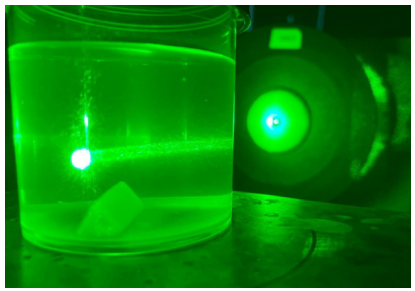
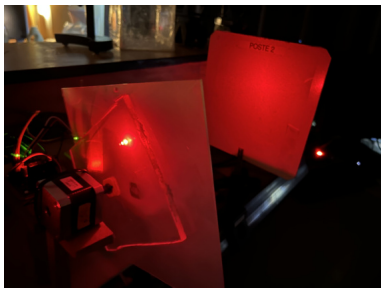
Remarques sur mon TIPE:

- Note 16
- Le titre ainsi que la conclusion du MCOT ne sont plus totalement cohérents avec le contenu final de mon TIPE. En effet, j'ai dû changer d'objectif en cours de réalisation, après la fermeture du module de modification du MCOT, afin de ne pas dépasser le temps imparti. La partie initialement prévue — que j'ai finalement supprimée — est néanmoins conservée à la fin des annexes.

Génération de clés d'accès numériques sécurisés à partir d'un phénomène aléatoire.

Eliott Fevret - N°12112

Conversion, transformation, transition

Utilisation des clés d'accès

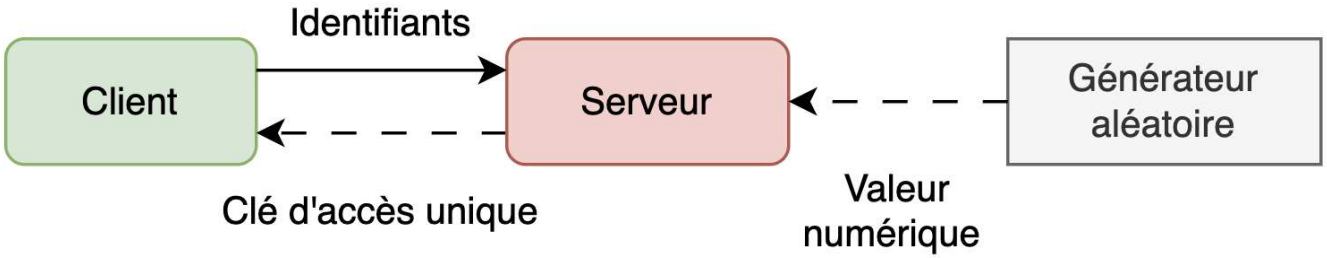


Figure 1.1 – Procédure de connection

Problématique

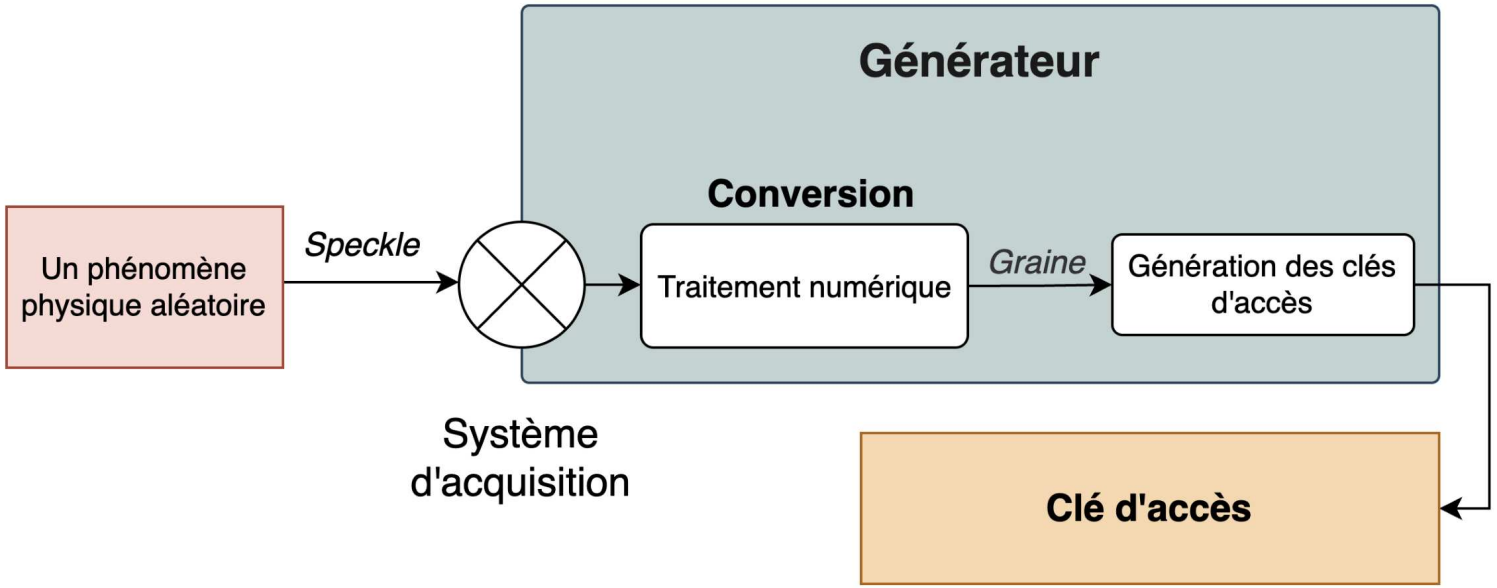


Figure 1.2 – Schéma fonctionnel du générateur de clé d'accès

Problématique

Comment combiner un phénomène aléatoire et le déterminisme des ordinateurs pour générer des clés d'accès sécurisées ?

Le phénomène de *speckle*

Définition

Le *speckle* est une structure **granulaire** résultant de la diffusion aléatoire de lumière **cohérente**. Il provient des **interférences** entre des ondes lumineuses cohérentes.

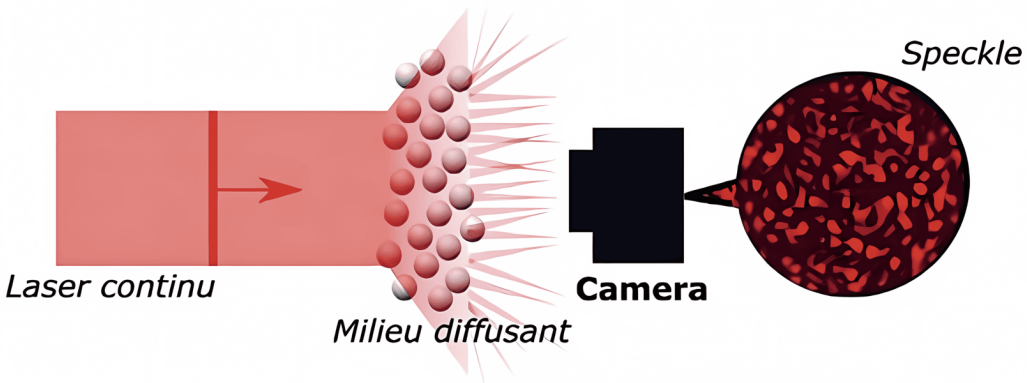


Figure 2.1 – Schéma explicatif

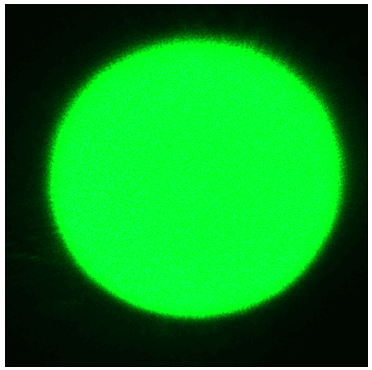


Figure 2.2 – Attente

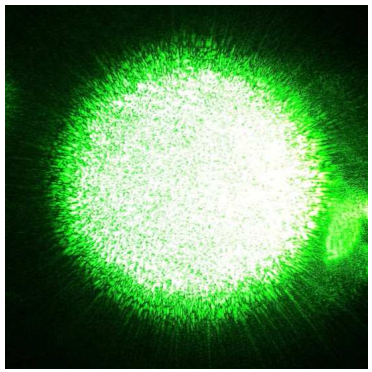
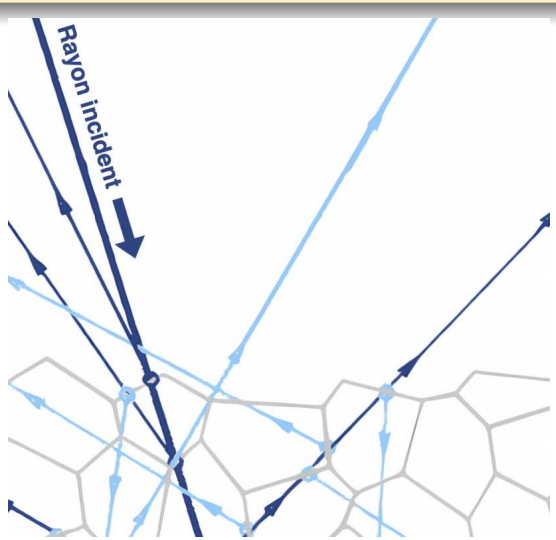


Figure 2.3 – Granularité du *speckle*

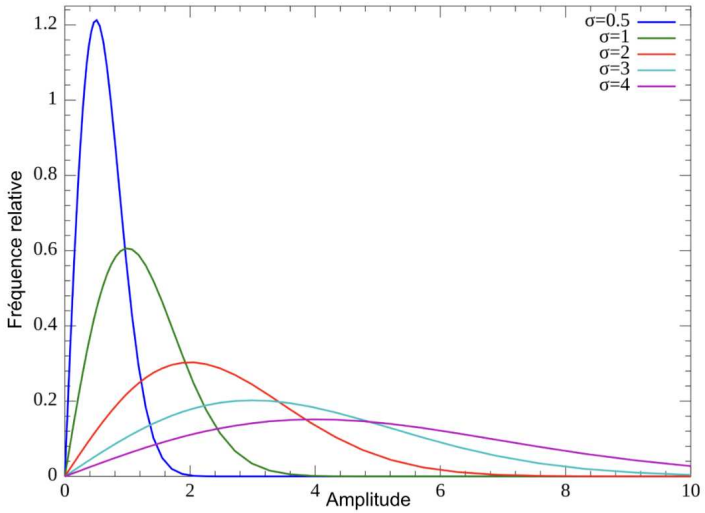
Le phénomène de *speckle*

Caractéristiques

- Le *speckle* apparaît quand une lumière cohérente éclaire une **surface rugueuse** à l'échelle de la longueur d'onde. Chaque point du motif résulte d'une **interférence aléatoire** entre des milliers d'ondes déphasées.
- **L'amplitude suit une distribution de *Rayleigh*.**

