Générateur de nombres aléatoires (GNA)

OUALI Maher

Introduction:

Domaines d’utilité d’un GNA

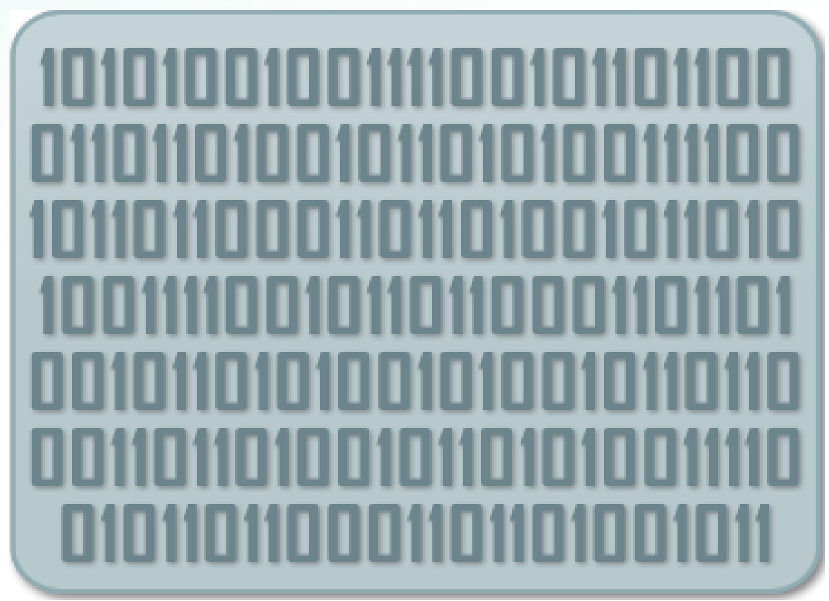
- Jeux d’argent
- Simulations
- Cryptographie
- Initialisation

Exigences d’un GNA

- Bon Caractère aléatoire
- Rapidité
- Immunité

Problématique:

Comment peut-on concevoir un tel système qui va ,d’une part , vérifier les exigences demandées , et d’autre part , être facile à construire ?



Plan:

- Conception et réalisation du GNA
- Vérification du caractère aléatoire à l’aide des tests statistiques
- Protection du système contre les attaques

Conception et réalisation du GNA

Source d’entropie → Traitement → Séquence aléatoire de bits

1. Phase de conception du GNA

a) Choix de la source d’entropie

Sources d'entropie / Caractéristiques	Source quantique	Bruit électrique	Gigue d'un oscillateur en anneau
Rapidité	Bonne	Moyenne	Moyenne
Qualité des données	Bonne	Bonne	Moyenne
Difficulté de fabrication	Difficile	Facile	Moyenne
Coût de fabrication	€€€	€	€€

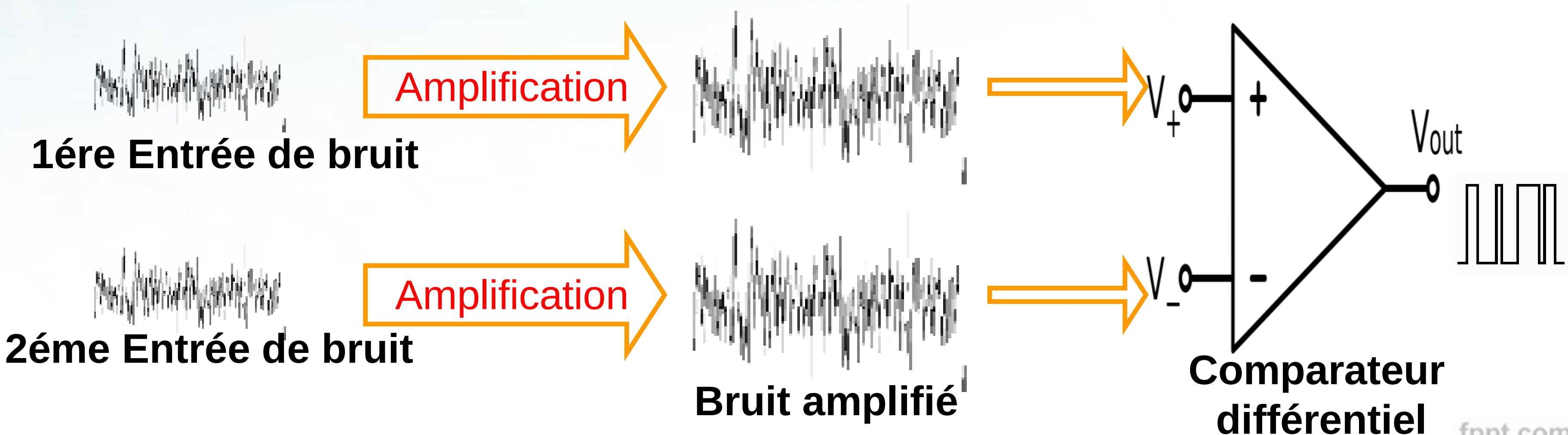
=> On choisit le Bruit électrique comme source d’entropie

