Algorithme

L'algorithme ci-dessous représente les concepts discutés dans le texte sous forme d'instructions algorithmiques, bien que cela ne soit pas dans un langage de programmation spécifique comme Python.

1. \*\*Génération des demi-périodes de l'oscillateur rapide :\*\*

- Entrées : Moyenne (`mean`), Variance (`variance`), Nombre d'échantillons (`num\_samples`)

- Sortie : Liste des demi-périodes

\*\*Algorithme :\*\*

Générer une séquence de demi-périodes d'un signal oscillant rapide :

Pour chaque k de 1 à num\_samples :

Xk <- échantillon de la distribution normale avec moyenne mean et écart-type sqrt(variance)

Fin Pour

Retourner la liste des demi-périodes X1, X2, ..., Xnum\_samples

2. \*\*Calcul du facteur de qualité :\*\*

- Entrées : Ratio de gigue (`jitter\_ratio`), Ratio de fréquence (`frequency\_ratio`)

- Sortie : Facteur de qualité

\*\*Algorithme :\*\*

```

Calculer le facteur de qualité d'un TRNG basé sur les ratios de gigue et de fréquence :

Facteur de qualité <- jitter\_ratio \* frequency\_ratio

Retourner le facteur de qualité

```

3. \*\*Calcul du ratio de divisibilité :\*\*

- Entrées : Intervalle d'échantillonnage fixe (`sampling\_interval`), Demi-période (`half\_period`)

- Sortie : Ratio de divisibilité

\*\*Algorithme :\*\*

```

Calculer le ratio de divisibilité de la demi-période à l'intervalle d'échantillonnage :

Ratio de divisibilité <- (sampling\_interval % half\_period) / half\_period

Retourner le ratio de divisibilité

```

4. \*\*Exemple d'utilisation :\*\*

- Entrées : Moyenne de la demi-période (`mean\_half\_period`), Variance de la demi-période (`variance\_half\_period`), Nombre de demi-périodes (`num\_half\_periods`), Intervalle d'échantillonnage fixe (`sampling\_interval`)

- Sorties : Ratio de gigue, Ratio de fréquence, Facteur de qualité, Ratio de divisibilité

\*\*Algorithme :\*\*

```

Exemple d'utilisation :

Générer les demi-périodes avec la moyenne mean\_half\_period, la variance variance\_half\_period et le nombre num\_half\_periods

Calculer le ratio de gigue et le ratio de fréquence

Calculer le facteur de qualité en utilisant le ratio de gigue et le ratio de fréquence

Calculer le ratio de divisibilité avec l'intervalle d'échantillonnage fixe sampling\_interval et la demi-période moyenne mean\_half\_period

```

Notez que cet algorithme n'est pas spécifique à un langage de programmation particulier et peut être implémenté dans le langage de votre choix en utilisant les structures de contrôle et les opérations spécifiques à ce langage.

Deux

L'algorithme ci-dessous représente les concepts discutés dans le texte sous forme d'instructions algorithmiques, bien que cela ne soit pas dans un langage de programmation spécifique comme Python.

1. \*\*Génération des demi-périodes de l'oscillateur rapide :\*\*

- Entrées : Moyenne (`mean`), Variance (`variance`), Nombre d'échantillons (`num\_samples`)

- Sortie : Liste des demi-périodes

\*\*Algorithme :\*\*

```

Générer une séquence de demi-périodes d'un signal oscillant rapide :

Pour chaque k de 1 à num\_samples :

Xk <- échantillon de la distribution normale avec moyenne mean et écart-type sqrt(variance)

Fin Pour

Retourner la liste des demi-périodes X1, X2, ..., Xnum\_samples

```

2. \*\*Calcul du facteur de qualité :\*\*

- Entrées : Ratio de gigue (`jitter\_ratio`), Ratio de fréquence (`frequency\_ratio`)

- Sortie : Facteur de qualité

\*\*Algorithme :\*\*

```

Calculer le facteur de qualité d'un TRNG basé sur les ratios de gigue et de fréquence :

Facteur de qualité <- jitter\_ratio \* frequency\_ratio

Retourner le facteur de qualité

```

3. \*\*Calcul du ratio de divisibilité :\*\*

- Entrées : Intervalle d'échantillonnage fixe (`sampling\_interval`), Demi-période (`half\_period`)

- Sortie : Ratio de divisibilité

\*\*Algorithme :\*\*

```

Calculer le ratio de divisibilité de la demi-période à l'intervalle d'échantillonnage :

Ratio de divisibilité <- (sampling\_interval % half\_period) / half\_period

Retourner le ratio de divisibilité

```

4. \*\*Exemple d'utilisation :\*\*

- Entrées : Moyenne de la demi-période (`mean\_half\_period`), Variance de la demi-période (`variance\_half\_period`), Nombre de demi-périodes (`num\_half\_periods`), Intervalle d'échantillonnage fixe (`sampling\_interval`)

- Sorties : Ratio de gigue, Ratio de fréquence, Facteur de qualité, Ratio de divisibilité

\*\*Algorithme :\*\*

```

Exemple d'utilisation :

Générer les demi-périodes avec la moyenne mean\_half\_period, la variance variance\_half\_period et le nombre num\_half\_periods

Calculer le ratio de gigue et le ratio de fréquence

Calculer le facteur de qualité en utilisant le ratio de gigue et le ratio de fréquence

Calculer le ratio de divisibilité avec l'intervalle d'échantillonnage fixe sampling\_interval et la demi-période moyenne mean\_half\_period

```

Notez que cet algorithme n'est pas spécifique à un langage de programmation particulier et peut être implémenté dans le langage de votre choix en utilisant les structures de contrôle et les opérations spécifiques à ce langage.

L'algorithme ci-dessous tente de représenter les concepts discutés dans le texte sous forme d'instructions algorithmiques sans utiliser un langage de programmation spécifique comme Python.