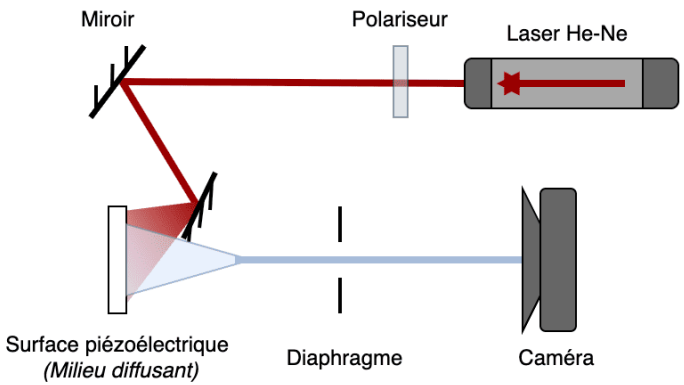


(a) Surface diffusante

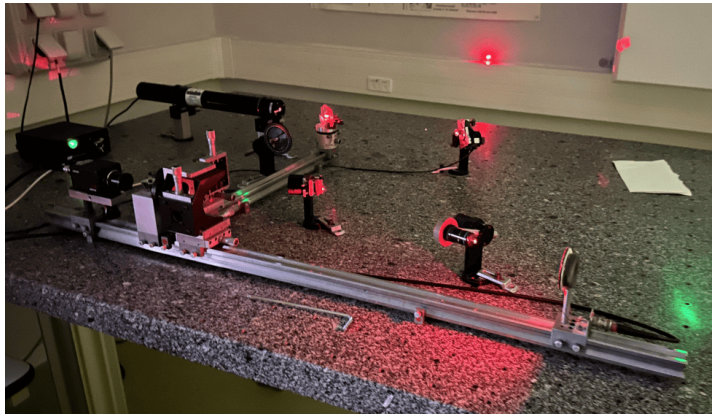


(b) Loi de Rayleigh

Expérience à l'école *SupOptique*

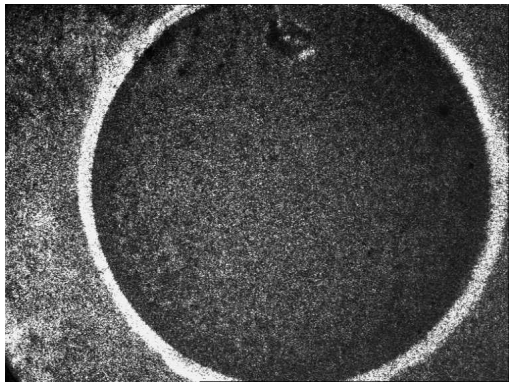


(a) Protocole



(b) Banc optique

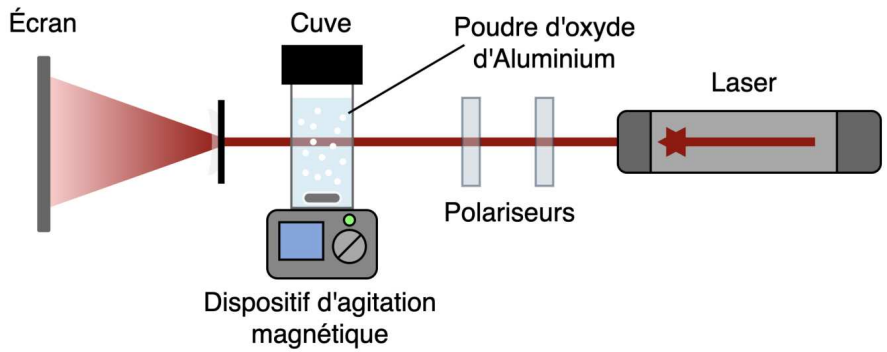
(c) Speckle obtenu



(d) Résultat final

Figure 3.1 – Réalisation de l'expérience n°1

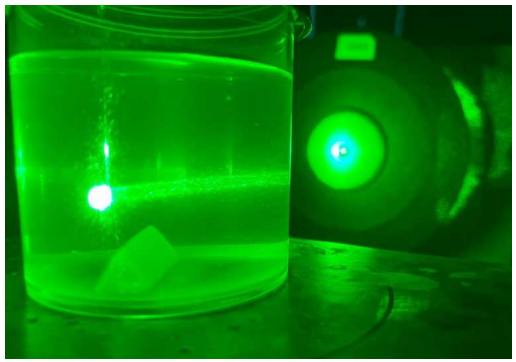
Production d'un *speckle* non stationnaire



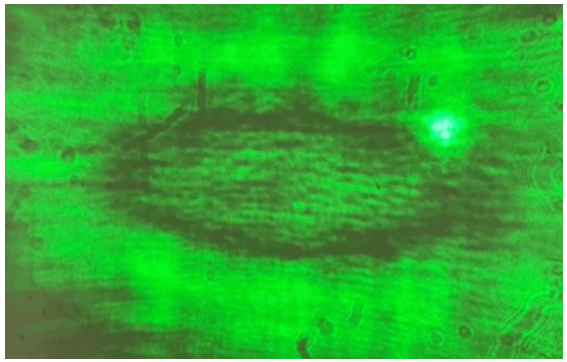
(a) Protocole

(b) Banc optique

(c) Manipulation



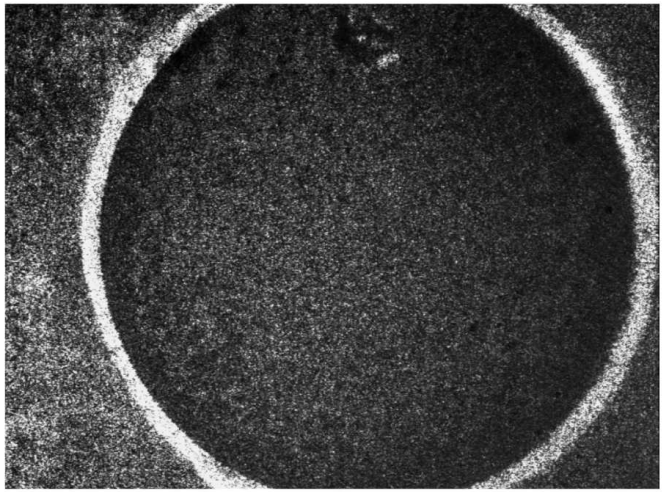
(d) Milieu diffusant

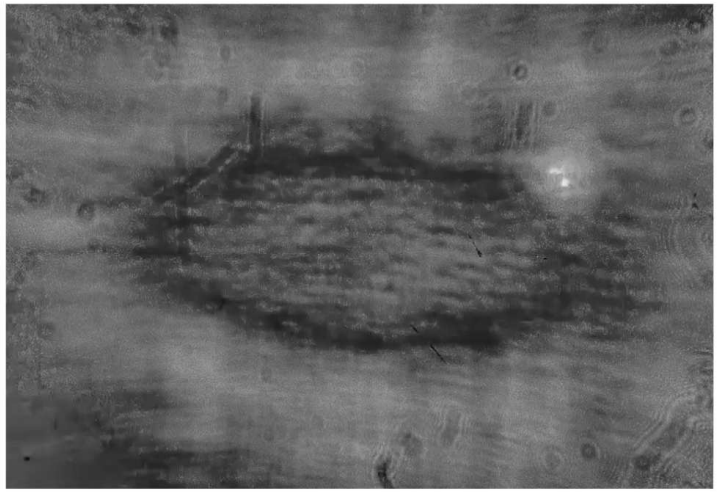


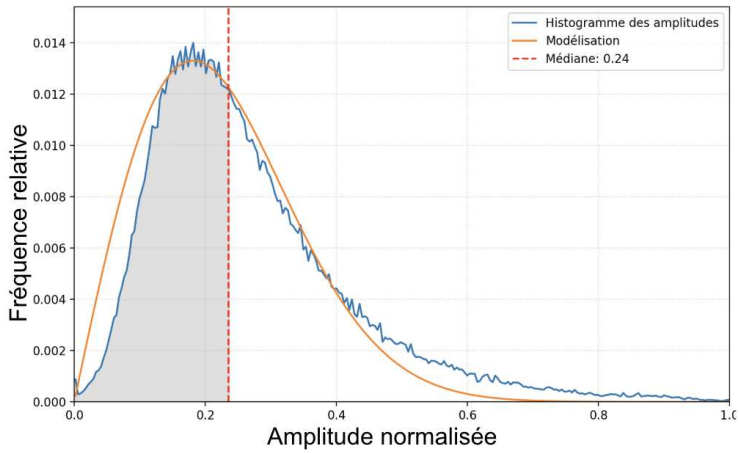
(e) Figure obtenue

Figure 3.2 – Réalisation de l'expérience n°2

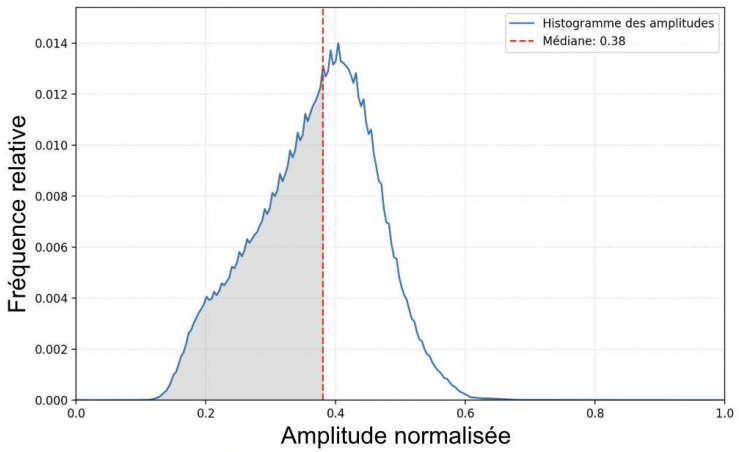
Caractérisation du Phénomène de Speckle







(a) Speckle obtenu à *Supoptique*



(b) Figure obtenue

Figure 3.3 – Comparaison des distributions d'amplitude

Production d'un *speckle* non stationnaire

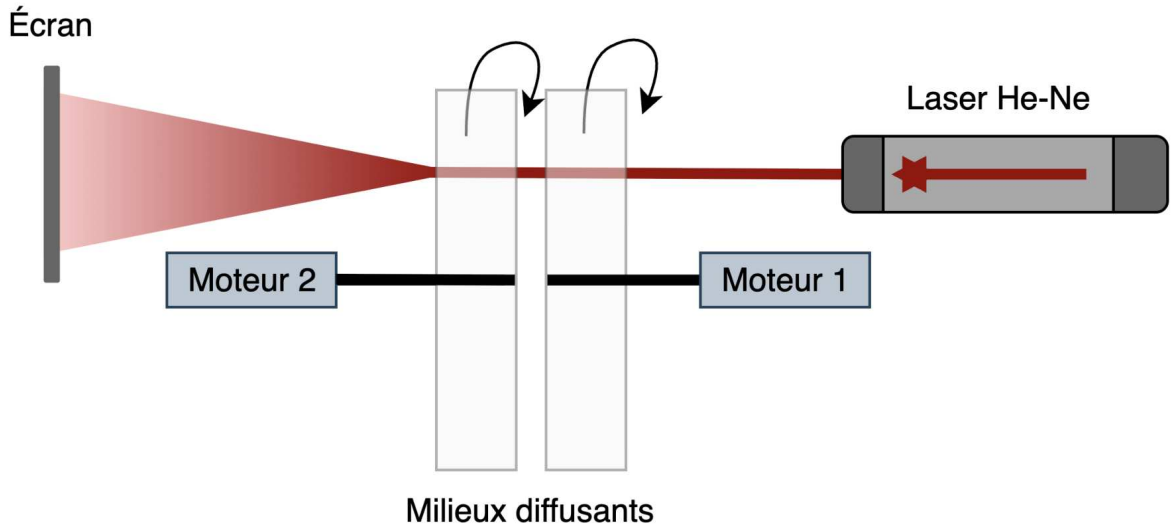
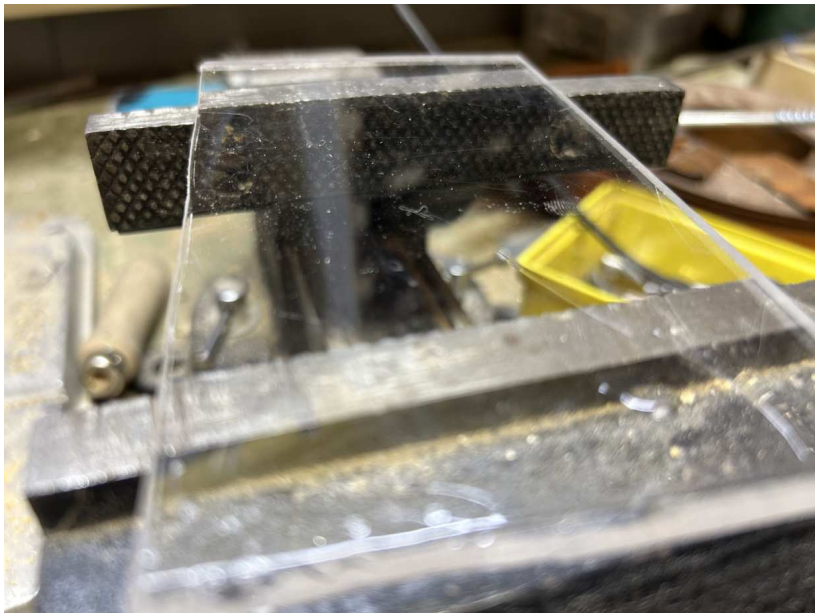


