L'algorithme AES (Advanced Encryption Standard) est un algorithme de chiffrement symétrique largement utilisé pour sécuriser des données sensibles. Voici une description générale de son fonctionnement :

1. \*\*Clé d'entrée :\*\*

- L'AES peut utiliser des clés de 128, 192 ou 256 bits.

- La clé est étendue en un tableau de clés rondes utilisant une procédure spécifique appelée Key Expansion.

2. \*\*Initialisation :\*\*

- L'état initial (state) est une matrice de 4x4 octets, qui est remplie avec les données à chiffrer.

3. \*\*AddRoundKey :\*\*

- La première opération consiste à combiner chaque octet de l'état avec la clé de tour (round key) à l'aide d'une opération XOR (ou exclusif).

4. \*\*Rounds :\*\*

- L'AES utilise un certain nombre de "rondes" (10 rondes pour AES-128, 12 pour AES-192, et 14 pour AES-256).

- Chaque ronde est composée de quatre opérations :

- \*\*SubBytes :\*\* Chaque octet de l'état est remplacé par un autre octet à l'aide d'une table de substitution (S-box).

- \*\*ShiftRows :\*\* Les octets dans chaque ligne de l'état sont décalés vers la gauche de différents offsets.

- \*\*MixColumns :\*\* Les colonnes de l'état sont mélangées en utilisant une opération de multiplication dans un corps fini.

- \*\*AddRoundKey :\*\* Une nouvelle clé de tour est combinée avec l'état.

5. \*\*Dernière ronde (Round) :\*\*

- La dernière ronde n'inclut pas l'opération MixColumns.

6. \*\*Résultat :\*\*

- Le résultat final est l'état modifié après la dernière ronde. Il est converti en une séquence d'octets pour obtenir le texte chiffré.

L'AES est réputé pour sa sécurité et son efficacité, et il est largement utilisé dans de nombreuses applications, notamment dans la protection des communications en ligne et le chiffrement des données stockées.

L'algorithme AES est généralement implémenté dans des bibliothèques de programmation, mais voici un exemple simple en pseudo-code pour illustrer les étapes de base de l'algorithme AES avec une clé de 128 bits :

```plaintext

Fonction AES(chaine clair, clé)

état = InitialiserEtat(clair)

clésRondes = KeyExpansion(clé)

// Ajouter la clé initiale

état = AddRoundKey(état, clésRondes[0])

// Rondes principales

Pour chaque ronde de 1 à 9

état = SubBytes(état)

état = ShiftRows(état)

état = MixColumns(état)

état = AddRoundKey(état, clésRondes[ronde])

// Dernière ronde (sans MixColumns)

état = SubBytes(état)

état = ShiftRows(état)

état = AddRoundKey(état, clésRondes[10])

// Convertir l'état en texte chiffré

texteChiffré = ConvertirEnOctets(état)

Retourner texteChiffré

Fin de la Fonction

```

Cet exemple utilise des fonctions telles que `SubBytes`, `ShiftRows`, `MixColumns`, et `AddRoundKey`, qui sont les opérations spécifiques à chaque ronde de l'algorithme AES. La fonction `KeyExpansion` est également utilisée pour étendre la clé initiale en un ensemble de clés de tour.

Veuillez noter que dans une implémentation réelle, les opérations et les tables de substitution seraient gérées de manière plus détaillée, en utilisant des opérations sur les octets et des calculs dans le corps fini.

Tests

Le test en ligne selon AIS31 (Algorithm for Random Number Generator Testing and Test Suite) et les tests du NIST (National Institute of Standards and Technology) sont des approches normalisées pour évaluer la qualité des générateurs de nombres aléatoires. Ces tests sont souvent utilisés pour évaluer les RNG (Random Number Generators) conformes aux normes de sécurité.

Voici une approche générale pour effectuer ces tests en ligne, en supposant que vous utilisez un anneau d'oscillateur comme source d'entropie dans votre RNG :

### Test en ligne selon AIS31 :

1. \*\*Initialisation :\*\*

- Configurez le générateur de nombres aléatoires avec la source d'entropie basée sur l'anneau d'oscillateur.