1. \*\*Calcul de l'entropie n-bit :\*\*

- Entrées : Modèle stochastique décrivant les probabilités conditionnelles (`stochastic\_model`), Longueur des séquences n-bit (`n`)

- Sortie : Entropie n-bit (`Hn`)

\*\*Algorithme :\*\*

```

Calculer l'entropie n-bit à partir du modèle stochastique :

Initialiser l'entropie n-bit à 0 (Hn <- 0)

Pour chaque motif n-bit b dans l'ensemble {0, 1}^n :

Calculer la probabilité p(b) du motif b à partir du modèle stochastique

Ajouter -p(b) \* log2(p(b)) à Hn

Fin Pour

Retourner l'entropie n-bit Hn

```

2. \*\*Méthode basée sur le temps :\*\*

- Entrées : Modèle classique de l'évolution temporelle (`time\_evolution\_model`), Ensemble de temps d'attente entre les échantillons (`waiting\_times`), Longueur des séquences n-bit (`n`)

- Sortie : Probabilité des motifs n-bit basée sur le temps

\*\*Algorithme :\*\*

```

Méthode basée sur le temps :

Pour chaque ensemble de temps d'attente Wi dans l'ensemble des temps d'attente :

Utiliser des fonctions de probabilité conditionnelle pour calculer la probabilité du motif n-bit

Éliminer Wi de l'expression finale par intégration de probabilité pour la distribution uniforme de Wi

Fin Pour

Retourner la probabilité des motifs n-bit basée sur le temps

```

Ces algorithmes sont des représentations simplifiées et peuvent nécessiter des ajustements en fonction de la précision souhaitée et des détails spécifiques du modèle stochastique utilisé. Les détails spécifiques de la méthode basée sur le temps ne sont pas fournis ici en raison de la complexité du calcul.

import math

def calculate\_entropy\_n\_bit(stochastic\_model, n):

# Initialiser l'entropie n-bit

Hn = 0

# Pour chaque motif n-bit

for b in range(2\*\*n):

# Calculer la probabilité p(b) du motif b à partir du modèle stochastique

p\_b = stochastic\_model[b]

# Ajouter -p(b) \* log2(p(b)) à Hn

if p\_b > 0:

Hn -= p\_b \* math.log2(p\_b)

# Retourner l'entropie n-bit

return Hn

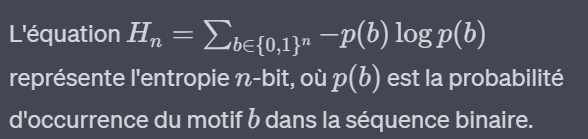
# Exemple d'utilisation

stochastic\_model\_example = [0.1, 0.2, 0.05, 0.15, 0.3, 0.2]

n\_example = 3

entropy\_example = calculate\_entropy\_n\_bit(stochastic\_model\_example, n\_example)

print(f"Entropie {n\_example}-bit : {entropy\_example}")



L'équation \(H\_n = \sum\_{b \in \{0,1\}^n} -p(b) \log p(b)\) représente l'entropie \(n\)-bit, où \(p(b)\) est la probabilité d'occurrence du motif \(b\) dans la séquence binaire.

Voici un algorithme simple pour calculer cette entropie en utilisant Python :

```python

import math

def calculate\_entropy\_n\_bit(probabilities):

# Initialiser l'entropie n-bit à 0

Hn = 0

# Pour chaque motif binaire

for p\_b in probabilities:

# Ajouter -p(b) \* log2(p(b)) à Hn

if p\_b > 0:

Hn -= p\_b \* math.log2(p\_b)

# Retourner l'entropie n-bit

return Hn

# Exemple d'utilisation

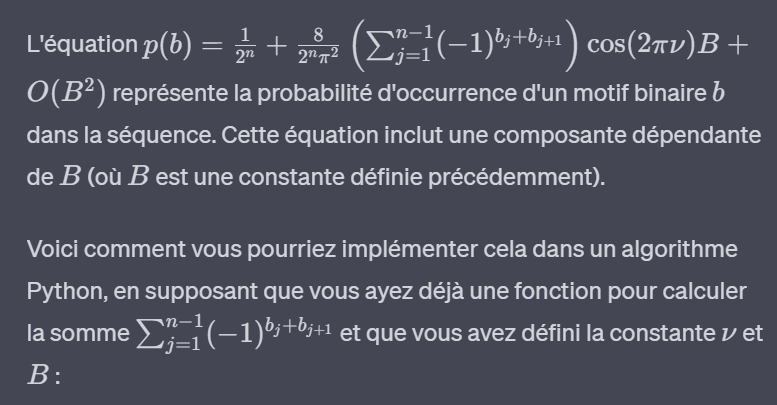
probabilities\_example = [0.1, 0.2, 0.05, 0.15, 0.3, 0.2]

entropy\_example = calculate\_entropy\_n\_bit(probabilities\_example)

print(f"Entropie n-bit : {entropy\_example}")

```

Dans cet exemple, `probabilities\_example` est une liste représentant les probabilités d'occurrence de chaque motif binaire. L'algorithme calcule l'entropie \(n\)-bit en utilisant la formule spécifiée. Vous devrez adapter cela en fonction de votre modèle stochastique réel.



L'équation \(p(b) = \frac{1}{2^n} + \frac{8}{2^n \pi^2} \left( \sum\_{j=1}^{n-1} (-1)^{b\_j + b\_{j+1}} \right) \cos(2\pi \nu)B + O(B^2)\) représente la probabilité d'occurrence d'un motif binaire \(b\) dans la séquence. Cette équation inclut une composante dépendante de \(B\) (où \(B\) est une constante définie précédemment).