b) Principe du fonctionnement du GNA

1ére idée:

Inconvénient majeur: la moyenne est difficile à déterminer.

2éme idée(plus optimale que la 1ére):



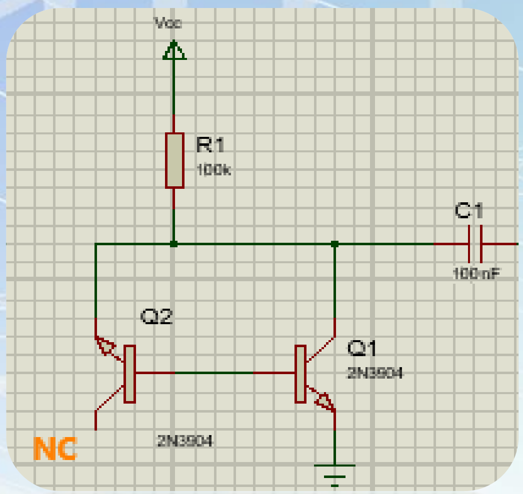
2 . Phase de fabrication du GNA proposé :

a) Génération du bruit:

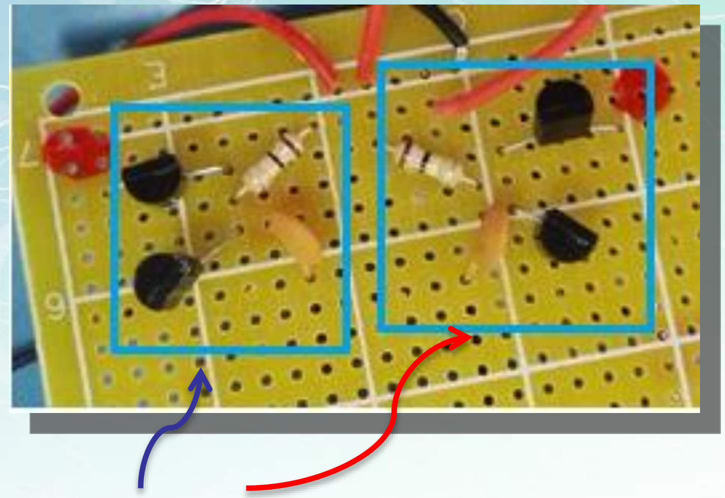
	Bruit thermique (Thermal noise)	Bruit d'avalanche	Bruit de scintillation (Flicker noise)
Amplitude (ordre de grandeur)	∿ 240µV (R=100KΩ ,Δf=1MHz,T=20°C)	∿ 70mV (Transistor NPN)	100µV
Spectre de fréquence	Constant	Freq ⬆ => Puis ⬇ (faiblement)	En 1/f
Dépendance de la température	OUI	Faible	NON
Dépendance de la tension	NON	OUI	OUI
Domaine de présence	Tous les conducteurs	Semi-conducteurs	Compsants actifs

=> On choisit le bruit d'avalanche comme source d'entropie à cause de la différence remarquable au niveau de l'amplitude du bruit

Le bruit d’avalanche est créé lorsque la jonction PN est utilisée dans le mode de claquage inverse. Ainsi , on adopte le circuit suivant:

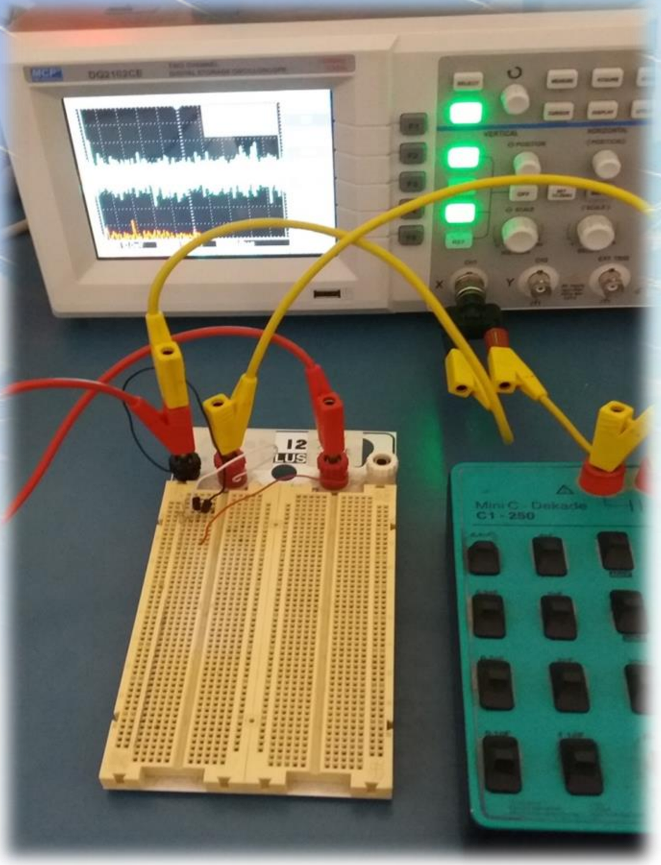


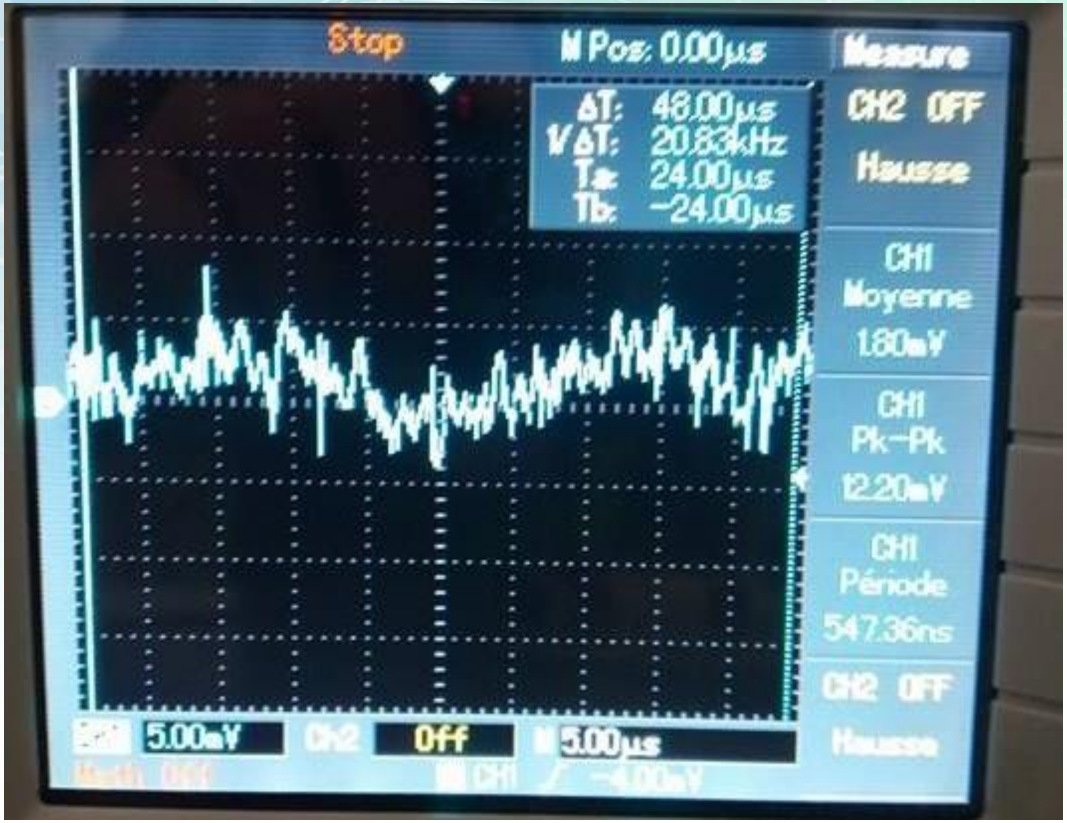
Schématique du circuit



Les deux sources de bruit sur une carte perforée

les deux signaux visualisé sur l’oscillo MCP

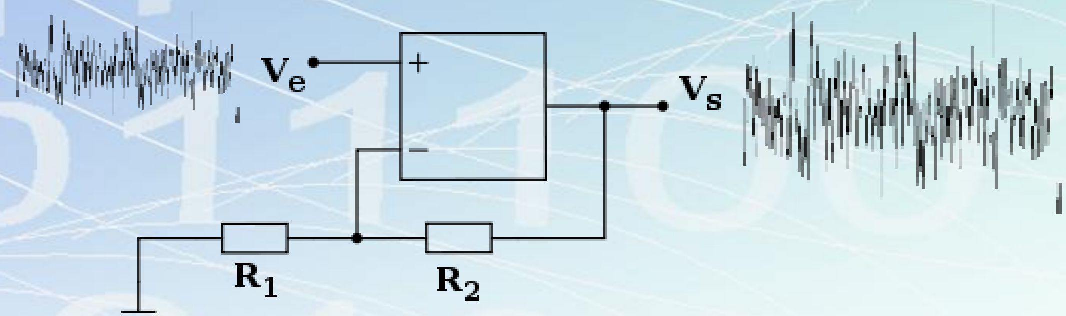




Implémentation du circuit sur un Breadboard et visualisation du bruit

b) Amplification du bruit

Pour cette phase , on va opter pour un circuit amplificateur non inverseur à base d'un amplificateur opérationnel (TL082).



Gain théorique:
G =(1+R2/R1)