2. \*\*Collecte de données :\*\*

- Commencez à générer des nombres aléatoires en utilisant votre RNG.

3. \*\*Test de distribution uniforme :\*\*

- Vérifiez que la distribution des nombres générés est uniforme en utilisant des tests statistiques appropriés.

4. \*\*Test de l'indépendance séquentielle :\*\*

- Vérifiez que les nombres générés sont indépendants les uns des autres.

5. \*\*Test de l'absence de corrélation :\*\*

- Vérifiez qu'il n'y a pas de corrélation entre les chiffres successifs.

### Tests du NIST :

1. \*\*Test du Monobit :\*\*

- Vérifiez que le nombre de 1 et de 0 dans la séquence générée est approximativement équilibré.

2. \*\*Test du Poker :\*\*

- Évaluez la distribution des motifs dans la séquence générée.

3. \*\*Test de série :\*\*

- Examinez la distribution des séries de bits consécutifs.

4. \*\*Test de longueur de série :\*\*

- Vérifiez la distribution des longueurs de séries consécutives de bits identiques.

5. \*\*Test du poker en blocs :\*\*

- Évaluez la distribution des motifs dans des blocs de bits.

### Considérations spécifiques à l'anneau d'oscillateur :

- Assurez-vous que l'anneau d'oscillateur fonctionne de manière stable et qu'il génère suffisamment d'entropie.

- Garantissez que le processus de génération de nombres aléatoires basé sur l'anneau d'oscillateur est correctement implémenté.

- Effectuez des tests supplémentaires spécifiques à votre implémentation pour garantir la sécurité de l'algorithme.

La mise en œuvre spécifique des tests de générateur de nombres aléatoires peut varier en fonction des normes et des recommandations spécifiques de l'AIS31 et du NIST. Ces tests sont généralement effectués en utilisant des suites de tests statistiques prédéfinies.

Voici un exemple très simplifié d'un test de générateur de nombres aléatoires basé sur le test du Monobit du NIST. Cet exemple est fourni à des fins d'illustration et n'est pas destiné à être utilisé dans des applications réelles sans adaptation appropriée :

```python

def monobit\_test(random\_sequence):

# Fonction de test du Monobit du NIST

count\_ones = sum(int(bit) for bit in random\_sequence)

# Effectuer un test statistique sur la distribution des bits

n = len(random\_sequence)

s\_obs = abs(count\_ones - n/2) / (n\*\*0.5)

# Seuil de confiance pour le test du Monobit (niveau de signification de 0.01)

threshold = 2.576

# Résultat du test

if s\_obs <= threshold:

return True # La séquence passe le test

else:

return False # La séquence échoue le test

# Exemple d'utilisation avec une séquence de bits aléatoires (0 et 1)

random\_sequence = "11001010101111001100011010101010110011110011010101"

result = monobit\_test(random\_sequence)

if result:

print("La séquence passe le test du Monobit.")

else:

print("La séquence échoue le test du Monobit.")

```

Dans cet exemple, la fonction `monobit\_test` mesure la différence entre le nombre d'uns et de zéros dans la séquence et compare cette valeur à un seuil statistique. Le seuil est basé sur la distribution normale, et si la différence observée est inférieure au seuil, la séquence passe le test.

Cependant, pour une évaluation complète de la qualité d'un générateur de nombres aléatoires, il est généralement recommandé d'utiliser des suites de tests plus complètes, comme celles fournies par le NIST ou d'autres organisations de normalisation. Ces suites comprennent plusieurs tests pour évaluer différents aspects de la qualité statistique des séquences générées.

N'oubliez pas que ces tests constituent une évaluation de la qualité statistique de la séquence générée et ne garantissent pas nécessairement la sécurité dans tous les contextes. Consultez les documents de référence de l'AIS31 et du NIST pour des détails spécifiques sur la mise en œuvre de ces tests.