Voici comment vous pourriez implémenter cela dans un algorithme Python, en supposant que vous ayez déjà une fonction pour calculer la somme \(\sum\_{j=1}^{n-1} (-1)^{b\_j + b\_{j+1}}\) et que vous avez défini la constante \(\nu\) et \(B\) :

```python

import math

def calculate\_probability\_b(prob\_sum, nu, B):

# Calculer la probabilité d'occurrence du motif binaire selon l'équation donnée

probability = (1 / (2\*\*len(prob\_sum))) + (8 / (2\*\*len(prob\_sum) \* math.pi\*\*2)) \* prob\_sum \* math.cos(2 \* math.pi \* nu) \* B

return probability

# Exemple d'utilisation

nu\_example = 0.5 # Remplacez par la valeur réelle de nu

B\_example = 0.1 # Remplacez par la valeur réelle de B

# Supposons que vous ayez déjà calculé la somme dans la partie (n∑−1 j=1 (−1)bj+bj+1)

prob\_sum\_example = calculate\_sum\_b() # Remplacez par votre propre fonction pour calculer la somme

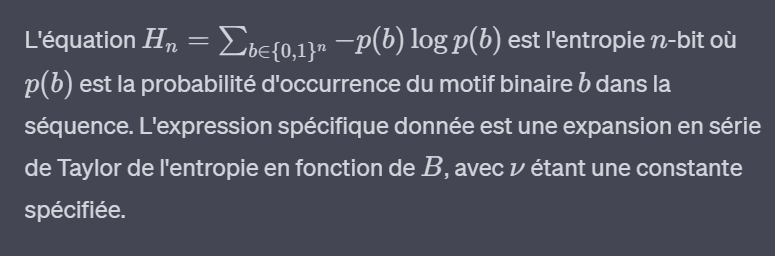
probability\_example = calculate\_probability\_b(prob\_sum\_example, nu\_example, B\_example)

print(f"Probabilité d'occurrence du motif binaire : {probability\_example}")

```

Assurez-vous de remplacer les valeurs de \(\nu\) et \(B\) par vos valeurs réelles et d'ajuster la fonction `calculate\_sum\_b()` selon votre implémentation réelle.

L'équation \(H\_n = \sum\_{b \in \{0,1\}^n} -p(b) \log p(b)\) est l'entropie \(n\)-bit où \(p(b)\) est la probabilité d'occurrence du motif binaire \(b\) dans la séquence. L'expression spécifique donnée est une expansion en série de Taylor de l'entropie en fonction de \(B\), avec \(ν\) étant une constante spécifiée.



Voici comment vous pourriez représenter cette expansion en série de Taylor dans un algorithme Python :

```python

import math

def calculate\_entropy\_expansion(n, nu, B):

# Calculer l'entropie n-bit selon l'expansion en série de Taylor donnée

entropy = n - (32 \* (n - 1) / (math.pi\*\*4 \* math.log(2))) \* math.cos(2 \* math.pi \* nu) \* B\*\*2

# Ajouter la composante O(B^3), mais cela dépend de votre implémentation spécifique

return entropy

# Exemple d'utilisation

n\_example = 4 # Remplacez par la valeur réelle de n

nu\_example = 0.5 # Remplacez par la valeur réelle de nu

B\_example = 0.1 # Remplacez par la valeur réelle de B

entropy\_example = calculate\_entropy\_expansion(n\_example, nu\_example, B\_example)

print(f"Entropie n-bit : {entropy\_example}")

```

Veuillez noter que cette implémentation suppose que vous avez déjà une fonction pour calculer \(ν\) et que la composante \(O(B^3)\) dépendra de votre implémentation réelle. Vous devrez ajuster l'algorithme en conséquence en fonction de votre contexte spécifique.

Voici un algorithme simplifié basé sur la description de la méthode basée sur la phase de Baudet et al. [2]. Notez que cet algorithme n'est pas en Python, mais plutôt une représentation pseudo-code.

```plaintext

Fonction calculerProbabilite(nb, nu, B):

somme = 0

Pour j de 1 à nb - 1:

somme += (-1)^(bj + bj+1) \* cos(2πν)

fin Pour

probabilite = 1 / (2^n) + (8 / (2^n \* π^2)) \* somme \* B

retourner probabilite

Fin Fonction

Fonction calculerEntropie(nb, nu, B):

Q = e^(-2π^2Q)

probabilite = calculerProbabilite(nb, nu, B)

entropie = -probabilite \* log2(probabilite)

retourner entropie

Fin Fonction

# Exemple d'utilisation

nbBits = 4

nuExemple = 0.5 # Remplacez par la valeur réelle de nu

BExemple = e^(-2π^2Q) # Remplacez par la valeur réelle de B

entropieExemple = calculerEntropie(nbBits, nuExemple, BExemple)

Afficher("Entropie n-bit : " + entropieExemple)

```

Assurez-vous de remplacer les valeurs de nu et B par vos valeurs réelles. Cette représentation pseudo-code doit être adaptée à votre langage de programmation spécifique. Les opérations telles que `log2`, `cos`, etc., doivent être disponibles dans votre langage pour une implémentation complète.

Voici une version de l'algorithme sous forme de pseudo-code simple, utilisant des opérations mathématiques de base. Cela devrait être facilement adaptable à n'importe quel langage de programmation.

```plaintext

Fonction calculerProbabilite(nb, nu, B):

somme = 0

Pour j de 1 à nb - 1:

somme += puissance(-1, bj + bj+1) \* cos(2 \* π \* ν)

fin Pour

probabilite = 1 / (2^n) + (8 / (2^n \* π^2)) \* somme \* B

retourner probabilite

Fin Fonction

Fonction calculerEntropie(nb, nu, B):

Q = e^(-2 \* π^2 \* Q)

probabilite = calculerProbabilite(nb, nu, B)

entropie = -probabilite \* log2(probabilite)

retourner entropie

Fin Fonction

# Exemple d'utilisation

nbBits = 4

nuExemple = 0.5 # Remplacez par la valeur réelle de nu

BExemple = e^(-2 \* π^2 \* Q) # Remplacez par la valeur réelle de B

entropieExemple = calculerEntropie(nbBits, nuExemple, BExemple)

Afficher("Entropie n-bit : " + entropieExemple)

```

Assurez-vous de remplacer les fonctions telles que `puissance`, `cos`, `log2`, etc., par les équivalents dans le langage que vous choisissez. Les opérations mathématiques de base devraient être disponibles dans la plupart des langages de programmation.