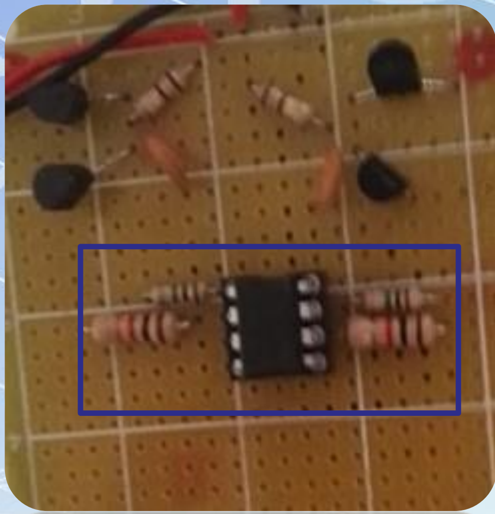
Vs(Pk-Pk)=890 mV

Ve(Pk-Pk)=70 mV

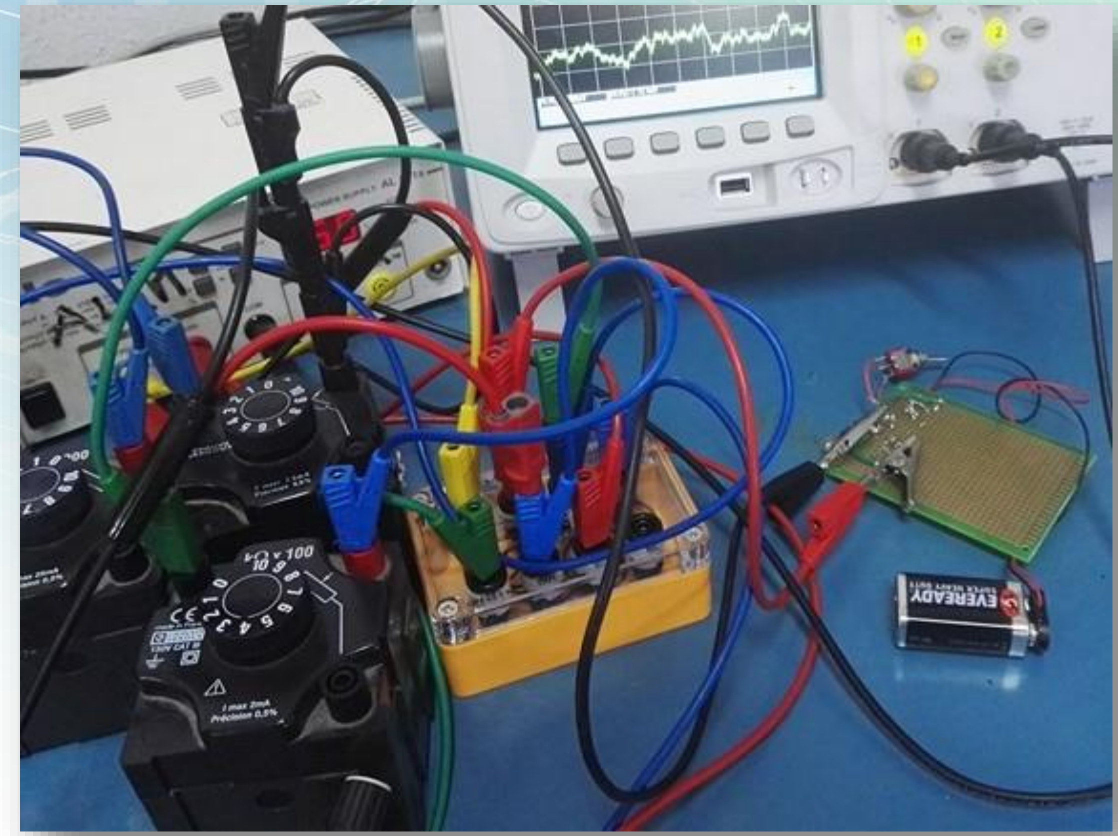
Gain pratique: 12.71

R2=1MΏ et R1=10K Ώ => G=101

=> On remarque que une différence entre le gain théorique et le gain pratique et c'est due à la bande passant limité (Δf= 2MHz) de l'AmpOp qui cause une distorsion du signal et élimine quelques composantes du bruit ce qui diminue le gain.



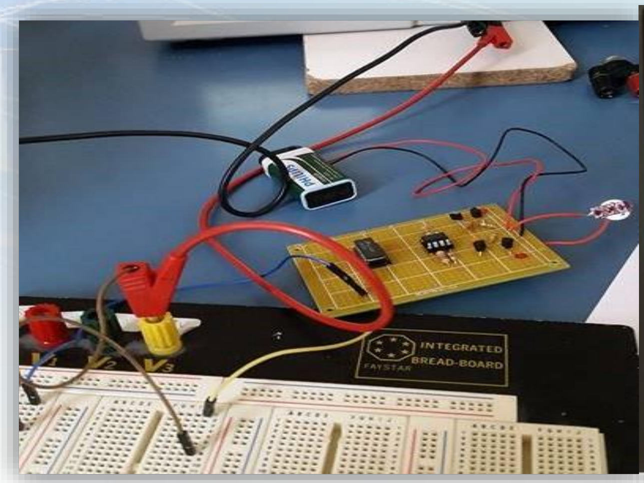
L’amplificateur de bruit sur une carte perforée (bloc en bleu)