Le test du poker est une méthode pour évaluer la distribution des motifs dans une séquence de nombres aléatoires. Il est souvent utilisé pour détecter des défauts dans un générateur de nombres aléatoires. Voici un exemple simple en Python pour mettre en œuvre le test du poker :

```python

def poker\_test(random\_sequence, block\_size=4):

n = len(random\_sequence)

if n % block\_size != 0:

raise ValueError("La longueur de la séquence doit être divisible par la taille du bloc.")

# Diviser la séquence en blocs

blocks = [random\_sequence[i:i+block\_size] for i in range(0, n, block\_size)]

# Compter les occurrences de chaque motif

pattern\_counts = {}

for block in blocks:

if block not in pattern\_counts:

pattern\_counts[block] = 1

else:

pattern\_counts[block] += 1

# Calculer la statistique du test

chi\_square = sum((count - n/block\_size)\*\*2 / (n/block\_size) for count in pattern\_counts.values())

# Seuil de confiance pour le test du poker (niveau de signification de 0.01)

threshold = 13.277

# Résultat du test

if chi\_square <= threshold:

return True # La séquence passe le test

else:

return False # La séquence échoue le test

# Exemple d'utilisation avec une séquence de chiffres aléatoires (supposant une base 10)

random\_sequence = "31415926535897932384626433832795"

result = poker\_test(random\_sequence)

if result:

print("La séquence passe le test du Poker.")

else:

print("La séquence échoue le test du Poker.")

```

Dans cet exemple, la fonction `poker\_test` prend une séquence de nombres aléatoires sous forme de chaîne de caractères et la divise en blocs de taille spécifiée (`block\_size`). Ensuite, elle compte le nombre d'occurrences de chaque motif dans la séquence et utilise le test du chi carré pour évaluer la distribution des motifs.

Ce code est un exemple simplifié et peut nécessiter des adaptations en fonction des spécificités de votre générateur de nombres aléatoires ou des exigences particulières de votre application. De plus, il est toujours recommandé d'utiliser des bibliothèques de génération de nombres aléatoires éprouvées et testées pour des applications réelles.

Le test de série évalue la distribution des séquences consécutives de bits dans une séquence de nombres aléatoires. Le test vise à détecter tout modèle ou régularité dans la séquence générée. Voici un exemple simple en Python pour mettre en œuvre le test de série :

```python

def serial\_test(random\_sequence, series\_length=3):

n = len(random\_sequence)

# Vérifier que la séquence est suffisamment longue

if n < series\_length:

raise ValueError("La séquence doit être plus longue que la longueur de la série.")

# Diviser la séquence en séries de bits consécutifs

series = [random\_sequence[i:i+series\_length] for i in range(0, n - series\_length + 1)]

# Compter les occurrences de chaque série

series\_counts = {}

for s in series:

if s not in series\_counts:

series\_counts[s] = 1

else:

series\_counts[s] += 1

# Calculer la statistique du test

chi\_square = sum((count - (n - series\_length + 1)/2)\*\*2 / ((n - series\_length + 1)/2) for count in series\_counts.values())

# Seuil de confiance pour le test de série (niveau de signification de 0.01)

threshold = 16.92

# Résultat du test

if chi\_square <= threshold:

return True # La séquence passe le test

else:

return False # La séquence échoue le test

# Exemple d'utilisation avec une séquence de chiffres aléatoires (supposant une base 10)

random\_sequence = "31415926535897932384626433832795"

result = serial\_test(random\_sequence)

if result:

print("La séquence passe le test de série.")

else:

print("La séquence échoue le test de série.")

```

Dans cet exemple, la fonction `serial\_test` prend une séquence de nombres aléatoires sous forme de chaîne de caractères et divise la séquence en séries de bits consécutives de longueur spécifiée (`series\_length`). Ensuite, elle compte le nombre d'occurrences de chaque série et utilise le test du chi carré pour évaluer la distribution des séries.

