEUR NOUS AVONS UN RAPPORT ET UN ORAL POUR NOTER CE PROYET NOUS AVONS EU UN TUTEUR DE PROYET SPECIALISTE EN STATISTIQUE POUR NOUS AIDER NOUS AVONS CHOISI D IMPLEMENTER UN CODE PERMETTANT DE DECHIFFRER UN MESSAGE CODER NOUS SOMMES FIERE DE COMPRENDRE LA PLUPART DES MESSAGES”
Le score de plausibilité est cette fois de -2.26. On remarque, intuitivement qu’il reste une erreur, le ‘Y’ est en fait un ‘J’.

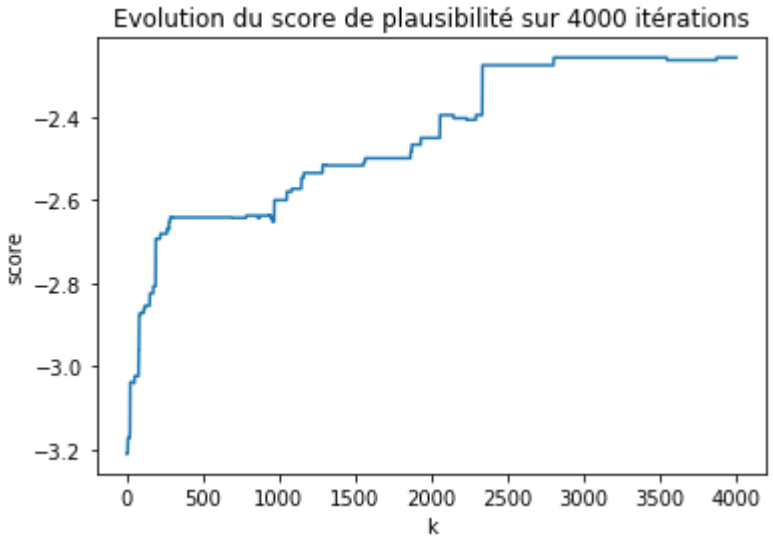

Figure 12 : Evolution du score de plausibilité sur 4000 itérations

On a un nombre de rejets de 45733 sur 50000 itérations.

On peut voir sur ce graphique qu’au bout de 2500 itérations l’algorithme converge vers une solution d’un score d’environ -2,3.

Ensuite, pour obtenir un résultat encore meilleur, on a utilisé un deuxième algorithme de Metropolis-Hasting, mais cette fois avec un autre score de plausibilité et en entrée de l’algorithme le code et la traduction du précédent. Cette fois-ci, on a récupéré le dictionnaire de la langue française et grâce à celui-ci, on va calculer le nombre de mots corrects dans la phrase traduite et on le divise par le nombre de mots de notre phrase. Notre nouveau score sera alors : $Score2(x) = \gamma \times p + Score1(x)$ Avec γ constante, ici γ = 4, p proportion de mots corrects et *Score1*(x) le premier score de plausibilité. De même que précédemment, on utilise l’algorithme de Metropolis-Hasting afin d’optimiser ce deuxième score dans le but d’avoir un

meilleur résultat. Dans ce deuxième algorithme on obtient à la fin Mots OK 60/63 score=0.9407894736842105

4 Conclusion

Lors de notre projet, nous avons découvert les méthodes MCMC et leur utilisation. Nous avons alors vu qu'elles servaient essentiellement à simuler des lois de probabilité. Nous avons alors implémenté l'algorithme de MCMC le plus connu et utilisé l'algorithme de Metropolis-Hasting afin de simuler des lois et d'utiliser différents paramètres afin d'avoir les meilleurs résultats possibles. Enfin, nous avons vu que cet algorithme pouvait aussi servir au décryptage. L'algorithme est cette fois utilisé à des fins d'optimisation et non de simulation. Il est important de différentier ces deux applications de l'algorithme de Metropolis-Hasting. Nous avons alors implémenté une deuxième fois cet algorithme, mais cette fois-ci pour déchiffrer des messages codés. Avec notre algorithme, nous pouvons alors décrypter la plupart des messages codés.