Figure 3.4 – Schéma du protocole expérimental utilisé

Conception du milieu diffusant



(a) Plexiglass non poli

(b) Plexiglass poli

Figure 3.5 – Fabrication et motorisation du milieu diffusant

Pilotage du système avec Arduino

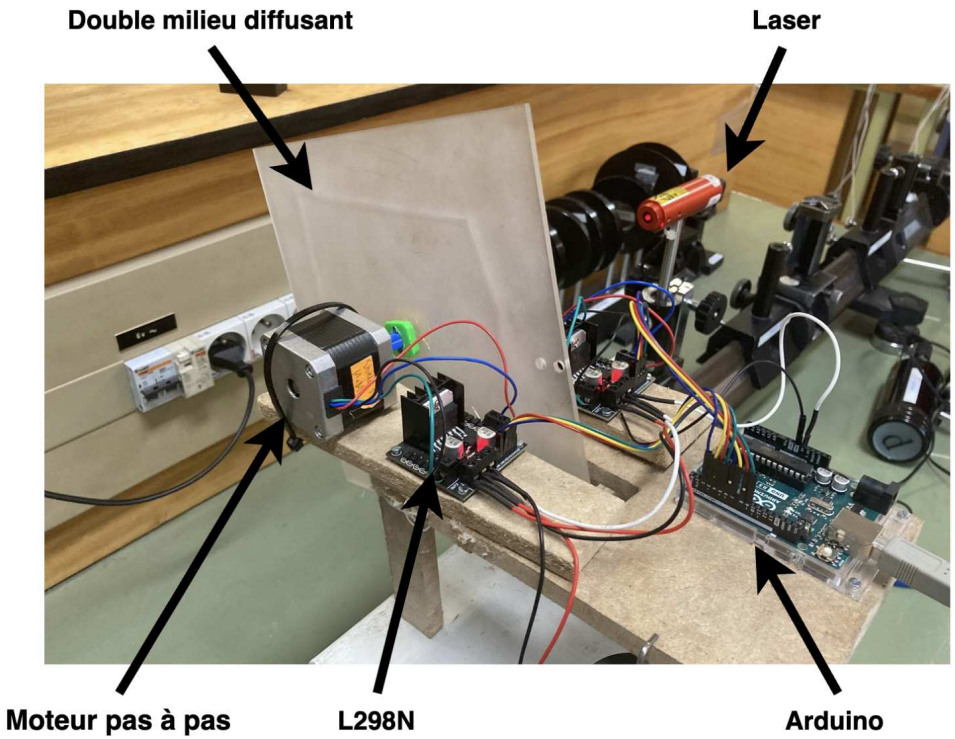
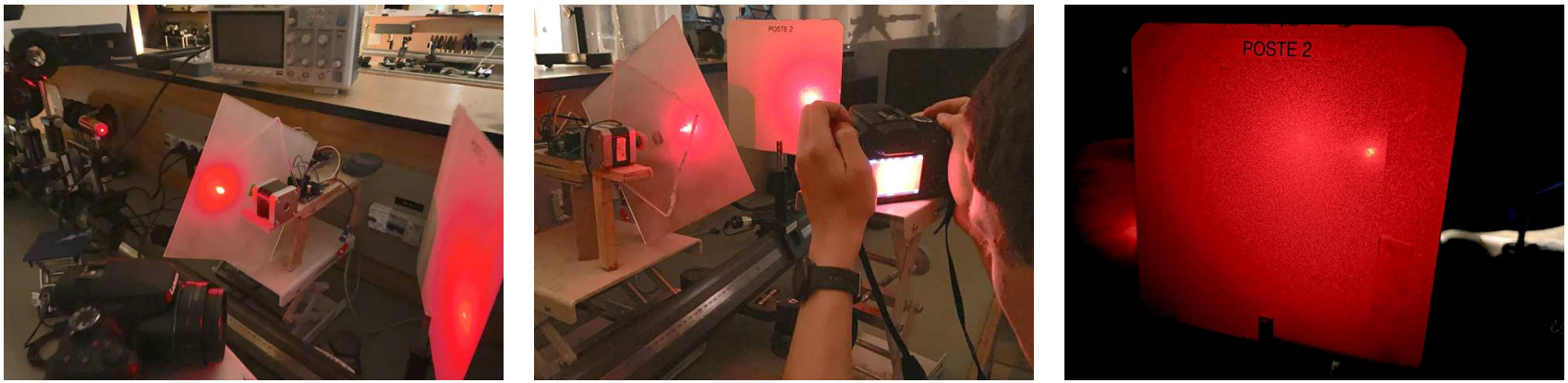


Figure 3.6 – Montage électronique

Remarque

Les moteurs tournent de 0,8°/pas. D'où $\left(\frac{360}{0{,}8}\right)^2 = 202\,500$ combinaisons possibles.

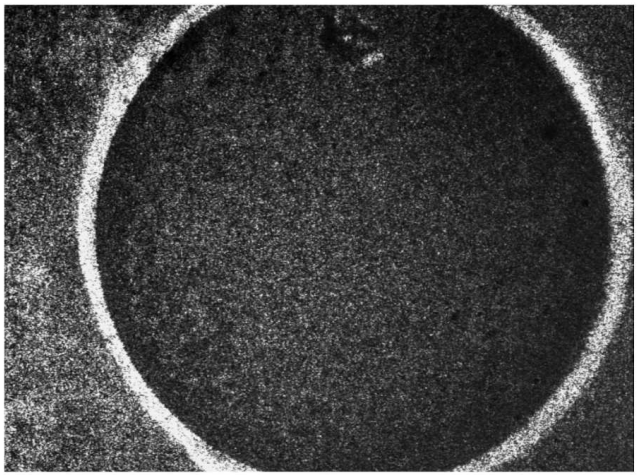
Expérience

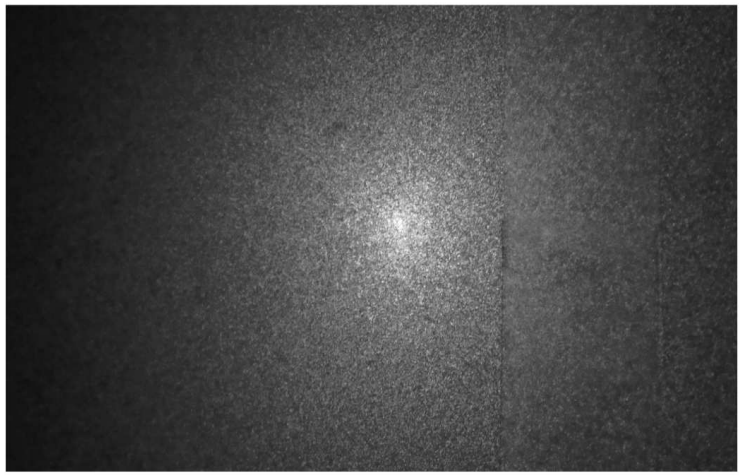


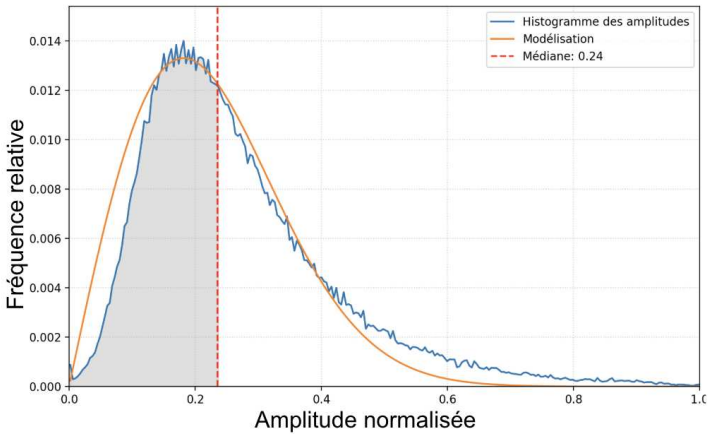
(a) Banc optique (b) Capture vidéo (c) Speckle obtenu