Comme toujours, cet exemple est simplifié et peut nécessiter des adaptations en fonction des spécificités de votre générateur de nombres aléatoires ou des exigences particulières de votre application. Utilisez des bibliothèques de génération de nombres aléatoires bien établies pour des applications réelles.

Le test de longueur de série évalue la distribution des longueurs des séquences consécutives de bits identiques dans une séquence de nombres aléatoires. Voici un exemple simple en Python pour mettre en œuvre le test de longueur de série :

```python

def run\_length\_test(random\_sequence):

n = len(random\_sequence)

# Initialiser les variables

current\_bit = random\_sequence[0]

current\_length = 1

lengths = []

# Parcourir la séquence et compter les longueurs des séries

for i in range(1, n):

if random\_sequence[i] == current\_bit:

current\_length += 1

else:

lengths.append(current\_length)

current\_bit = random\_sequence[i]

current\_length = 1

# Ajouter la dernière longueur

lengths.append(current\_length)

# Compter les occurrences de chaque longueur

length\_counts = {}

for length in lengths:

if length not in length\_counts:

length\_counts[length] = 1

else:

length\_counts[length] += 1

# Calculer la statistique du test

chi\_square = sum((count - n/2)\*\*2 / (n/2) for count in length\_counts.values())

# Seuil de confiance pour le test de longueur de série (niveau de signification de 0.01)

threshold = 16.92

# Résultat du test

if chi\_square <= threshold:

return True # La séquence passe le test

else:

return False # La séquence échoue le test

# Exemple d'utilisation avec une séquence de chiffres aléatoires (supposant une base 2)

random\_sequence = "1101001111011110000101001101"

result = run\_length\_test(random\_sequence)

if result:

print("La séquence passe le test de longueur de série.")

else:

print("La séquence échoue le test de longueur de série.")

```

Dans cet exemple, la fonction `run\_length\_test` prend une séquence de nombres aléatoires sous forme de chaîne de caractères et parcourt la séquence pour compter les longueurs des séquences consécutives de bits identiques. Elle utilise ensuite le test du chi carré pour évaluer la distribution des longueurs.

Comme toujours, cet exemple est simplifié et peut nécessiter des adaptations en fonction des spécificités de votre générateur de nombres aléatoires ou des exigences particulières de votre application. Utilisez des bibliothèques de génération de nombres aléatoires bien établies pour des applications réelles.

Le test du poker en blocs, également connu sous le nom de test du poker ou chi-squared poker test, vise à évaluer la distribution des motifs dans des blocs de nombres aléatoires. Voici un exemple simple en Python pour mettre en œuvre le test du poker en blocs :

```python

def block\_poker\_test(random\_sequence, block\_size=4):

n = len(random\_sequence)

# Vérifier que la séquence est suffisamment longue

if n % block\_size != 0:

raise ValueError("La longueur de la séquence doit être divisible par la taille du bloc.")

# Diviser la séquence en blocs de taille spécifiée

blocks = [random\_sequence[i:i+block\_size] for i in range(0, n, block\_size)]

# Compter les occurrences de chaque motif de bloc

block\_counts = {}

for block in blocks:

if block not in block\_counts:

block\_counts[block] = 1

else:

block\_counts[block] += 1

# Calculer la statistique du test

chi\_square = sum((count - n/block\_size)\*\*2 / (n/block\_size) for count in block\_counts.values())

# Seuil de confiance pour le test du poker en blocs (niveau de signification de 0.01)

threshold = 13.277

# Résultat du test

if chi\_square <= threshold:

return True # La séquence passe le test

else:

return False # La séquence échoue le test

# Exemple d'utilisation avec une séquence de chiffres aléatoires (supposant une base 10)

random\_sequence = "31415926535897932384626433832795"

result = block\_poker\_test(random\_sequence)

if result:

print("La séquence passe le test du poker en blocs.")

else:

print("La séquence échoue le test du poker en blocs.")

```

Dans cet exemple, la fonction `block\_poker\_test` prend une séquence de nombres aléatoires sous forme de chaîne de caractères et la divise en blocs de taille spécifiée (`block\_size`). Ensuite, elle compte le nombre d'occurrences de chaque motif de bloc et utilise le test du chi carré pour évaluer la distribution des motifs.