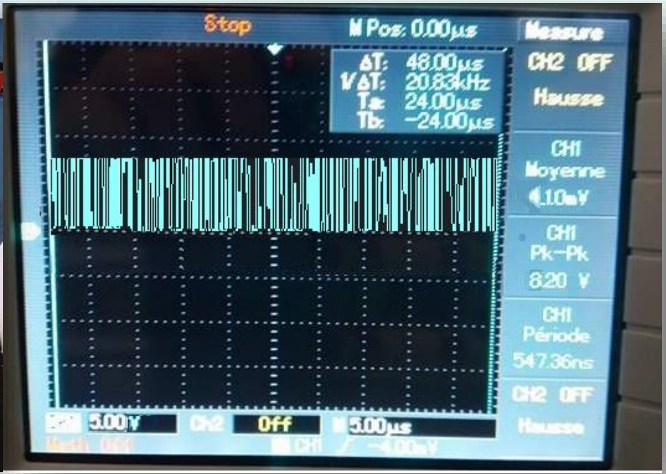


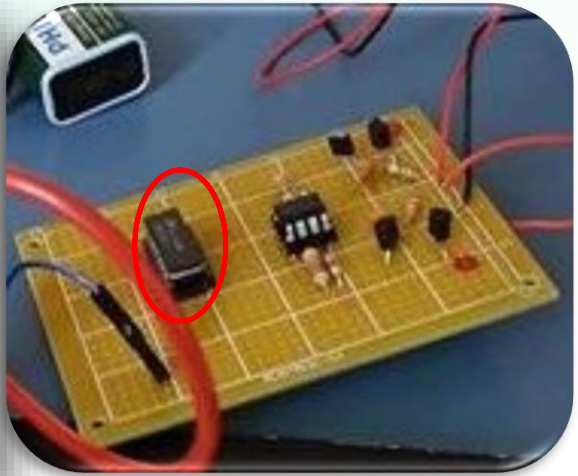
Test de l’amplification avec des outils du labo de l’électronique

c) Comparaison des deux bruits

Pour cette phase , on va opter pour un comparateur (LM139j) qui peut détecter une tension minimale de l’ordre de 2 mV donc on peut aisément comparer les deux signaux.







a) Test de la comparaison

b) Visualisation du Vs

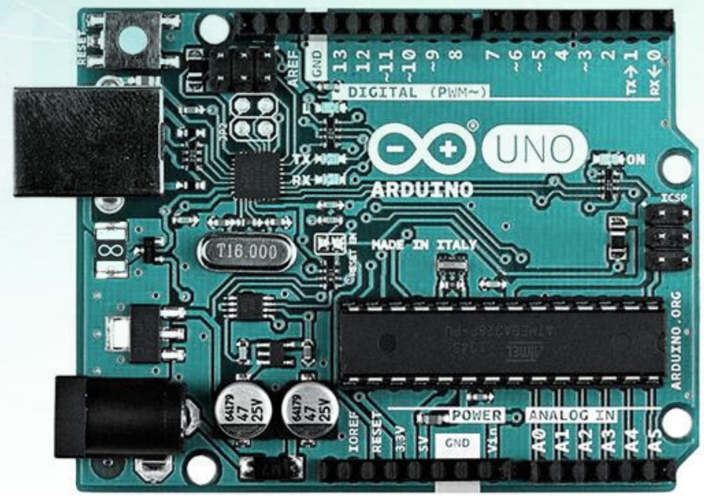
c) comparateur sur carte perforée (bloc en rouge)

d) Echantillonnage du signal

Pour cette phase , on va utiliser une carte ArduinoUno avec le code suivant:


```
int analogPin = 3;
int val = 0;          // variable to store the value read

void setup()
{
  Serial.begin(9600);          //  setup serial
}
void loop()
{
  val = analogRead(analogPin);    // read the input pin
  if val == 0
  {Serial.println('1');}
  else
  {Serial.println('0');}
              // debug value
}
```



Débit : 9600 bits / seconde

=> Critère de la rapidité est validé

Vérification du caractère aléatoire

Le rôle de cette partie est de valider le 1er critère (le plus important):

a) Test visuel: Bitmap Binaire

- Implémentation sur Python:

Principe simple: 1 => Point Noir   0=> Point Blanc

```
from PIL import Image
import numpy as np
### test visuel bitmap image d'une sequence aleatoire
def bitmap(ch):
        n,h=len(ch),int(sqrt(len(ch)))
        data = [[[255*(1-int(ch[h*i+j]))]*3 for j in range(h)]for i in range(h)]
        data = np.array(data, dtype=np.uint8)
        img = Image.fromarray(data, 'RGB')
        img.save('my.png')
        img.show()
```