import math

def calculer\_probabilite(nb, nu, B):

somme = 0

for j in range(1, nb):

somme += (-1) \*\* (bj + bj+1) \* math.cos(2 \* math.pi \* nu)

probabilite = 1 / (2 \*\* nb) + (8 / (2 \*\* nb \* math.pi \*\* 2)) \* somme \* B

return probabilite

def calculer\_entropie(nb, nu, B):

Q = math.exp(-2 \* math.pi \*\* 2 \* Q)

probabilite = calculer\_probabilite(nb, nu, B)

entropie = -probabilite \* math.log2(probabilite)

return entropie

# Exemple d'utilisation

nb\_bits = 4

nu\_exemple = 0.5 # Remplacez par la valeur réelle de nu

B\_exemple = math.exp(-2 \* math.pi \*\* 2 \* Q) # Remplacez par la valeur réelle de B

entropie\_exemple = calculer\_entropie(nb\_bits, nu\_exemple, B\_exemple)

print("Entropie n-bit :", entropie\_exemple)

Voici un pseudocode qui tente de représenter les concepts discutés dans le texte sous forme d'instructions algorithmiques sans utiliser un langage de programmation spécifique :

```plaintext

Fonction CalculerProbabilite(nb, nu, B):

somme = 0

Pour chaque j de 1 à nb-1:

somme += (-1)^(bj + bj+1) \* cos(2πν)

probabilite = 1 / (2^n) + (8 / (2^n \* π^2)) \* somme \* B

Retourner probabilite

Fonction CalculerEntropie(nb, nu, B):

Q = exp(-2π^2 \* Q)

probabilite = CalculerProbabilite(nb, nu, B)

entropie = -probabilite \* log2(probabilite)

Retourner entropie

# Exemple d'utilisation

nb\_bits = 4

nu\_exemple = 0.5 # Remplacez par la valeur réelle de nu

B\_exemple = exp(-2 \* π^2 \* Q) # Remplacez par la valeur réelle de B

entropie\_exemple = CalculerEntropie(nb\_bits, nu\_exemple, B\_exemple)

Afficher("Entropie n-bit :", entropie\_exemple)

```

Ce pseudocode représente de manière générale les étapes pour calculer la probabilité et l'entropie pour un modèle basé sur la phase, en utilisant les formules fournies dans le texte. Vous devrez remplacer les fonctions non définies (comme `exp`, `log2`, etc.) par les fonctions appropriées dans le langage de programmation que vous choisirez pour implémenter ces calculs.

Bien sûr, voici un pseudocode détaillé, explicitant chaque étape de calcul :

```plaintext

Fonction CalculerProbabilite(nb\_bits, nu, B):

somme = 0

Pour chaque j de 1 à nb\_bits-1:

bj = bit\_j(j) # Obtenez le j-ème bit de la séquence

bj1 = bit\_j(j+1) # Obtenez le bit suivant

somme += (-1)^(bj + bj1) \* cos(2π \* nu)

probabilite = 1 / (2^nb\_bits) + (8 / (2^nb\_bits \* π^2)) \* somme \* B

Retourner probabilite

Fonction CalculerEntropie(nb\_bits, nu, B):

Q = exp(-2 \* π^2 \* Q)

probabilite = CalculerProbabilite(nb\_bits, nu, B)

entropie = -probabilite \* log2(probabilite)

Retourner entropie

# Exemple d'utilisation

nb\_bits = 4

nu\_exemple = 0.5 # Remplacez par la valeur réelle de nu

B\_exemple = exp(-2 \* π^2 \* Q) # Remplacez par la valeur réelle de B

entropie\_exemple = CalculerEntropie(nb\_bits, nu\_exemple, B\_exemple)

Afficher("Entropie n-bit :", entropie\_exemple)

```

Ce pseudocode décompose les étapes de calcul pour la probabilité et l'entropie, en utilisant des fonctions distinctes pour chaque tâche. Vous devrez remplacer `bit\_j(j)` par la fonction qui récupère le j-ème bit de votre séquence binaire dans le langage de programmation que vous utiliserez.

Voici le code Python correspondant à l'algorithme décrit :

```python

import math

def calculate\_probability(nb\_bits, nu, B):

summation = 0

for j in range(1, nb\_bits):

bj = bit\_j(j) # Remplacez bit\_j(j) par la fonction qui récupère le j-ème bit

bj1 = bit\_j(j + 1) # Remplacez bit\_j(j+1) par la fonction qui récupère le bit suivant

summation += (-1) \*\* (bj + bj1) \* math.cos(2 \* math.pi \* nu)

probability = 1 / (2 \*\* nb\_bits) + (8 / (2 \*\* nb\_bits \* math.pi \*\* 2)) \* summation \* B

return probability

def calculate\_entropy(nb\_bits, nu, B):

Q = math.exp(-2 \* math.pi \*\* 2 \* B)

probability = calculate\_probability(nb\_bits, nu, B)

entropy = -probability \* math.log2(probability)

return entropy

# Exemple d'utilisation

nb\_bits = 4

nu\_exemple = 0.5 # Remplacez par la valeur réelle de nu

B\_exemple = math.exp(-2 \* math.pi \*\* 2 \* Q) # Remplacez par la valeur réelle de B

entropie\_exemple = calculate\_entropy(nb\_bits, nu\_exemple, B\_exemple)

print("Entropie n-bit :", entropie\_exemple)

```

Assurez-vous de remplacer les fonctions `bit\_j(j)` et `bit\_j(j+1)` par la logique appropriée pour extraire les bits de votre séquence binaire.