Figure 3.7 – Réalisation de l'expérience

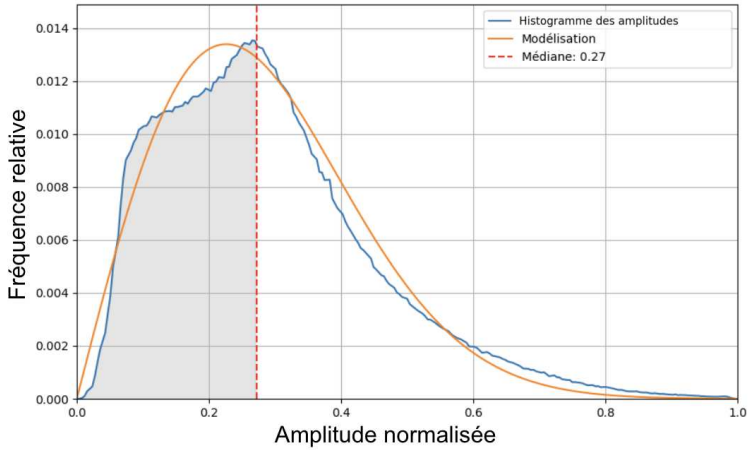
Caractérisation du Phénomène de Speckle







(a) Speckle obtenu à *Supoptique*



(b) Figure obtenue

Figure 3.8 – Comparaison des distributions d'amplitude

Numérisation

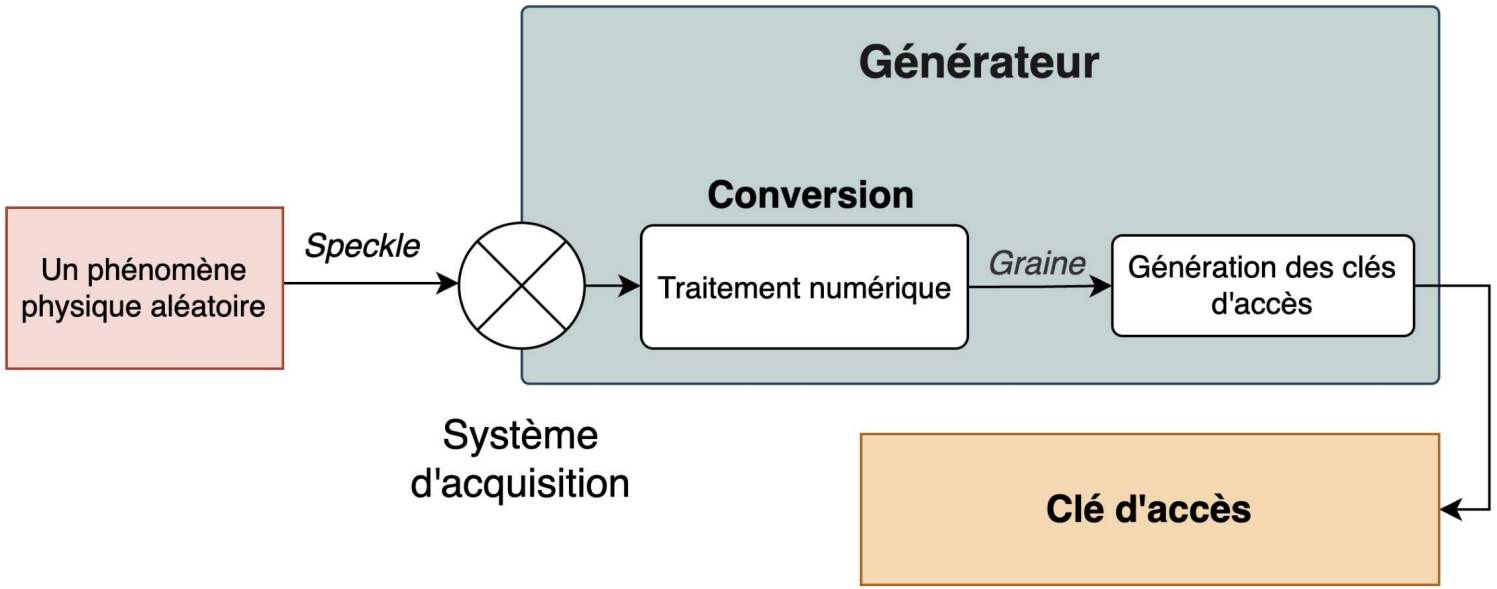
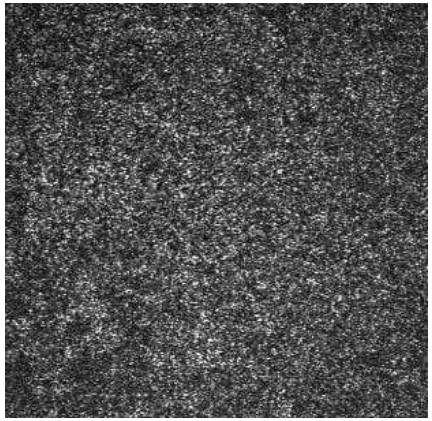


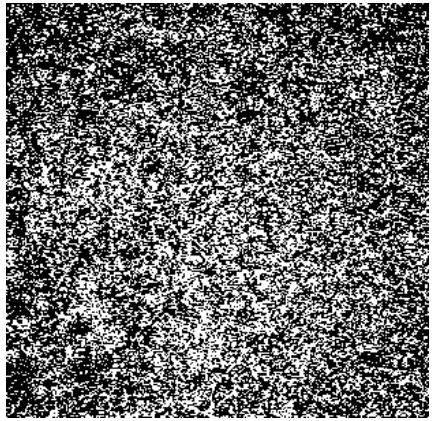
Figure 4.1 – Schéma fonctionnel du générateur de clé d'accès

Normalisation du Speckle

(a) Speckle expérimental

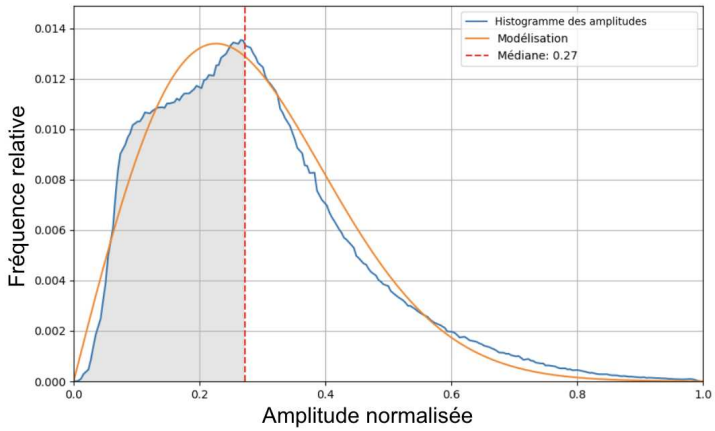


(b) Speckle rogné



(c) Speckle Binaire

Figure 4.2 – Traitement informatique du Speckle initial



Fonction de hachage

Définition

Fonction de chiffrement déterministe et uniforme qui transforme une entrée numérique en une signature, dont **un changement infime** en entrée créé une **variation drastique** de la sortie.

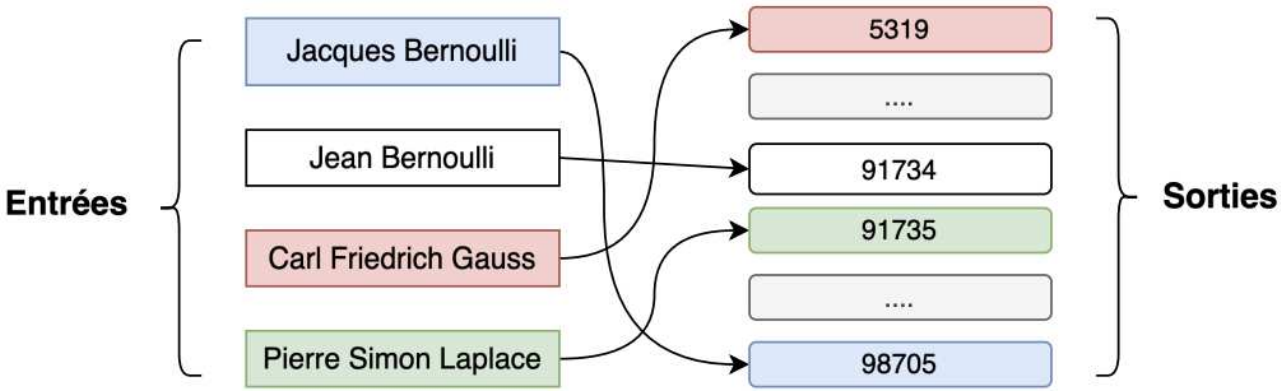
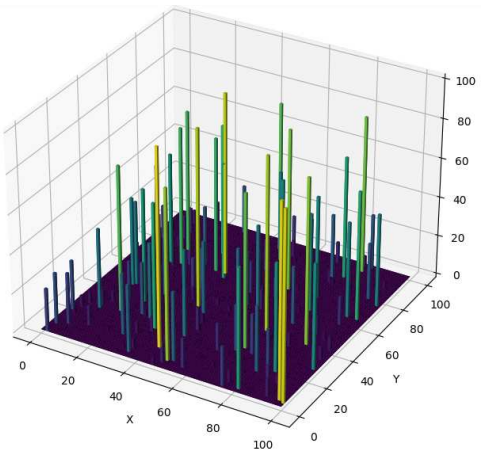


Figure 4.3 – Illustration d'une fonction de hachage

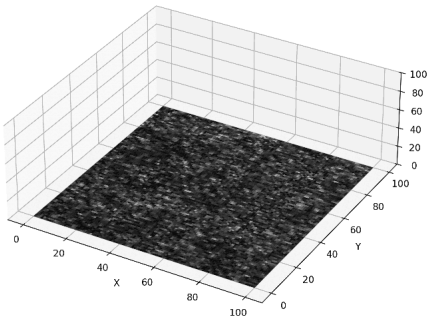
Une 1ère fonction de hachage intuitive

En notant G la clé d'accès liée à l'image :

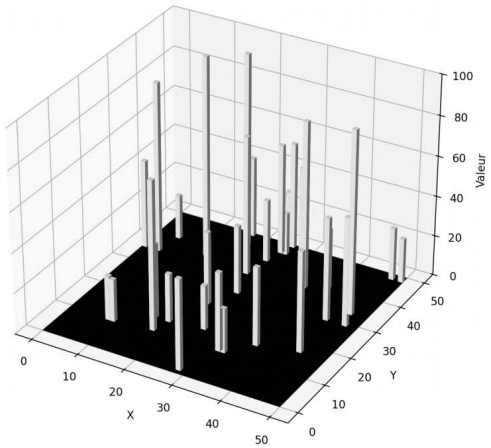
$$G = \sum_{(x,y,\text{poids})} \text{intensite}_{x,y} \times \text{poids}_{x,y}$$



(a) Carte des poids



(b) Speckle



(c) Résultat final

Figure 4.4 – Exemple d'une première fonction de hachage

Algorithme de *SHA-2*

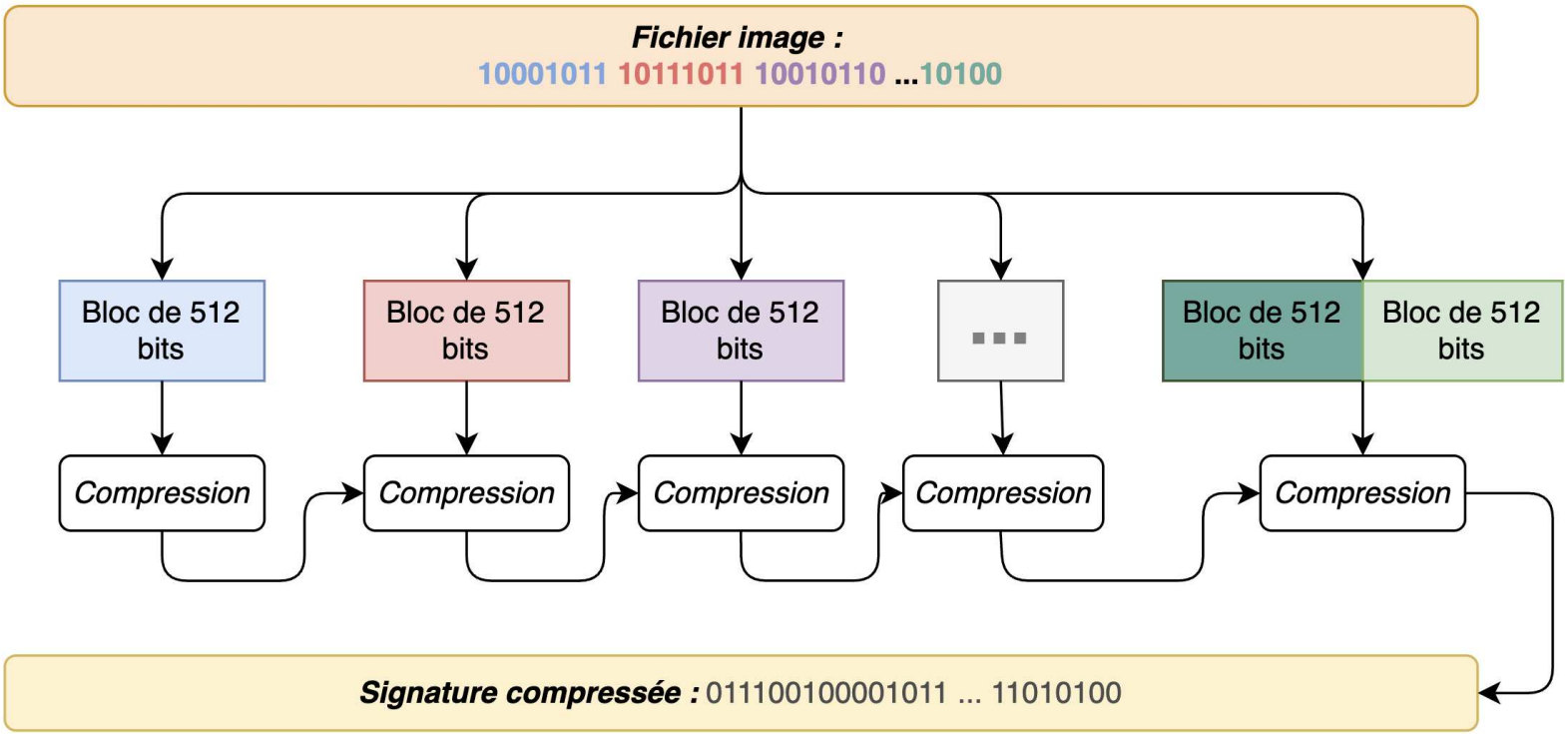


Figure 4.5 – Schéma de principe du SHA-2

Génération finale des clés d'accès

Indice i	**Clé d'accès** G_i
1	345615778932156701293847...18
2	847261982736487263487126...93
3	127346127346127346127346...05
4	913847265938472659384726...44
5	732465982374659827346598...23
6	182736459812736459812736...12
7	598273465982374659823746...59
8	128374659827346598273465...06
9	762348917263498172634981...77
10	394857123984751239847512...38

Table 1 – Les premières clés d'accès générées.