Si l'algorithme de Metropolis-Hasting était plutôt simple à implémenter afin de simuler des lois de probabilité, il a été plus difficile pour nous de mettre en place celui de décryptage. Cet algorithme possède de nombreuses fonctions afin de pouvoir traiter un livre et de mettre en place le score de plausibilité. Ces fonctions nous ont donc pris du temps. Voir l'algorithme de Metropolis-Hasting à des fins d'optimisation était plutôt déroutant au début, il nous a alors fallu du temps à comprendre comment cela fonctionnait.

5 Annexe

5.1 Algorithme de Metropolis-Hasting pour simuler une loi

```
def Metropolis(x,thinning,burning,N,var):
    rejet = 0 #On initialise le rejet à 0 pour pouvoir compter combien de fois on a rejeter l'avancement
    for i in range(1,N):
        y = np.sqrt(var)*float(np.random.normal(size=1)) + x[i-1] #On
        p = np.minimum(posteriori(y) / posteriori(x[i-1]),1)
        test = np.random.binomial(1,p) #On test si on fais varier
        if (test==1): #Si le resultat du test est 1 alors on
            x.append(y)
        else: # sinon ... et par conséquent on incrémente le rejet de 1
            x.append(x[i-1]) #
            rejet += 1
    print("Var = ", np.var(x)) #On affiche à l'utilisateur la variance de notre vecteur x
    print("E = ",np.mean(x))
    print("rejet = ", rejet) #On affiche à l'utilisateur la valeur de rejet
    y = []
    for i in range(len(x)):
        if i%thinning == 0:
            y.append(x[i])
    x1 = y[burnin:]
    return x1, rejet
```

5.2 Création et affichage de la matrice des probabilités de bigramme

```
def calculBigramme(text, outfile, imagefile = None):

    # Nombre de caractères ASCII pour vérifier que tout est ok
    count = np.zeros(10000)
    for c in text:
        count[ord(c)] += 1

    for i in range(10000):
        if count[i] > 10:
            print(str(i) + " " + chr(i) + " " + str(count[i]))


    # Nous sommes maintenant prêts à compter les bigrammes
    bigrams = np.zeros((27,27),dtype='int32')
    i = 0
    for c in text:
        if count[ord(c)] > 10:
            j = 0 if c == " " else ord(c) - 64
            bigrams[i,j] += 1
            i = j

    bigrams.tofile(outfile)
```

```
 if imagefile != None:
     # graphique
     p2D=bigrams.astype('float')/np.tile(sum(bigrams.T),(27,1)).T
     p2D[np.isnan(p2D)] = 0

     alpha = 0.33
     p2Da = p2D**alpha
     plt.figure(figsize=(8,8))
     plt.imshow(p2Da,interpolation='nearest', cmap = 'cividis')
     plt.axis('off')

     for ip, i in enumerate([32]+list(range(65,91))):
         plt.text(-1,ip,chr(i),horizontalalignment='center',
                               verticalalignment='center')
         plt.text(ip,-1,chr(i),horizontalalignment='center',
                               verticalalignment='center')
     plt.savefig(imagefile)

     return p2Da
```

```
# creation bigramme
Comptage_bigram = True

livre = "Musso.txt"
bigramme_fichier = "bigrams.dat"


if Comptage_bigram:
    #on met tout dans une chaine de caractere
    text = OuvertureEtMiseEnString(livre)
    #on la transforme sans accents...
    text = TransformeLaChaine(text)
    #on calcule les probabilité qu'une lettre vienne après une autre
    calculBigramme(text, bigramme_fichier, "bigrams.png")
```