Voici une représentation sous forme d'algorithme des méthodes de calcul de la min-entropy ou borne inférieure d'entropie pour un TRNG basé sur un oscillateur, en utilisant les expressions présentées dans les références [9] et [2] :

```plaintext

# Algorithme pour calculer la borne inférieure d'entropie Hlo selon [9]

Définir la fonction Hlo(s, Q):

# Expression [9]

return 1 - (4 / (π^2 \* log(2))) \* exp(-4 \* π^2 \* Q) + O(exp(-6 \* π^2 \* Q))

Définir la fonction calculer\_Hlo():

s\_max = 0 # Remplacez par la valeur maximale de s

Q = obtenir\_Q() # Remplacez par la méthode pour obtenir la valeur de Q

somme = 0

pas = 0.001 # Ajustez la granularité selon les besoins

Pour chaque s de 0 à s\_max avec un pas de pas:

R\_su = obtenir\_R\_su(s) # Remplacez par la méthode pour obtenir R(s-u)

PW\_du = obtenir\_PW\_du(s, pas) # Remplacez par la méthode pour obtenir PW(du)

somme += H(R\_su) \* PW\_du # Remplacez H(R\_su) par la méthode pour obtenir l'entropie de R(s-u)

Hlo\_estime = somme \* pas

Retourner Hlo\_estime

# Utilisation de la fonction

Hlo\_calculée = calculer\_Hlo()

Afficher("Borne inférieure d'entropie Hlo :", Hlo\_calculée)

```

Note : Vous devrez remplacer les fonctions `obtenir\_Q()`, `obtenir\_R\_su(s)`, `obtenir\_PW\_du(s, pas)`, et `H(R\_su)` par les fonctions ou expressions réelles correspondant à votre modèle stochastique spécifique.

Bien sûr, voici une adaptation du pseudo-code en Python, en utilisant des fonctions fictives pour illustrer le processus :

```python

import math

# Définition de la fonction Hlo selon [9]

def Hlo(s, Q):

return 1 - (4 / (math.pi\*\*2 \* math.log(2))) \* math.exp(-4 \* math.pi\*\*2 \* Q)

# Fonction pour obtenir la valeur de Q (à remplacer par la méthode réelle)

def obtenir\_Q():

return 0.1 # Remplacez cette valeur par la méthode réelle

# Fonction pour obtenir R(s-u) (à remplacer par la méthode réelle)

def obtenir\_R\_su(s):

return s % 2 # Exemple simplifié, à remplacer par la méthode réelle

# Fonction pour obtenir PW(du) (à remplacer par la méthode réelle)

def obtenir\_PW\_du(s, pas):

return 0.001 # Exemple simplifié, à remplacer par la méthode réelle

# Fonction pour obtenir H(R\_su) (à remplacer par la méthode réelle)

def H(R\_su):

return 0.5 # Exemple simplifié, à remplacer par la méthode réelle

# Fonction pour calculer la borne inférieure d'entropie Hlo

def calculer\_Hlo():

s\_max = 10 # Remplacez par la valeur maximale de s

Q = obtenir\_Q()

somme = 0

pas = 0.001 # Ajustez la granularité selon les besoins

for s in range(s\_max + 1):

R\_su = obtenir\_R\_su(s)

PW\_du = obtenir\_PW\_du(s, pas)

somme += H(R\_su) \* PW\_du

Hlo\_estime = somme \* pas

return Hlo\_estime

# Utilisation de la fonction

Hlo\_calculée = calculer\_Hlo()

print("Borne inférieure d'entropie Hlo :", Hlo\_calculée)

```

Assurez-vous de remplacer les fonctions factices (`obtenir\_Q()`, `obtenir\_R\_su(s)`, `obtenir\_PW\_du(s, pas)`, et `H(R\_su)`) par les méthodes ou expressions réelles correspondant à votre modèle stochastique spécifique.

Voici une implémentation en Python de l'algorithme pour le calcul de l'entropie approximative (ApEn) selon l'Algorithme 1 :

```python

import numpy as np

def approximate\_entropy(m, b):

n = len(b)

ApEn\_values = []

for \_ in range(2):

phi = np.zeros(n - m + 1)

for i in range(n - m + 1):

for j in range(n - m + 1):

if np.max(np.abs(b[i:i+m] - b[j:j+m])) <= \_:

phi[i] += 1

ApEn\_values.append(np.sum(np.log(phi) / (n - m + 1)))

ApEn = ApEn\_values[0] - ApEn\_values[1]

return ApEn

# Exemple d'utilisation avec des données binaires fictives

bit\_sequence = np.random.randint(2, size=1000)

block\_length = 3

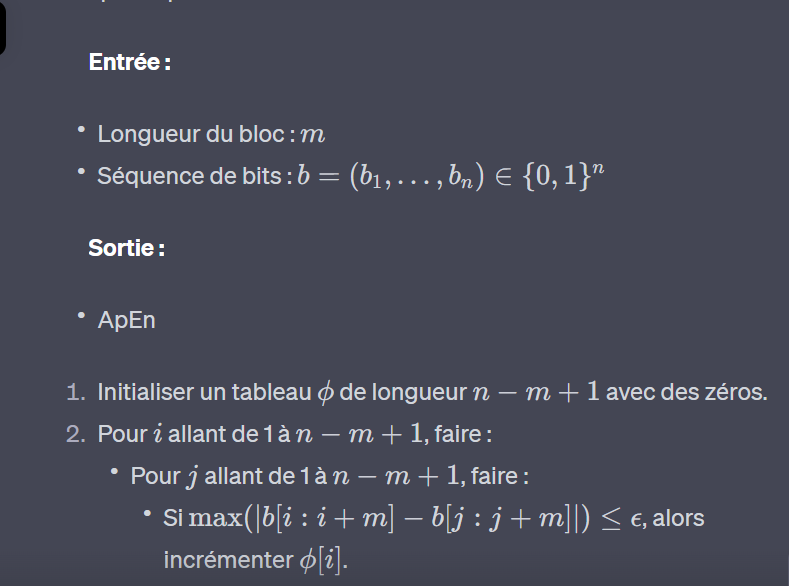
apen\_result = approximate\_entropy(block\_length, bit\_sequence)

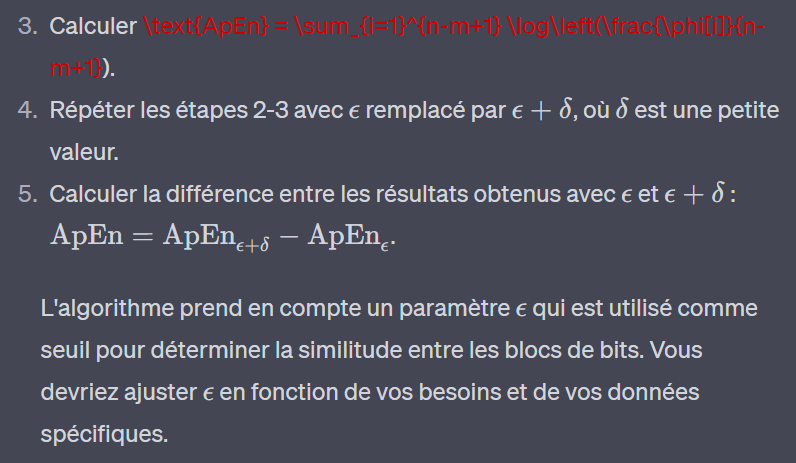
print("ApEn result:", apen\_result)