Vérification fréquentielle du générateur

Intérêt

On vérifie que les clés d'accès sont bien générées de façon uniforme.

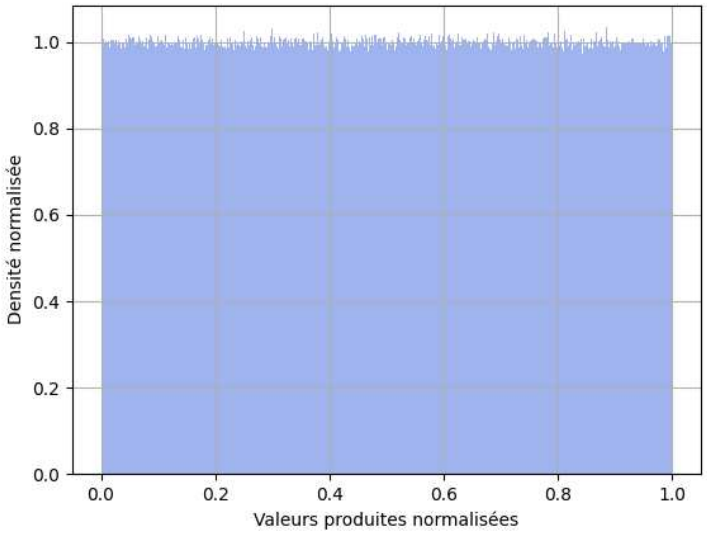


Figure 4.6 – Histogramme normalisé des clés générées

Récapitulatif et conclusion

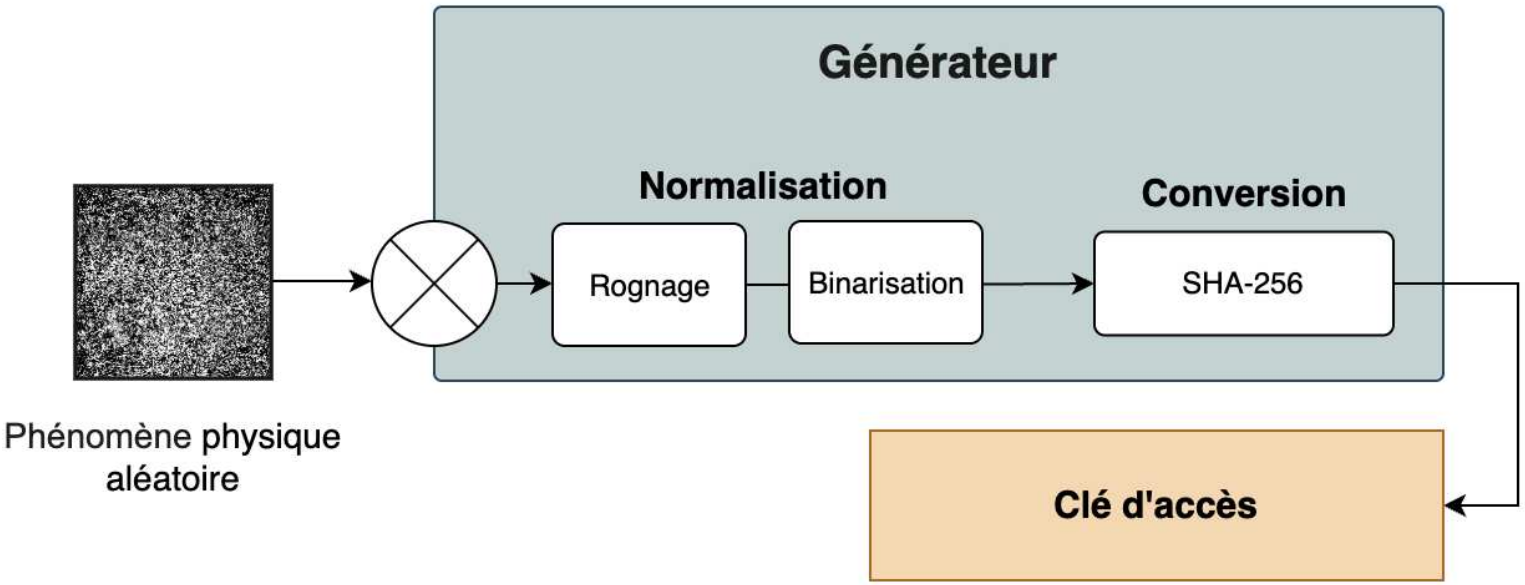


Figure 4.7 – Récapitulatif du TIPE

Merci pour votre attention.

Références bibliographiques

[1] Damien Vergnaud : L'aléatoire, clé de voûte de la sécurité informatique. *La Recherche, Juillet-Août 2019, n°549*

[2] Jean Taboury : Utilisation du 'Speckle' comme Générateur Rapide de Tableaux Aléatoires Binaires : Optimisation des Paramètres. *Annales des Télécommunications, 1988*

[3] Jean Taboury : Transmission désordonnée et speckle. *Photoniques, 2011*

[4] Felipe A. Louza : True random number generator based on laser speckle images from particles under Brownian motion. SSRN

[5] Communautaire : Secure Hash Algorithms 2. Wikipedia

[6] Donald Ervin Knuth : The Art of Computer Programming, *Chapitre 3*

[7] Benjamin Billet : Nombres pseudo-aléatoires. https://benjaminbillet.fr/media/prng$_m$*athreport.pdf*

[8] Philippe Lalanne : Les réseaux de neurones formels et leurs réalisations optoélectroniques. Génération optique de tableaux de nombres aléatoires, 1989. https://pastel.hal.science/pastel-00730634v1

Rappel sur la loi de Rayleigh

Rappel (*Source :* https://fr.wikipedia.org/wiki/Loi-de-Rayleigh)

Cette loi est définie par :

$$f(x,\sigma) = \frac{x}{\sigma^2}\exp\left(-\frac{x^2}{2\sigma^2}\right)$$

Dans le cadre d'une marche aléatoire symétrique, la distance D_n à laquelle une particule se trouve de son point de départ, après n pas, suit une loi de Rayleigh de paramètre $\sqrt{n}$.

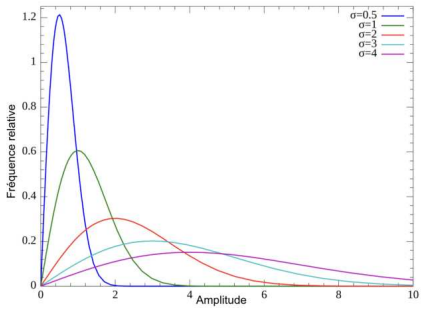


Figure 6.1 – Distribution de Rayleigh

Histogramme des amplitudes normalisée d'un Speckle

```
1  image_path = './speckle1_rognée.jpg'
2  image = Image.open(image_path).convert('L')
3
4  # Convertir l'image en tableau numpy
5  image_array = np.array(image)
6
7  # Calculer l'histogramme des amplitudes
8  hist, bins = np.histogram(np.sqrt(image_array.flatten()), bins=256, range=[0, 256])
9  hist = hist / hist.sum()
10
11 median_intensity = np.median(image_array) # Calculer la médiane
12 normalized_bins = bins[:-1] / 255
13
14 plt.figure(figsize=(10, 6))
15 plt.plot(normalized_bins, hist, label='Histogramme des amplitudes')
16
17 plt.fill_between(normalized_bins, hist, where=(normalized_bins <=
18             median_intensity / 255), color='silver', alpha=0.4)
19
20 plt.axvline(x=median_intensity / 255, color='red', linestyle='dashed',
21         label=f'Médiane: {median_intensity:.2f}')
22
23 plt.xlabel('Amplitudes normalisée (0 à 1)')
24 plt.ylabel('Fréquence relative')
25 plt.xlim([0, 1])
26 plt.grid()
27 plt.legend()
28 plt.show()
```

Montage Arduino

```
#include <Stepper.h>

const int NbPasParTour = 450; // 0.8  /pas => 450 pas/tour

// Moteurs branchés sur L298N
Stepper moteur1(NbPasParTour, 2, 3, 4, 5);
Stepper moteur2(NbPasParTour, 6, 7, 8, 9);

// Vitesse moteur 1 : 1 pas par seconde => 1 ms entre chaque pas
const unsigned long delaiMoteur1 = 1000;
unsigned long dernierPas1 = 0;
int compteurPasMoteur1 = 0;

void setup() {
  Serial.begin(9600);
}

void loop() {
  unsigned long maintenant = millis();
  // Moteur 1 : avance de 1 pas toutes les 1000 ms
  if (maintenant - dernierPas1 >= delaiMoteur1) {
    moteur1.step(1);
    dernierPas1 = maintenant;
    compteurPasMoteur1++;
    // Quand moteur 1 a fait un tour complet (450 pas), moteur 2 avance  d un  pas
    if (compteurPasMoteur1 >= NbPasParTour) {
      moteur2.step(1);
      compteurPasMoteur1 = 0;
    }
  }
}
```

Montage Arduino

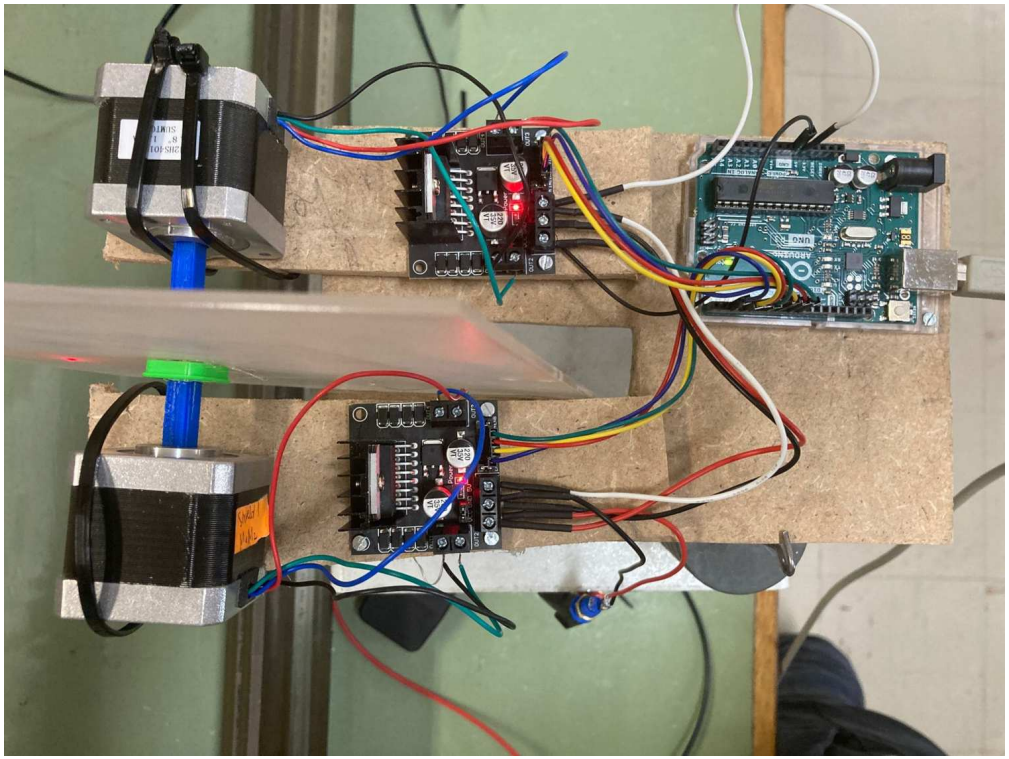
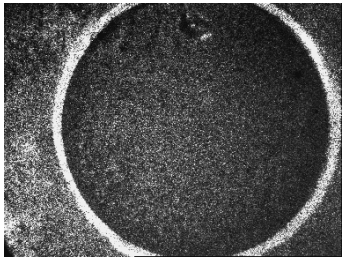