Comme toujours, cet exemple est simplifié et peut nécessiter des adaptations en fonction des spécificités de votre générateur de nombres aléatoires ou des exigences particulières de votre application. Utilisez des bibliothèques de génération de nombres aléatoires bien établies pour des applications réelles.

Le test de distribution uniforme vérifie si la séquence de nombres générés par un générateur de nombres aléatoires suit une distribution uniforme. Voici un exemple simple en Python pour mettre en œuvre le test de distribution uniforme :

```python

def uniformity\_test(random\_sequence):

n = len(random\_sequence)

# Compter le nombre d'occurrences de chaque chiffre

digit\_counts = {str(i): 0 for i in range(10)}

for digit in random\_sequence:

digit\_counts[digit] += 1

# Calculer la statistique du test

chi\_square = sum((count - n/10)\*\*2 / (n/10) for count in digit\_counts.values())

# Seuil de confiance pour le test de distribution uniforme (niveau de signification de 0.01)

threshold = 18.307

# Résultat du test

if chi\_square <= threshold:

return True # La séquence passe le test

else:

return False # La séquence échoue le test

# Exemple d'utilisation avec une séquence de chiffres aléatoires (supposant une base 10)

random\_sequence = "31415926535897932384626433832795"

result = uniformity\_test(random\_sequence)

if result:

print("La séquence passe le test de distribution uniforme.")

else:

print("La séquence échoue le test de distribution uniforme.")

```

Dans cet exemple, la fonction `uniformity\_test` prend une séquence de nombres aléatoires sous forme de chaîne de caractères (supposant une base 10) et compte le nombre d'occurrences de chaque chiffre. Elle utilise ensuite le test du chi carré pour évaluer la distribution des chiffres.

Comme toujours, cet exemple est simplifié et peut nécessiter des adaptations en fonction des spécificités de votre générateur de nombres aléatoires ou des exigences particulières de votre application. Utilisez des bibliothèques de génération de nombres aléatoires bien établies pour des applications réelles.

Le test de l'indépendance séquentielle évalue si les nombres générés par un générateur de nombres aléatoires sont statistiquement indépendants les uns des autres. Voici un exemple simple en Python pour mettre en œuvre le test de l'indépendance séquentielle :

```python

def sequential\_independence\_test(random\_sequence, lag=1):

n = len(random\_sequence)

# Vérifier que la séquence est suffisamment longue pour le lag spécifié

if n <= lag:

raise ValueError("La séquence doit être plus longue que la valeur de lag.")

# Créer des paires de nombres consécutifs avec le décalage (lag)

pairs = [(random\_sequence[i], random\_sequence[i + lag]) for i in range(n - lag)]

# Compter les occurrences de chaque paire

pair\_counts = {}

for pair in pairs:

if pair not in pair\_counts:

pair\_counts[pair] = 1

else:

pair\_counts[pair] += 1

# Calculer la statistique du test

chi\_square = sum((count - (n - lag))\*\*2 / (n - lag) for count in pair\_counts.values())

# Seuil de confiance pour le test d'indépendance séquentielle (niveau de signification de 0.01)

threshold = 16.92

# Résultat du test

if chi\_square <= threshold:

return True # La séquence passe le test

else:

return False # La séquence échoue le test

# Exemple d'utilisation avec une séquence de chiffres aléatoires (supposant une base 10)

random\_sequence = "31415926535897932384626433832795"

result = sequential\_independence\_test(random\_sequence)

if result:

print("La séquence passe le test d'indépendance séquentielle.")

else:

print("La séquence échoue le test d'indépendance séquentielle.")

```

Dans cet exemple, la fonction `sequential\_independence\_test` prend une séquence de nombres aléatoires sous forme de chaîne de caractères (supposant une base 10) et crée des paires de nombres consécutifs avec un décalage spécifié (`lag`). Elle compte ensuite le nombre d'occurrences de chaque paire et utilise le test du chi carré pour évaluer l'indépendance séquentielle.