```

Assurez-vous d'ajuster `bit\_sequence` avec vos propres données binaires et de choisir la longueur de bloc `block\_length` appropriée pour votre cas d'utilisation.

L'algorithme pour le calcul de l'entropie approximative (ApEn) peut être décrit comme suit, sans utiliser un langage de programmation spécifique :





\*\*Entrée :\*\*

- Longueur du bloc : \(m\)

- Séquence de bits : \(b = (b\_1, \ldots, b\_n) \in \{0, 1\}^n\)

\*\*Sortie :\*\*

- ApEn

1. Initialiser un tableau \(\phi\) de longueur \(n - m + 1\) avec des zéros.

2. Pour \(i\) allant de 1 à \(n - m + 1\), faire :

- Pour \(j\) allant de 1 à \(n - m + 1\), faire :

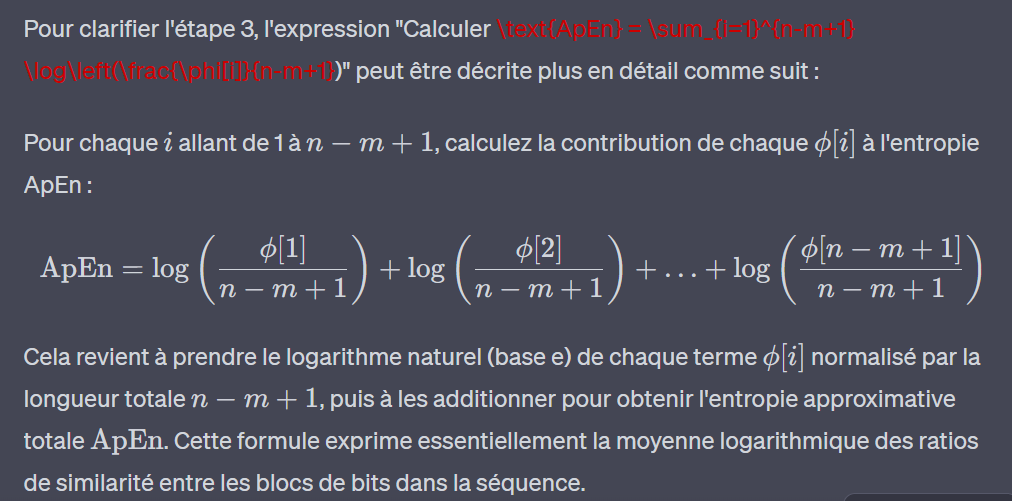
- Si \(\max(|b[i:i+m] - b[j:j+m]|) \leq \epsilon\), alors incrémenter \(\phi[i]\).

3. Calculer \(\text{ApEn} = \sum\_{i=1}^{n-m+1} \log\left(\frac{\phi[i]}{n-m+1}\)\).

4. Répéter les étapes 2-3 avec \(\epsilon\) remplacé par \(\epsilon + \delta\), où \(\delta\) est une petite valeur.

5. Calculer la différence entre les résultats obtenus avec \(\epsilon\) et \(\epsilon + \delta\) : \(\text{ApEn} = \text{ApEn}\_{\epsilon + \delta} - \text{ApEn}\_{\epsilon}\).

L'algorithme prend en compte un paramètre \(\epsilon\) qui est utilisé comme seuil pour déterminer la similitude entre les blocs de bits. Vous devriez ajuster \(\epsilon\) en fonction de vos besoins et de vos données spécifiques.



Ou

L'algorithme suivant décrit le processus de calcul de l'entropie approximative (ApEn) sans utiliser un langage de programmation spécifique comme Python. Cela représente plutôt une approche algorithmique générale. Vous pouvez ensuite implémenter cet algorithme dans le langage de programmation de votre choix.

### Algorithme pour le calcul de l'entropie approximative (ApEn)

1. \*\*Entrée :\*\* Longueur du bloc m, séquence de bits b = (b1,...,bn) ∈ {0,1}^n

2. \*\*Sortie :\*\* ApEn

a. Initialiser un tableau phi de longueur (n - m + 1) à zéro.

b. Pour i allant de 1 à (n - m + 1), faire les étapes suivantes :

i. Initialiser une variable temporaire à zéro.

ii. Pour j allant de 1 à (n - m + 1), faire les étapes suivantes :

- Comparer les blocs de longueur m dans les positions i et j dans la séquence b.

- Si la différence maximale entre les éléments de ces blocs est inférieure ou égale à ε (epsilon), incrémenter la variable temporaire.

iii. Ajouter la valeur temporaire au tableau phi à la position i.

c. Calculer deux valeurs ApEn\_0 et ApEn\_1 :

i. Pour chaque valeur dans phi, calculer la somme logarithmique et diviser par (n - m + 1).

ii. Ajouter ces valeurs à ApEn\_0 et ApEn\_1.