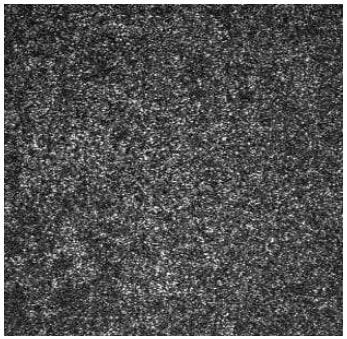
Figure 6.2 – Montage Arduino

Traitement informatique : Rognage de l'image

```python
from PIL import Image

def rogner_image(image_path, output_path_rognee):
    image = Image.open(image_path)

    largeur, hauteur = image.size
    if largeur < 200 or hauteur < 200:
        print("L'image est trop petite.")
        return None

    # Définition de la zone de rognage pour 100x100 pixels
    left = (largeur - 100) // 2
    top = (hauteur - 100) // 2
    right = left + 100
    bottom = top + 100

    image_rognee = image.crop((left, top, right, bottom))

    image_rognee.save(output_path_rognee)
    print("L'image rognée a été sauvegardée")

    return image_rognee
```



(a) Speckle capturé expérimentalement



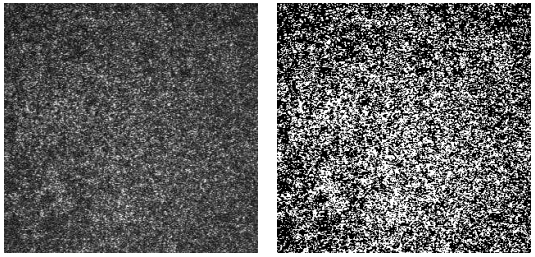
(b) Speckle rogné

Extraction des images

```python
import cv2

def capture_video_et_generer(video_source):
    cap = cv2.VideoCapture(video_source)
    if not cap.isOpened():
        raise IOError(f"Impossible d'ouvrir la source video : {video_source}")

    frame_count = 0

    while True:
        ret, frame = cap.read()
        if not ret:
            print("Fin de la video ou erreur de lecture.")
            break

        frame_count += 1
        if frame_count % 10 == 0:
            cle_d_acces = generateur(frame)
            print(f"Clé générée: {cle_d_acces}")

capture_video_et_generer(video_source='capture_speckle.mp4')
```

Traitement informatique : Binarisation du Speckle

```python
from PIL import Image
import numpy as np

def binariser_image(image_rognee, output_path_binaire):
    # Convertir l'image en niveaux de gris
    image_rognee_gris = image_rognee.convert("L")

    image_np = np.array(image_rognee_gris)
    intensite_mediane = np.median(image_np)

    # Binariser l'image : 255 si supérieur à la médiane, 0 sinon
    image_binaire_np = np.where(image_np > intensite_mediane, 255, 0)

    # Créer l'image binaire à partir du tableau numpy
    image_binaire = Image.fromarray(np.uint8(image_binaire_np)).convert("1")

    image_binaire.save(output_path_binaire, format="PNG")
    print("L'image binaire a été sauvegardée.")

    return image_binaire
```



(a) Speckle rogné (b) Speckle binaire

Génération de la carte des poids

```python
def generer_carte_poids(taille_cible, seuil=10):
    hauteur, largeur = taille_cible
    # Les poids suivent une loi exponentielle
    poids_flottants = np.random.exponential(scale=20, size=(hauteur, largeur))
    poids = np.clip(poids_flottants, 0, 100).astype(int)
    poids[poids < seuil] = 0  # Filtrage des poids faibles
    return poids

def afficher_carte_poids_3D(carte_poids, seuil):
    # Affiche la carte des poids sous forme de barres 3D, avec une échelle de couleurs
    hauteur, largeur = carte_poids.shape
    X, Y = np.meshgrid(np.arange(largeur), np.arange(hauteur))
    figure = plt.figure(figsize=(8, 8))
    axe = figure.add_subplot(111, projection='3d')
    poids_normalises = carte_poids / 100.0  # Normalisation pour l'affichage
    palette_couleur = plt.cm.viridis
    couleurs = palette_couleur(poids_normalises.flatten())
    axe.bar3d(X.flatten(), Y.flatten(), np.zeros_like(poids_normalises).flatten(),
              1, 1, poids_normalises.flatten(), color=couleurs)
    axe.set_xlabel('X')
    axe.set_ylabel('Y')
    axe.set_zlim(0, 100)
    plt.show()

taille_carte = (100, 100)
seuil_poids = 10
carte_generee = generer_carte_poids(taille_carte, seuil_poids)

afficher_carte_poids_3D(carte_generee, seuil_poids)
```

Produit scalaire entre le *speckle* et la carte de poids

```python
import numpy as np
from PIL import Image

def binarize_and_resize(img_path, w_map, seuil_poids=0, size=(50, 50), seuil_bin=128):
    img = Image.open(img_path).convert('L').resize(size)
    bin_img = (np.array(img) > seuil_bin).astype(int)
    poids = np.array(Image.fromarray(w_map.astype(np.float32))
        .resize(size, Image.Resampling.BILINEAR)).astype(int)
    poids[poids < seuil_poids] = 0
    return bin_img, poids

def calc_signature(bin_img, poids):
    return np.sum(bin_img * poids)

img_path = './speckle_rogne_binaire.jpg'
taille_carte = (100, 100)
seuil_poids = 10

carte_generee = generer_carte_poids(taille_carte, seuil_poids)
binaire, poids_resized = binarize_and_resize(img_path, poids)
signature = calc_signature(binaire, poids_resized)
print(f"Signature: {signature}")
```

Codage du SHA-2

```python
import struct
# Ces constantes sont les 32 premiers bits fractionnaires des racines carrées des
# 8 premiers nombres premiers
CONSTANTES_K = [0x428a2f98, 0x71374491, ..., 0xc67178f2]

# Valeurs de hachage initiales
VALEURS_H_INITIALES = [0x6a09e667, 0xbb67ae85, ..., 0x5be0cd19]

def rotation_droite(n, b):
    """Effectue une rotation circulaire à droite de 'n' de 'b' bits."""
    return ((n >> b) | (n << (32 - b))) & 0xFFFFFFFF

# Prétraitement du message
def sha256_bourrage(message_octets):
    """
    Applique le bourrage au message pour que sa longueur soit un multiple de 512 bits.
    """
    # Calcul de la longueur originale du message en bits
    longueur_orig_bits = (8 * len(message_octets)) & 0xffffffffffffffff

    # Ajoute un bit '1' (représenté par 0x80 en octet)
    message_octets.append(0x80)

    # Ajoute des bits '0' jusqu'à ce que la longueur soit 56 octets modulo 64
    while len(message_octets) % 64 != 56:
        message_octets.append(0)

    # Ajoute la longueur originale du message (en bits) sous forme de 8 octets (64 bits)
    message_octets += struct.pack('>Q', longueur_orig_bits)
    return message_octets
```

```
1  def sha256_hachage(tableau_binaire_entree):
2      tableau_bits = (tableau_binaire_entree // 255).astype(np.uint8) # Convertir des vale
3
4      # Convertir le tableau de bits en un tableau d'octets, chaque 8 bits du tableau_bits
5      longueur_bits = len(tableau_bits)
6      bits_restants = longueur_bits % 8
7      if bits_restants != 0:
8          tableau_bits = np.append(tableau_bits, np.zeros(8 - bits_restants, dtype=np.uint
9          longueur_bits = len(tableau_bits)
10
11     message_en_octets = bytearray()
12     for i in range(0, longueur_bits, 8):
13         octet_valeur = 0
14         for k in range(8):
15             octet_valeur |= (tableau_bits[i + k] << (7 - k))
16         message_en_octets.append(octet_valeur)
17
18     message_bourre = sha256_bourrage(message_en_octets)
19
20     # Copier les valeurs initiales H pour ce calcul de hachage
21     registres_h = list(VALEURS_H_INITIALES)
22
23     # Traiter le message par blocs de 64 octets (512 bits)
24     for i in range(0, len(message_bourre), 64):
25         mots_w = [0] * 64
26         bloc_actuel = message_bourre[i:i + 64]
27
28         # Initialiser les 16 premiers mots du bloc
29         for j in range(16):
30             mots_w[j] = struct.unpack('>I', bloc_actuel[j * 4:(j + 1) * 4])[0]
```

```
    for j in range(16, 64):
        s0 = rotation_droite(mots_w[j - 15], 7) ^
             rotation_droite(mots_w[j - 15], 18) ^ \
             (mots_w[j - 15] >> 3)
        s1 = rotation_droite(mots_w[j - 2], 17) ^ \
             rotation_droite(mots_w[j - 2], 19) ^ \
             (mots_w[j - 2] >> 10)
        mots_w[j] = (mots_w[j - 16] + s0 + mots_w[j - 7] + s1) & 0xFFFFFFFF

    a, b, c, d, e, f, g, h = registres_h

    # Boucle principale de compression (64 itérations)
    for j in range(64):
        S1 = rotation_droite(e, 6) ^ rotation_droite(e, 11) ^ rotation_droite(e, 25)
        ch = (e & f) ^ ((~e) & g)
        temp1 = (h + S1 + ch + CONSTANTES_K[j] + mots_w[j]) & 0xFFFFFFFF
        S0 = rotation_droite(a, 2) ^ rotation_droite(a, 13) ^ rotation_droite(a, 22)
        maj = (a & b) ^ (a & c) ^ (b & c)
        temp2 = (S0 + maj) & 0xFFFFFFFF
        h, g, f, e, d, c, b, a = g, f, e, (d + temp1) & 0xFFFFFFFF, c, b, a, (temp1 + te

    # Ajouter le résultat de ce bloc aux registres de hachage H
    registres_h[0] = (registres_h[0] + a) & 0xFFFFFFFF
    registres_h[1] = (registres_h[1] + b) & 0xFFFFFFFF
    ...
    registres_h[7] = (registres_h[7] + h) & 0xFFFFFFFF

# Produire le hachage final en concaténant les valeurs H
hash_hex = ''.join(f'{x:08x}' for x in registres_h)
hash_binaire = bin(int(hash_hex, 16))[2:].zfill(256)

return hash_binaire
```

Génération des clés d'accès

```python
def image_to_sha256_binary(image_path):
    """Convertir une image en niveaux de gris, binariser par médiane,
    et la passer dans SHA-256 (résultat en binaire)."""
    image = Image.open(image_path).convert('L')
    image_array = np.array(image)

    median_value = np.median(image_array)
    binary_array = np.where(image_array > median_value, 255, 0)

    return sha256_hachage(binary_array)  # Retourne une chaîne binaire

def generate_access_key(image_path):
    """Générer une clé d'accès en base 10 à partir de l'image binarisée."""
    sha256_binary = image_to_sha256_binary(image_path)  # Résultat binaire

    access_key = int(sha256_binary, 2)

    print(f"Clé d'accès générée (base 10) : {access_key}")
    return access_key
```

Générateurs de nombre pseudo-aléatoires

But : Générer, à partir de notre graine, des nombres déterministes mais indiscernables de vrais nombres aléatoires.