5.3 Calcul du score de plausibilité

```
def scorePlausibilite(s):
    #On creer une matrice de taille 27x27 avec les bigrammes de chaque lettres
    bigramme = np.fromfile(bigramme_fichier,dtype="int32").reshape(27,27)
    #dans un nouveau tableau p de taille 27x27 on calcul
    #les probabilités de chaque bigramme pour chaque lettres
    p=bigramme.astype('float')/np.tile(sum(bigramme.T),(27,1)).T
    #On remplace par 0 les valeurs de p = NaN
    p[np.isnan(p)] = 0
    #On initialise EPSILON très petit pour ne pas avoir de 0 dans notre matrice p
    EPSILON = 1e-6

    logp = np.log(p + EPSILON)
    #logp = np.log(p * p)
    #On initialise notre variable qui nous permet de faire
    #la somme des lopg[i,j] à 0 et c1 le premier caractère de
    #la chaine
    res = 0
    c1 = s[0]
#On parcourt tous les caractère de la chaine un à un à partir du second
for c2 in s[1:]:
    #On donne à i et j la valeur des identifiant respectif de leur caractère
    i = char_a_id(c1)
    j = char_a_id(c2)
    # res somme toutes les valeurs des logp[i,j] de tous
    #les caractères se suivant dans la chaine
    res += logp[i,j]
    #c1 prend la valeur c2 pour que c1 et c2 soient
    #toujours les caractères dans la chaine
    c1 = c2
#Pour finir, on retourne notre somme des logp sur la
#longueur de la chaine pour obtenir notre socre de plausibilité
return res/len(s)
```

5.4 Score avec nombre de mots corrects

```
for k in range(NITER2):

    # Build a tentative move and compute score
    i = np.random.randint(1,27)
    j = np.random.randint(1,27)
    tt_code = permutationCode(code, i, j)
    tt_trad = changerMotAvecNouvCode(text_cripter, tt_code)
    tt_word_score = scoreMotCorrect(tt_trad, dico)
    tt_score = scorePlausibilite(tt_trad)
    tt_score2 = GAMMA * tt_word_score + tt_score


    # Test whether move should be accepted
    x = np.random.rand()
    p = np.exp((tt_score-score)/temperature)
    temperature = temperature * rho

    if(x < p):
        code = tt_code.copy()
        trad = tt_trad
        score2 = tt_score2

        if(score2 > best_score2):
            best_code = code
            best_score2 = score2
            best_trad = trad
            print(tt_trad + " W={0:.2f}".format(tt_word_score,
```

6 Bibliographie

6.1 Webographie

Méthode de Monte-Carlo par chaînes de Markov
Disponible sur : https://fr.wikipedia.org/wiki/M%C3%A9thode_de_Monte-Carlo_par_cha%C3%AEnes_de_Markov

Autocorrelation of Time Series Data in Python
Disponible sur : https://www.alpharithms.com/autocorrelation-time-series-python-432909/

Markov Chain Monte Carlo, ou comment déchiffrer n'importe quel code secret – Science étonnante
Disponible sur : https://scienceetonnante.com/2021/04/23/mcmc-code/

Applications de MCMC pour la cryptographie et l'optimisation
Disponible sur : https://ichi.pro/fr/applications-de-mcmc-pour-la-cryptographie-et-l-optimisation-34742563729714

6.2 Bibliographie

Méthodes de Monte Carlo et Chaînes de Markov pour la simulation
Disponible sur : https://www.math.univ-toulouse.fr/~gfort/Preprints/HDR.pdf

L'Algorithme Metropolis-Hastings Projet de recherche CRSNG
Disponible sur : https://dms.umontreal.ca/~bedard/BergeronL_rapport_final.pdf

Diaconis, P. (2009). The markov chain monte carlo revolution. Bulletin of the American Mathematical Society.
Disponible sur : https://math.uchicago.edu/~shmuel/Network-course-readings/MCMCRev.pdf

Échantillonnage de Gibbs et autres applications économétriques des chaînes markoviennes.
Disponible sur : https://www.erudit.org/fr/revues/ae/1996-v72-n1-ae2748/602194ar.pdf