Comme toujours, cet exemple est simplifié et peut nécessiter des adaptations en fonction des spécificités de votre générateur de nombres aléatoires ou des exigences particulières de votre application. Utilisez des bibliothèques de génération de nombres aléatoires bien établies pour des applications réelles.

Le test de l'absence de corrélation vise à évaluer si les nombres générés par un générateur de nombres aléatoires sont corrélés les uns aux autres. Voici un exemple simple en Python pour mettre en œuvre le test de l'absence de corrélation :

```python

def correlation\_absence\_test(random\_sequence, lag=1):

n = len(random\_sequence)

# Vérifier que la séquence est suffisamment longue pour le lag spécifié

if n <= lag:

raise ValueError("La séquence doit être plus longue que la valeur de lag.")

# Créer des paires de nombres consécutifs avec le décalage (lag)

pairs = [(random\_sequence[i], random\_sequence[i + lag]) for i in range(n - lag)]

# Compter les occurrences de chaque paire

pair\_counts = {}

for pair in pairs:

if pair not in pair\_counts:

pair\_counts[pair] = 1

else:

pair\_counts[pair] += 1

# Calculer la statistique du test

chi\_square = sum((count - (n - lag)/2)\*\*2 / ((n - lag)/2) for count in pair\_counts.values())

# Seuil de confiance pour le test d'absence de corrélation (niveau de signification de 0.01)

threshold = 16.92

# Résultat du test

if chi\_square <= threshold:

return True # La séquence passe le test

else:

return False # La séquence échoue le test

# Exemple d'utilisation avec une séquence de chiffres aléatoires (supposant une base 10)

random\_sequence = "31415926535897932384626433832795"

result = correlation\_absence\_test(random\_sequence)

if result:

print("La séquence passe le test d'absence de corrélation.")

else:

print("La séquence échoue le test d'absence de corrélation.")

```

Dans cet exemple, la fonction `correlation\_absence\_test` prend une séquence de nombres aléatoires sous forme de chaîne de caractères (supposant une base 10) et crée des paires de nombres consécutifs avec un décalage spécifié (`lag`). Elle compte ensuite le nombre d'occurrences de chaque paire et utilise le test du chi carré pour évaluer l'absence de corrélation.

Comme toujours, cet exemple est simplifié et peut nécessiter des adaptations en fonction des spécificités de votre générateur de nombres aléatoires ou des exigences particulières de votre application. Utilisez des bibliothèques de génération de nombres aléatoires bien établies pour des applications réelles.

La collecte de données est une étape cruciale pour évaluer la qualité d'un générateur de nombres aléatoires (RNG). Voici un exemple simple en Python illustrant comment vous pourriez collecter des données à partir d'un RNG et effectuer quelques tests de base sur la séquence générée :

```python

import random

def collect\_data(sample\_size):

# Utilisation de la bibliothèque random de Python pour générer une séquence

random\_sequence = [str(random.randint(0, 9)) for \_ in range(sample\_size)]

return "".join(random\_sequence)

def uniformity\_test(random\_sequence):

n = len(random\_sequence)

digit\_counts = {str(i): 0 for i in range(10)}

for digit in random\_sequence:

digit\_counts[digit] += 1

chi\_square = sum((count - n/10)\*\*2 / (n/10) for count in digit\_counts.values())

threshold = 18.307

if chi\_square <= threshold:

return True # La séquence passe le test

else:

return False # La séquence échoue le test

def main():

# Collecte de données

sample\_size = 1000 # Taille de l'échantillon

random\_sequence = collect\_data(sample\_size)

# Test de distribution uniforme

result\_uniformity = uniformity\_test(random\_sequence)

# Affichage des résultats

print(f"Séquence aléatoire générée : {random\_sequence}")

print("Résultats des tests :")

print(f"- Distribution uniforme : {'Passé' if result\_uniformity else 'Échoué'}")

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

main()

```

Dans cet exemple, la fonction `collect\_data` utilise la bibliothèque `random` de Python pour générer une séquence de chiffres aléatoires. La fonction `uniformity\_test` effectue ensuite un test de distribution uniforme sur la séquence générée.