Générateur pseudo-aléatoire

Générateur capable de produire une suite de nombres aléatoires à partir d'une expression mathématique. Bien qu'ils ne soient pas intrinsèquement aléatoires, leur conception rend la prédiction des nombres pratiquement impossible sans connaître la graine d'entrée, d'où le terme **pseudo**-aléatoire.

Blum Blum Shub (BBS)

Définition : Le générateur BBS [1] est défini par la relation de récurrence :

$$\boxed{X_{k+1} = X_k^2 \mod N}$$

Paramètres :

- $N = p \times q$, où p et q sont deux nombres premiers congrus à 3 modulo 4.
- $X_0 = G_i$ tel que $\text{PGCD}(X_0, N) = 1$.

Critères de sécurité

p et q doivent être de grande taille pour empêcher la factorisation de n.

1. Proposé en 1986 par Lenore Blum, Manuel Blum et Michael Shub.

Implémentation de BBS

On prend les constantes suivantes :

- $p = 666641161166556834 2 \ldots 0543340193765623694 07$
- $q = 5102067086522710460 5 \ldots 2464452847547897870 7$
- $n = p \times q = 340124792696050 \ldots 8506875226646939898 69$

Soit $(X_i)_{i \in \mathbb{N}^*}$ la suite des valeurs générées.

$$\boxed{U_i = \frac{X_i \bmod 2^{\lfloor \log_2(\log_2 n) \rfloor}}{2^{\lfloor \log_2(\log_2 n) \rfloor}}} \quad \in [0, 1]$$

i	U_i
1	0,0078125
2	0,765625
3	0,21875
$\ldots$	etc

Vérification statistique

La suite de valeurs aléatoires $(U_i)_{i\in\mathbb{N}^*}$ doit vérifier :

- **Espérance** : $\mu = \frac{1}{n}\sum_{i=1}^{n} U_i \underset{n\to+\infty}{\longrightarrow} \frac{1}{2}$
- **Variance** : $v = \left(\frac{1}{n}\sum_{i=1}^{n} (U_i)^2 - \mu^2\right) \underset{n\to+\infty}{\longrightarrow} \frac{1}{12}$
- **Autocorrélation** : $r_k = \frac{1}{n}\sum_{i=1}^{n} (U_i)\cdot(U_{i+k}) \underset{n\to+\infty}{\longrightarrow} \frac{1}{4}$

Après avoir effectué ces tests, nous obtenons les résultats suivants : $\mu = 0.500081419$, $v = 0.083310375$, $r_1 = 0.250061410$.

Conclusion

Pour les 20 essais on a bien des valeurs proches de celles théoriques. On peut donc considéré que notre générateur génère bien des nombres aléatoires de façon uniforme.

Résultats des tests statistiques

Essai	**Moyenne**	**Variance**	**Autocorrélation r_1 et r_2**
1	0.498312	0.082913	0.249187 ; 0.124891
2	0.501213	0.083105	0.251127 ; 0.125612
3	0.500002	0.083310	0.250431 ; 0.125022
4	0.500812	0.082995	0.250008 ; 0.124975
5	0.499734	0.083398	0.249598 ; 0.124800
6	0.500106	0.083256	0.250672 ; 0.125341
7	0.499936	0.083221	0.250224 ; 0.125003
8	0.500489	0.083101	0.250056 ; 0.124951
9	0.500311	0.083322	0.250615 ; 0.125378
10	0.499847	0.083176	0.249784 ; 0.124932

Global	**Moyenne μ**	**Variance v**	**Autocorrélation r_1 et r_2**
Fusion	0.500012	0.083320	0.250001 ; 0.125010

Pourquoi BBS est sécurisé ?

La **sécurité de l'algorithme Blum-Blum-Shub (BBS)** repose sur la **difficulté du problème de factorisation** des grands entiers.

- **Principe de base :** La génération des bits dans BBS est basée sur la congruence $x_{n+1} = x_n^2 \bmod N$. Prédire le bit suivant (ou les précédents) sans connaître la factorisation de N est équivalent au problème, tout aussi difficile, d'extraire des racines carrées modulo N.
- **Protection par sortie partielle :** Pour renforcer la sécurité, BBS ne révèle qu'une partie des bits générés à chaque étape, rendant la prédiction encore plus ardue.

En conclusion, sans la connaissance des facteurs p et q de N, un attaquant ne peut ni prédire ni reconstruire la séquence de bits générée par BBS.

Explication de la sortie du générateur BBS

Le générateur Blum Blum Shub produit une sortie de $\lfloor \log_2(n) \rfloor$ bits, où $n = p \cdot q$ est le module de BBS.

$$\underbrace{\underbrace{1001010101001100011010 10}_{\lfloor \log_2(\log_2 n) \rfloor \text{ bits pour l'utilisateur}} 0010 \ldots 0011010010}_{\lfloor \log_2(n) \rfloor \text{ bits générés}}$$

- Seuls les $\lfloor \log_2(\log_2 n) \rfloor$ bits les plus faibles sont sécurisés et peuvent être utilisés par l'utilisateur sans compromettre la sécurité cryptographique du générateur.
- Le reste est utilisé en interne par le générateur BBS pour calculer l'état suivant.
- Cela limite volontairement l'information exposée pour garantir la sécurité.

Nombre de possibilités pour l'utilisateur : $2^{\lfloor \log_2(\log_2 n) \rfloor}$

Limite de l'approche statistique

Suite (n=10)	Moyenne	Variance	Auto-corrélation
(0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5)	0.5	0	0.225

Implémentation de BBS

```python
import math
from decimal import Decimal, getcontext

def bbs_decimal_securise(p, q, Graine_Initiale, precision_decimale, nombre_sorties):
    n = p * q
    x = Graine_Initiale
    getcontext().prec = precision_decimale + 1
    log_log_n_bits = max(1, int(math.log2(math.log2(n))))
    liste_resultats = []
    for _ in range(nombre_sorties):
        x = (x * x) % n
        representation_binaire = bin(x)[2:].zfill(n.bit_length())
        bits_faible_poids = representation_binaire[-log_log_n_bits:]
        if bits_faible_poids:
            valeur_entier = int(bits_faible_poids, 2)
            denominateur = Decimal(2) ** len(bits_faible_poids)
            partie_fractionnaire = Decimal(valeur_entier) / denominateur
            liste_resultats.append(partie_fractionnaire)
        else:
            liste_resultats.append(Decimal(0))
    return liste_resultats

p = 666641161166556834225948703554306705433401937656314150371236940 7
q = 510206708652271046050482463791470224452847547892963460707870286 7
precision_decimale = 50
nombre_sorties = 100000
graine_initiale = 123456789

nombres_decimaux_aleatoires = bbs_decimal_securise(p, q, graine_initiale,
precision_decimale, nombre_sorties)
```

Test de fréquence du générateur du BBS

```
resultat = generateur(a, b, m, X_0, n)

nombre_colonnes = 100
plt.hist(resultat, bins=nombre_colonnes, color='skyblue', alpha=0.7, rwidth=0.8)
plt.title('Histogramme des valeurs générées')
plt.xlabel('Valeurs générées')
plt.ylabel('Fréquence')

# Afficher l'histogramme
plt.grid(axis='y', alpha=0.5)
plt.show()
```

Espérance

D'après le cours, si $X_i \sim \mathcal{U}([1, n])$, alors l'espérance de X_i est donnée par :

$$\mathbb{E}(X_i) = \frac{n+1}{2}$$

Or, on a $U_i = \frac{X_i}{n}$. Par linéarité de l'espérance, on obtient :

$$\mathbb{E}(U_i) = \frac{1}{n}\mathbb{E}(X_i) = \frac{1}{n} \times \frac{n+1}{2} = \frac{n+1}{2n}$$

En faisant tendre n vers l'infini, on obtient bien :

$$\mathbb{E}(U_i) \to \frac{1}{2}$$

Ce qui montre que l'espérance de U_i tend vers $\frac{1}{2}$.

Variance

D'après le cours, la variance d'une variable $X_i \sim \mathcal{U}([1, n])$ est donnée par :

$$\mathrm{Var}(X_i) = \frac{n^2 - 1}{12}$$

Or, $U_i = \frac{X_i}{n}$. Donc

$$V(U_i) = \frac{1}{n^2} V(X_i) = \frac{1}{n^2} \frac{n^2 - 1}{12} \to \frac{1}{12}$$

Ce qui montre que la variance de U_i tend bien vers $\frac{1}{12}$ pour $n \to \infty$.

Décomposition du champ

Le champ complexe au point d'observation s'écrit :

$$E = \sum_{n=1}^{N} A_n e^{i\phi_n} = \sum_{n=1}^{N} A_n(\cos\phi_n + i\sin\phi_n)$$

Donc :

$$\Re(E) = \sum_{n=1}^{N} A_n \cos\phi_n \qquad \Im(E) = \sum_{n=1}^{N} A_n \sin\phi_n$$

Hypothèses :

- Les phases $\phi_n \sim \mathcal{U}[0, 2\pi]$ sont **indépendantes**.
- On a : $\mathbb{E}[\cos\phi_n] = \mathbb{E}[\sin\phi_n] = 0$
- Et : $\mathrm{Var}[\cos\phi_n] = \mathrm{Var}[\sin\phi_n] = \frac{1}{2}$

Conséquence : Par le **théorème central limite**, pour $N \gg 1$:

$$\Re(E), \Im(E) \sim \mathcal{N}(0, \sigma^2) \quad \text{(gaussiennes centrées, indépendantes)}$$

Distribution du module du champ : loi de Rayleigh

On a établi que :

$$E = \Re(E) + i\Im(E) \sim \mathcal{N}_{\mathbb{C}}(0, \sigma^2)$$

c'est-à-dire que $\Re(E)$ et $\Im(E)$ sont :

Le module $|E|$ suit alors une **loi de Rayleigh** :

$$|E| = \sqrt{\Re(E)^2 + \Im(E)^2}$$

Sa densité est donnée par :

$$f_{|E|}(r) = \frac{r}{\sigma^2} e^{-\frac{r^2}{2\sigma^2}} \quad \text{pour } r \geq 0$$

Et donc : l'intensité $I = |E|^2$ suit une **loi exponentielle** :

$$f_I(I) = \frac{1}{\langle I \rangle} e^{-I/\langle I \rangle}$$

En 2015, lors de la sortie Tesla Model S, des chercheurs en sécurité ont révélé une faille majeure : il était possible d'ouvrir simplement toutes les Tesla en moins de 10 secondes, et celà uniquement à cause d'une vulnérabilité dans la génération des clés d'accès qui permettait d'accéder au véhicule. *Rq: C'est une anecdote que j'ai inventé.*

C'est la raison pour laquelle, je vais vous présenter aujourd'hui le travail que j'ai mené cette année dans le cadre de mon TIPE, où j'ai cherché à générer **des clés d'accès numériques sécurisées à partir d'un phénomène physique aléatoire**.

De nos jours, les clés d'accès sont au cœur de la sécurité informatique. Chaque fois que vous vous connectez à un site sécurisé en entrant votre email et votre mot de passe, le serveur en échange génère pour vous une clé d'accès temporaire qui vous permettra de naviguer sur le site. Cette clé d'accès est générée aléatoirement de telle sorte à ce qu' uniquement vous et le serveur puissiez la connaître, et donc grâce à cette clé secrète personne ne peut usurper votre identité et accéder à vos donnés. Mais si le processus de génération est biaisé ou prévisible comme dans le cas de la tesla, un attaquant pourrait facilement deviner la clé et donc accéder à votre espace personnel et avoir accès à vos informations privées. On comprend donc l'importance cruciale de disposer de **clés réellement aléatoires et imprévisibles**.