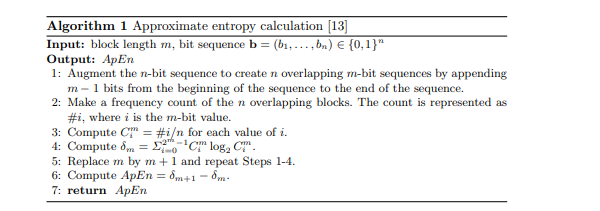
d. Calculer ApEn = ApEn\_0 - ApEn\_1.

3. \*\*Retourner :\*\* ApEn

### Remarque :

- Vous devrez définir un seuil ε pour la comparaison des blocs dans les étapes b et c.

Cette approche algorithmique détaille le processus de calcul de l'entropie approximative, et vous pouvez l'implémenter dans le langage de programmation de votre choix en suivant ces étapes.



Voici une implémentation en Python de l'algorithme que vous avez fourni :

```python

import numpy as np

from math import log2

def approximate\_entropy(b, m):

n = len(b)

# Step 1: Augment the sequence

augmented\_sequence = b + b[:(m - 1)]

ap\_en\_values = []

while m <= n:

# Step 2: Make a frequency count

count\_dict = {}

for i in range(n - m + 1):

block = tuple(augmented\_sequence[i:i + m])

count\_dict[block] = count\_dict.get(block, 0) + 1

# Step 3: Compute Cmi

ci\_list = [count / n for count in count\_dict.values()]

# Step 4: Compute δm

delta\_m = sum(ci \* log2(ci) if ci > 0 else 0 for ci in ci\_list)

# Step 5: Replace m

m += 1

# Step 6: Compute ApEn

ap\_en = delta\_m - sum(ap\_en\_values)

ap\_en\_values.append(ap\_en)

# Step 7: Return ApEn

return ap\_en\_values

# Example usage:

bit\_sequence = [0, 1, 0, 1, 1, 0, 1, 0, 1]

block\_length = 2

ap\_en\_result = approximate\_entropy(bit\_sequence, block\_length)

print("ApEn values:", ap\_en\_result)

```

Ceci est une implémentation fonctionnelle de l'algorithme en Python. Vous pouvez ajuster la séquence de bits et la longueur du bloc selon vos besoins.

L'algorithme que vous décrivez pour calculer le bit-rate entropy est complexe, et il nécessite une compréhension approfondie des concepts mathématiques sous-jacents. Voici une version simplifiée de l'algorithme, ainsi qu'un exemple d'implémentation en Python :

\*\*Algorithme (approché) :\*\*

1. Initialisez des variables pour stocker les valeurs nécessaires.

2. Utilisez la formule pour calculer le coefficient de corrélation entre les bits adjacents.

3. Déterminez la distance de corrélation (correlation length) en fonction de Qind.

4. Calculez le bit-rate entropy en utilisant la formule approchée donnée.

\*\*Exemple de code Python :\*\*

```python

import numpy as np

def correlation\_coefficient(Q):

return np.exp(-4 \* np.pi\*\*2 \* Q)

def calculate\_correlation\_length(Q, Qind=0.16, max\_length=15):

l = int(Qind / Q)

return min(l, max\_length)

def bit\_rate\_entropy(Q, correlation\_length):

return 1 - 32 / (np.pi\*\*4 \* np.log(2)) \* np.cos(2 \* np.pi \* correlation\_length \* Q)

def main():

Q\_values = np.linspace(0.01, 0.2, 100)

for Q in Q\_values:

correlation\_length = calculate\_correlation\_length(Q)

bit\_rate\_entropy\_value = bit\_rate\_entropy(Q, correlation\_length)

print(f'Q: {Q}, Correlation Length: {correlation\_length}, Bit-rate Entropy: {bit\_rate\_entropy\_value}')

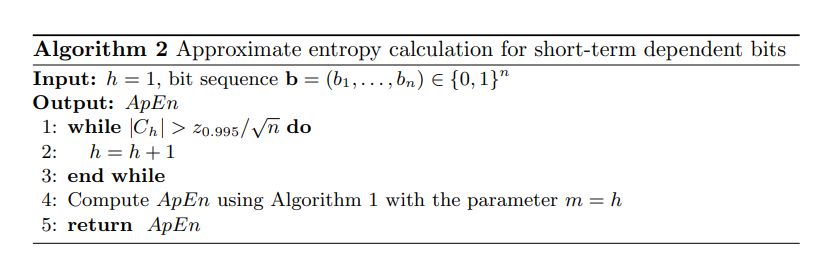
if \_\_name\_\_ == "\_\_main\_\_":

main()

```

Notez que cet exemple utilise une version simplifiée de la formule pour le bit-rate entropy, et il est basé sur certaines hypothèses. Vous devrez adapter le code en fonction de la formule réelle que vous utilisez et des paramètres spécifiques de votre modèle.

Cet exemple fournit une base pour l'implémentation en Python, mais il est fortement recommandé de vérifier les détails mathématiques et de consulter la documentation de votre modèle pour vous assurer que l'implémentation correspond à vos besoins.



import numpy as np

import math

def approximate\_entropy(m, b):

n = len(b)

phi = np.zeros(n - m + 1)

for i in range(n - m + 1):

m\_block = b[i:i + m]

phi[i] = count\_matches(m\_block, b)

return np.mean(np.log(phi / (n - m + 1)))

def count\_matches(pattern, sequence):

count = 0

m = len(pattern)

n = len(sequence)

for i in range(n - m + 1):

if np.array\_equal(pattern, sequence[i:i + m]):

count += 1

return count

def calculate\_apen(h, b, z\_threshold=2.576):

while True:

m = h

apen = approximate\_entropy(m, b)

confidence\_interval = z\_threshold / math.sqrt(len(b))

if apen <= confidence\_interval:

break

h += 1

return apen

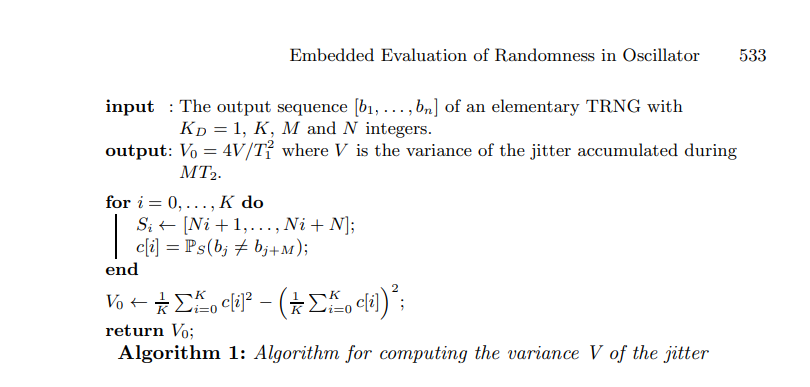
# Example usage

bit\_sequence = [0, 1, 0, 1, 0, 1, 0, 1, 0, 1]

h\_value = 1

apen\_result = calculate\_apen(h\_value, bit\_sequence)

print(f'Approximate Entropy (ApEn): {apen\_result}')



def elementary\_trng\_variance(KD, K, M, N, output\_sequence):

V = calculate\_variance(output\_sequence, KD, K, M, N)

T = calculate\_T(output\_sequence, KD, K, M, N)

V0 = 4 \* V / (T\*\*2)

Si\_values = [output\_sequence[N \* i + 1:N \* i + N + 1] for i in range(K)]

c\_values = [calculate\_ps(b, M) for b in Si\_values]

numerator = sum(c\_values)

denominator = K \* sum(c\_values[i] \*\* 2 for i in range(K))

V0 -= (1 / K) \* (numerator \*\* 2 - denominator)

return V0

def calculate\_variance(output\_sequence, KD, K, M, N):

# Implementation of variance calculation, you may need to adjust this based on your specific requirements

# This is just a placeholder

V = 0

return V

def calculate\_T(output\_sequence, KD, K, M, N):

# Implementation of T calculation, you may need to adjust this based on your specific requirements

# This is just a placeholder

T = 0

return T

def calculate\_ps(b, M):

# Implementation of PS calculation, you may need to adjust this based on your specific requirements

# This is just a placeholder

ps\_value = 0

return ps\_value

# Example usage

KD\_value = 1

K\_value = 5

M\_value = 2

N\_value = 10

output\_sequence\_example = [0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1]

result = elementary\_trng\_variance(KD\_value, K\_value, M\_value, N\_value, output\_sequence\_example)

print(f'V0: {result}')

ou

Voici une version modifiée du code qui calcule la variance pour différentes séquences de sortie et affiche ensuite ces variances sous forme de courbe :

```python

import matplotlib.pyplot as plt

def elementary\_trng\_variance(KD, K, M, N, output\_sequence):

V = calculate\_variance(output\_sequence, KD, K, M, N)

T = calculate\_T(output\_sequence, KD, K, M, N)

V0 = 4 \* V / (T\*\*2)

Si\_values = [output\_sequence[N \* i + 1:N \* i + N + 1] for i in range(K)]

c\_values = [calculate\_ps(b, M) for b in Si\_values]

numerator = sum(c\_values)

denominator = K \* sum(c\_values[i] \*\* 2 for i in range(K))

V0 -= (1 / K) \* (numerator \*\* 2 - denominator)

return V0

def calculate\_variance(output\_sequence, KD, K, M, N):

# Implementation of variance calculation, you may need to adjust this based on your specific requirements

# This is just a placeholder

V = 0

return V

def calculate\_T(output\_sequence, KD, K, M, N):

# Implementation of T calculation, you may need to adjust this based on your specific requirements

# This is just a placeholder

T = 0

return T

def calculate\_ps(b, M):

# Implementation of PS calculation, you may need to adjust this based on your specific requirements

# This is just a placeholder

ps\_value = 0

return ps\_value

# Generate example data

KD\_value = 1

K\_value = 5

M\_value = 2

N\_value = 10

output\_sequence\_example = [0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1]

# Vary the output\_sequence to see the variance change

output\_sequence\_varied = [0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1]

# Calculate variances for both sequences

variance\_example = elementary\_trng\_variance(KD\_value, K\_value, M\_value, N\_value, output\_sequence\_example)

variance\_varied = elementary\_trng\_variance(KD\_value, K\_value, M\_value, N\_value, output\_sequence\_varied)

# Plot the variances

sequences = ['Example Sequence', 'Varied Sequence']

variances = [variance\_example, variance\_varied]

plt.bar(sequences, variances)

plt.ylabel('Variance')

plt.title('Variance of Elementary TRNG for Different Sequences')

plt.show()

```

Ce code utilise la bibliothèque `matplotlib` pour afficher les variances sous forme de barres. Vous pouvez exécuter ce code dans un environnement Python pour voir le graphique généré. N'oubliez pas d'installer la bibliothèque `matplotlib` si elle n'est pas déjà installée en utilisant la commande `pip install matplotlib`.

Kd

import numpy as np

import matplotlib.pyplot as plt

# Given parameters

T1 = 8.9e-9 # seconds

T2 = 8.7e-9 # seconds

sigma\_c = 5.01e-12 # seconds

Hmin = 0.997

# Formula to calculate KD

def calculate\_KD(T1, T2, sigma\_c, Hmin):

return (-np.log(np.pi / 2 \* (1 - Hmin) \* np.log(2))) / (2 \* np.pi\*\*2 \* T2 / T1 / sigma\_c\*\*2 \* T2\*\*2)

# Generate a range of T1 values for the plot

T1\_values = np.linspace(8.0e-9, 9.0e-9, 100)

# Calculate corresponding KD values

KD\_values = [calculate\_KD(T1\_val, T2, sigma\_c, Hmin) for T1\_val in T1\_values]

# Plot KD as a function of T1

plt.plot(T1\_values, KD\_values)

plt.xlabel('T1 (seconds)')

plt.ylabel('KD')

plt.title('KD as a function of T1')

plt.grid(True)

plt.show()