Adaptez ce code en fonction de vos besoins spécifiques et des caractéristiques de votre générateur de nombres aléatoires. Ajoutez d'autres tests en fonction des critères de qualité que vous souhaitez évaluer. Notez également que des tests plus avancés, tels que ceux basés sur les normes du NIST, peuvent être nécessaires pour une évaluation complète de la qualité d'un RNG.

La norme AIS31 (Algorithm for Random Number Generator Testing and Test Suite) propose un ensemble de tests pour évaluer la qualité d'un générateur de nombres aléatoires (RNG), y compris des tests en ligne. Le jitter, ou fluctuation temporelle, peut être une composante de l'entropie dans le cas d'une source d'entropie basée sur des caractéristiques temporelles, comme un jitter d'horloge.

Voici comment vous pourriez adapter le test de l'entropie basée sur le jitter pour un RNG avec une source d'entropie caractérisée par le jitter :

1. \*\*Collecte de données (entropie) :\*\*

- Utilisez la source d'entropie basée sur le jitter pour générer une séquence de bits.

- Mesurez et enregistrez le jitter associé à chaque bit généré.

2. \*\*Test de l'entropie en ligne :\*\*

- Divisez la séquence de bits en blocs de bits consécutifs.

- Calculez la différence entre les mesures de jitter associées à chaque bit du bloc.

- Appliquez un test statistique pour évaluer la variabilité des différences de jitter.

Voici un exemple simplifié de la mise en œuvre de ces étapes en Python :

```python

def collect\_jitter\_entropy(rng, block\_size=8):

# Simule la collecte de données à partir d'une source d'entropie basée sur le jitter

entropy\_sequence = rng.generate\_entropy\_sequence()

jitter\_measurements = rng.measure\_jitter(entropy\_sequence)

return entropy\_sequence, jitter\_measurements

def online\_entropy\_test(entropy\_sequence, jitter\_measurements, block\_size=8):

n = len(entropy\_sequence)

passed\_tests = 0

for i in range(0, n - block\_size + 1, block\_size):

block\_jitter\_diff = [jitter\_measurements[i + j + 1] - jitter\_measurements[i + j] for j in range(block\_size - 1)]

# Appliquez un test statistique (par exemple, test de chi-carré) sur les différences de jitter

chi\_square = perform\_statistical\_test(block\_jitter\_diff)

# Seuil de confiance pour le test en ligne AIS31 (niveau de signification de 0.01)

threshold = 16.92

if chi\_square <= threshold:

passed\_tests += 1

# Retourne le pourcentage de blocs passés par le test

return (passed\_tests / ((n - block\_size) / block\_size)) \* 100

def perform\_statistical\_test(data):

# Exemple : test du chi-carré

observed\_frequencies = {diff: data.count(diff) for diff in set(data)}

expected\_frequency = len(data) / len(set(data))

chi\_square = sum((observed - expected\_frequency)\*\*2 / expected\_frequency for observed in observed\_frequencies.values())

return chi\_square

class MockRNG:

def generate\_entropy\_sequence(self):

# Simule la génération de séquence d'entropie

return "0110101110101101"

def measure\_jitter(self, entropy\_sequence):

# Simule la mesure du jitter associé à chaque bit de la séquence

return [0.1, 0.5, 0.2, 0.3, 0.4, 0.1, 0.6, 0.2]

def main():

rng = MockRNG()

entropy\_sequence, jitter\_measurements = collect\_jitter\_entropy(rng)

pass\_percentage = online\_entropy\_test(entropy\_sequence, jitter\_measurements)

print(f"Pourcentage de blocs passés par le test : {pass\_percentage}%")

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

main()

```

Notez que cet exemple est simplifié et ne représente pas un test statistique réellement significatif. Vous devrez adapter ces méthodes en fonction des spécificités de votre générateur de nombres aléatoires, de la source d'entropie basée sur le jitter, et des exigences de la norme AIS31. Utilisez des bibliothèques et des méthodes de test bien établies pour des applications réelles.