Toutefois un problème majeur apparaît : **les ordinateurs sont par nature déterministes**. Un même algorithme, avec les mêmes données d'entrée, produira toujours le même résultat. Donc l'aléatoire n'a pas sa place dans un ordinateur.

La solution pour dépasser cette limite ? c'est d'**Introduire une source physique véritablement aléatoire** comme graine d'entrée du générateur. Il suffira ensuite de la convertir en une donnée traitable par l'ordinateur pour générer ensuite des clés d'accès véritablement aléatoire.

C'est précisément l'objectif de mon TIPE : **combiner un phénomène physique aléatoire avec le déterminisme déterminisme des ordinateurs pour générer des clés d'accès sécurisées.**

Donc comme je l'ai dit, il fallait tout d'abord que je trouve un phénomène aléatoire qui servirait de source pour mon générateur.

Et le phénomène que j'ai choisi pour introduire cet aléatoire est le **speckle**.

Le phénomène de speckle correspond à une figure granuleuse qu'on observe lorsqu'un faisceau laser éclaire une surface rugueuse, c'est-à-dire diffusante. Intuitivement, on pourrait s'attendre, étant donné la précision d'un faisceau laser, à obtenir une image parfaitement nette. Pourtant, ce n'est pas le cas : on observe au contraire une multitude de taches lumineuses aléatoires, présentant des variations d'intensité très marquées.

Ce phénomène s'explique par le fait que chaque irrégularité microscopique de la surface diffuse la lumière dans toutes les directions. Les ondes issues de ces multiples diffusions interfèrent alors entre elles : dans certaines zones, les interférences sont constructives, renforçant l'intensité lumineuse ; ailleurs, elles sont destructives. C'est donc cette superposition complexe et aléatoire qui donne naissance au motif caractéristique du speckle.

De plus, au cours de mes recherches bibliographiques, j'ai également appris que l'amplitude des taches de speckle suit une distribution statistique : la loi de Rayleigh, qui est connue pour traduire justement les phénomènes de marches aléatoires.

Pour cela, j'ai défini un critère fondamental qui va guider l'ensemble de mon travail. Pour que la figure de speckle soit exploitable comme source d'aléa, deux conditions devront simultanément être respectées :

- d'une part, la granularité doit être clairement être grossièrement visible à l'œil nu.
- d'autre part, la distribution statistique de l'amplitude lumineuse doit suivre une loi de Rayleigh comme c'est la cas dans le modèle théorique.

Ce critère me permettra de valider que la figure d'interférence obtenue présente bien les caractéristiques d'un phénomène aléatoire, et donc qu'elle est apte à servir de base pour la suite du projet.

Et donc maintenant que l'on a un phénomène aléatoire théorique, il faut le réaliser expérimentalement.

Si l'on se réfère uniquement à la théorie, j'aurais très bien pu simplement prendre un laser et l'orienter sur une surface diffusante classique dans le laboratoire de mon lycée pour obtenir le speckle final. Mais il s'avère que j'ai vu qu'à SupOptique, ils proposaient un TP spécifique sur le speckle. J'ai donc contacté **Madame [censuré]** la responsable du laboratoire expérimental de SupOptique afin de réaliser ce TP.
Premièrement, Là-bas, j'ai pu profiter de matériel bien plus performant que ce que

j’aurais pu avoir dans le laboratoire de mon lycée, notamment une **caméra haute résolution**, ce qui m’a permis de capturer des speckle de très haute qualité.

Mais surtout, ce qui a été **particulièrement précieux**, c’est de pouvoir échanger avec Madame Bernard. Elle a vraiment pris le temps de comprendre mon problème, et de m’éclairer sur plusieurs points qui restaient encore flous pour moi. En particulier, elle a bien valider mon critère expérimental pour savoir si on avait bien un speckle aléatoire à savoir vérifier **Si à la fois il présente une granularité visible grossièrement et que sa distribution d'amplitude suit une loi de Rayleigh**

J'ai donc obtenu lors de cette expérience ce speckle valide. Mais un problème subsistait.

Ce montage produit un **speckle statique** : les figures de speckle obtenues restent stationnaires. Or, si l’objectif est de générer une multitude de clés aléatoires, il faut pouvoir produire une multitude de graines différentes, donc avoir une multitude de speckles différents.

Il fallait donc trouver un moyen de **rendre cette expérience non stationnaire**, de façon à générer à chaque fois un nouveau motif de speckle aléatoire.

J’ai donc cherché à créer un milieu diffusant dynamique. Mon idée initiale était d’utiliser une cuve contenant de la poudre d’oxyde d’aluminium en suspension, agitée mécaniquement à l’aide d’un barreau magnétique et de faire passer le faisceau laser à travers.

Sur le papier, le principe était séduisant : le mouvement brownien des particules devait maintenir un milieu diffusant en constante évolution, générant ainsi en continu de nouvelles figures de speckle uniques. En pratique cependant, le résultat s’est révélé très décevant : les images obtenues étaient dépourvues de la granularité caractéristique du speckle, et l’analyse de l’histogramme des amplitudes montrait clairement l’écart par rapport à la loi de Rayleigh attendue.

Après analyse des premières images, j’ai rapidement identifié l’origine du problème : il s’agissait d’un décalage entre les échelles de temps caractéristiques du phénomène et celles du temps d'exposition de la caméra.

En effet, les particules diffusantes étaient très agitées, ce qui faisait apparaître et disparaître les motifs de speckle à une vitesse extrêmement rapide. Le temps caractéristique de fluctuation des figures de speckle était de l’ordre de quelques

dixièmes de millisecondes. En revanche, le temps d’exposition de la caméra était quant à lui de l'ordre de quelques centièmes de seconde, soit largement trop long pour figer ces variations rapides.

En conséquence, l’image capturée correspondait en réalité à une moyenne temporelle sur de nombreuses configurations instantanées du speckle, ce qui avait pour effet de capturer une superposition floutée de multiples figures. Ce qui rend notre expérience inutilisable.

Pour résoudre ce problème, j’ai conçu un nouveau système permettant de **contrôler finement l’évolution du milieu diffusant,** c'est -à -dire qu'il viendrait bouger s'arrêter le temps que l'on capture le speckle, puis bouger de nouveau.

Pour cela, je me suis dit que je pouvais utiliser des milieux diffusants sous forme de plaques semi-transparentes. Fixées sur deux moteurs, ces plaques pourraient être mises en rotation de manière contrôlée : en les faisant tourner, puis en les arrêtant successivement, je pourrais faire évoluer le milieu diffusant à volonté, créant ainsi de nouvelles configurations aléatoires à chaque arrêt.

Mais vous allez surement me dire pourquoi deux milieux diffusants et deux moteurs? Car le problème c'est que lorsque le moteur a fait un tour le milieu diffusant redevient le même et donc on obtient de nouveau les mêmes figures. Mais en mettant deux moteurs qui tournent à des vitesses différentes, on démultiplie le nombre de combinaisons possibles et à fortiori le nombre de speckle générés.

Ici pour mes milieux diffusants je les ai réalisé avec du plexiglass que je suis venu poncer avec du papier de verre afin d'avoir ce côté rugueux qui diffuse.

Puis j'ai réalisé un système électronique avec arduino. Pour les moteurs, j'ai opté pour des moteurs pas à pas, leur avantage c'est que je pouvais facilement les faire tourner et les arrêter tout en connaissant leur position angulaire .Les moteurs que j'ai utilisé tournaient de 0.8 deg/pas et donc le système offre ainsi au maximum 202 500 combinaisons angulaires différentes, ce qui est largement suffisant pour générer un très grand nombre de speckles différents. J’ai donc programmé l’Arduino pour que le deuxième moteur tourne d'un pas, quand le premier moteur a fait un tour complet.

Une fois que j'ai créé ce milieu diffusant évolutif. J'ai laissé tourner mon expéreicne dans le noir pendant un week end complet. Et avec une caméra 4k j'ai pu filmer en continu les speckles obtenus. J'ai ensuite extrait tous les speckle et voici ce que j'ai analysé pour un speckle pris au hasard. Contrairement à l'expérience précédente : cette

fois-ci, la distribution de l'amplitude s’ajuste conformément à la loi de Rayleigh. Le critère est donc vérifié. Donc le phénomène aléatoire est donc validé.

Maintenant que j’ai obtenu un grand nombre d’images de speckle différentes, il me restait à transformer ces motifs optiques aléatoires en données numériques exploitables pour la génération de clés d'accès.

La première étape a été de normaliser les images.
Tout d’abord, j’ai procédé à un recadrage pour ne conserver que la zone utile. Puis, j’ai converti chaque image en niveaux de gris, et je les ai binarisées.

Pour la binarisation, j’ai utilisé un seuil basé sur l’intensité médiane de chaque image :

- les pixels d’intensité inférieure à la médiane sont convertis en 0 (noir),
- ceux supérieurs deviennent 1 (blanc).

Cette méthode permet d’équilibrer la proportion de 0 et de 1 autour de 50 %, ce qui est important pour éliminer d’éventuels biais lumineux globaux et obtenir une séquence binaire équilibrée.

Nous disposons ainsi de chaînes binaires aléatoires qui traduisent l’information du speckle. Mais un problème subsistait toujours : d’une image à l’autre, malgré le caractère aléatoire global, il pouvait subsister certaines corrélations locales. Or, pour une utilisation cryptographique, il est essentiel de briser totalement ces corrélations résiduelles afin d’obtenir un aléa véritablement imprévisible.

Pour cela, j’ai appliqué une fonction de hachage cryptographique. Les fonctions de hachage ont plusieurs propriétés intéressantes : elles sont uniformes, déterministes et extrêmement sensibles à la moindre variation de l’entrée. Le moindre changement, ne serait-ce que d’un seul bit entraîne une sortie totalement différente et imprévisible.

Dans un premier temps, j’avais testé une approche naïve, en réalisant un simple produit scalaire entre l’image binaire et une carte de poids prédéfinis. Mais cette méthode étant linéaire, elle ne présentait pas la non-linéarité forte nécessaire : de petites variations dans l’image entraînaient des variations faibles du résultat, ce qui est inacceptable en cryptographie, on veut quelque chose de clairement différent.

J’ai donc choisi d’utiliser une fonction de hachage reconnue : **SHA-256**, standard international développé initialement par la NSA.

Concrètement, l’image binaire est d’abord transformée en une séquence linéaire de bits, qui est ensuite découpée en blocs de 512 bits.
Chaque bloc est traité par des opérations non-linéaires successives : permutations, additions modulaires, décalages, etc,...
Au final, on obtient un condensat de 256 bits, extrêmement sensible à la moindre variation de l’entrée.

J'ai donc recodé de 0 cette fonction sur python, puis je l’ai appliqué aux plusieurs milliers d’images de speckle que j'avais capturé lors de l'expérience précédentes pour avoir les clés d'accès finales dont voici les 10 premières générées.

Enfin, afin de vérifier que les clés générées étaient bien aléatoires, j’ai tracé l’histogramme des clés d’accès obtenues. Comme le montre la figure (page 20), la distribution est très proche d’une loi uniforme : chaque clé a donc la même probabilité d’apparaître.
Par conséquent, aucun biais statistique n’est exploitable par un attaquant pour anticiper les clés générées, ce qui confirme le bon fonctionnement du protocole que j’ai développé.

En résumé :

À partir d’un phénomène optique chaotique — le speckle — j’ai réussi à extraire un aléa exploitable sous forme de clés cryptographiques robustes et imprévisibles, en combinant l’indétermination physique et le traitement informatique.

Ainsi, j’ai répondu à la problématique initiale de mon TIPE : convertir un véritable hasard physique à l'aide d'un ordinateur, afin de générer des clés d'accès fiables et sécurisées.