Le NIST (National Institute of Standards and Technology) propose également un ensemble de tests pour évaluer la qualité des générateurs de nombres aléatoires, y compris des tests en ligne. L'un des documents de référence pour ces tests est la publication SP800-22 du NIST, intitulée "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications". Cependant, il est important de noter que les tests en ligne peuvent être plus complexes et dépendent des caractéristiques spécifiques de la source d'entropie et du générateur de nombres aléatoires utilisé.

Voici un exemple simplifié de la manière dont vous pourriez adapter un test en ligne basé sur NIST à un générateur de nombres aléatoires avec une source d'entropie caractérisée par le jitter :

1. \*\*Collecte de données (entropie) :\*\*

- Utilisez la source d'entropie basée sur le jitter pour générer une séquence de bits.

- Mesurez et enregistrez le jitter associé à chaque bit généré.

2. \*\*Test en ligne basé sur NIST :\*\*

- Divisez la séquence de bits en blocs de bits consécutifs.

- Appliquez les tests spécifiés dans la suite de tests SP800-22 sur chaque bloc.

- Suivez les procédures et les critères spécifiques de NIST pour évaluer la qualité de la séquence.

Cependant, il est important de noter que l'implémentation complète des tests du NIST est un processus complexe et dépendant des spécificités du générateur de nombres aléatoires. La suite de tests SP800-22 inclut divers tests, tels que le test de fréquence, le test du poker, le test du runs, etc.

Voici comment vous pourriez structurer un test en ligne simple basé sur NIST en Python, utilisant la bibliothèque `NistRng` qui est une adaptation simplifiée pour illustrer le concept :

```python

import numpy as np

from scipy.stats import chisquare

from NistRng import nist\_rng

def collect\_jitter\_entropy(rng, block\_size=8):

entropy\_sequence = rng.generate\_entropy\_sequence()

jitter\_measurements = rng.measure\_jitter(entropy\_sequence)

return entropy\_sequence, jitter\_measurements

def nist\_online\_test(entropy\_sequence, block\_size=8):

n = len(entropy\_sequence)

passed\_tests = 0

for i in range(0, n - block\_size + 1, block\_size):

block = entropy\_sequence[i:i + block\_size]

# Exemple : test de fréquence avec chisquare

expected\_frequencies = np.full(2\*\*block\_size, n / 2\*\*block\_size)

observed\_frequencies, \_ = np.histogram([int(b, 2) for b in block], bins=2\*\*block\_size)

\_, p\_value = chisquare(observed\_frequencies, f\_exp=expected\_frequencies)

# Seuil de confiance pour le test (niveau de signification de 0.01)

threshold = 0.01

if p\_value > threshold:

passed\_tests += 1

return (passed\_tests / ((n - block\_size) / block\_size)) \* 100

class MockRNG:

def generate\_entropy\_sequence(self):

return "0110101110101101"

def measure\_jitter(self, entropy\_sequence):

return [0.1, 0.5, 0.2, 0.3, 0.4, 0.1, 0.6, 0.2]

def main():

rng = MockRNG()

entropy\_sequence, \_ = collect\_jitter\_entropy(rng)

pass\_percentage = nist\_online\_test(entropy\_sequence)

print(f"Pourcentage de blocs passés par le test NIST : {pass\_percentage}%")

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

main()

```

Dans cet exemple, la fonction `nist\_online\_test` applique le test de fréquence (chi-carré) du NIST sur des blocs successifs de la séquence générée. Notez que cela est un exemple simplifié et qu'une mise en œuvre complète nécessiterait une compréhension approfondie des spécifications de la suite de tests SP800-22 et des caractéristiques spécifiques du générateur de nombres aléatoires. Utilisez des bibliothèques et des méthodes de test bien établies pour des applications réelles.