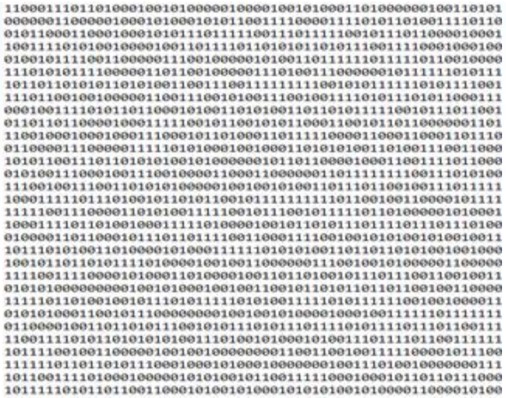
- Test sur un séquence générée par le GNA:

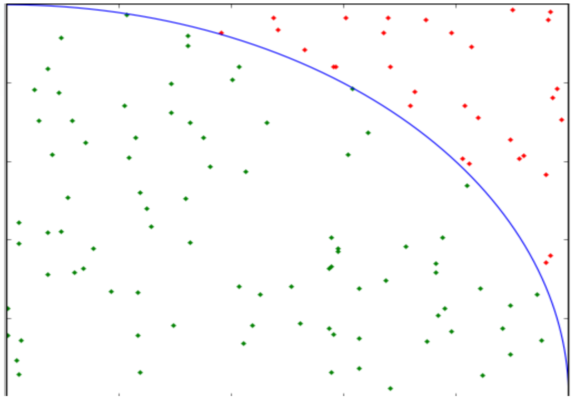
<u>Rq:</u> On remarque l'absence d'un motif particulier qui se répète

b) Détermination de la valeur approximative de π par la méthode de Monte Carlo

- <u>1ére étape</u>: on construit à partir de 16 bits un couple de réels entre 0 et 1 et on le projette sur le plan (x,y).
- <u>2éme étape</u>: on calcule le nombre 'nbr' de points à l'intérieur du quart du cercle de centre (0,0) et de rayon 1
- <u>3éme étape</u>: la valeur est (4*nbr/nombre total de couples)

```python
def montecarlo(ch):
    p=len(ch)//8
    T=[]
    for i in range(p):
        if(i%2==0):
            a1=0
            for j in range(8):
                a1+=int(ch[8*i+j])*(1/float(2**(j+1)))
        else:
            a2=0
            for j in range(8):
                a2+=int(ch[8*i+j])*(1/float(2**(j+1)))
            T.append((a1,a2))
    somme_out,somme_in=0,0
    X_in,Y_in,X_out,Y_out=[],[],[],[]
    for i in range(len(T)):
        if(T[i][0]**2+T[i][1]**2>1):
            somme_out+=1
            X_out.append(T[i][0])
            Y_out.append(T[i][1])
        else:
            somme_in+=1
            X_in.append(T[i][0])
            Y_in.append(T[i][1])
    plt.plot(X_in,Y_in,'g.')
    plt.plot(X_out,Y_out,'r.')
    circle()
    plt.show()
    n=len(T)
    approx_pi=4*somme_in/float(n)
    print('la valeur approximative de pi selon la methode monte carlo est ',approx_pi)
```





Val_approx = 2.931034482758620 6

c) Tests statistiques de NIST (National Institute of Standards and Technology)

Séquence de bits → 15 tests de NIST → Séquence est aléatoire
Ou
Non aléatoire

(suivant le modèle de NIST)

Le but c'est ne pas d'avoir une séquence aléatoire parfaite mais d'avoir une séquence proche (marge d'erreur prés) du modèle mathématique défini par les chercheurs du NIST .

Critères d'Aléatoire cherchés par NIST:

1. Freq(1) = Freq(0)

a) Globalement: (Dans la séquence entière) => Test #1: **Monobit**

Modèle mathématique: **0 ⇔ -1 et 1 ⇔ 1 puis on somme, ce qui nous donne une variable aléatoire $S \approx B(n, \frac{1}{2})$**

Pour 'n' assez grand → **Théorème central limite** → **$S \approx N(0,n)$**

- Implémentation sur python du 1er test:

```
from scipy import special
from math import *
def monobittest(ch):
    """monobit test: determiner si la frequence des 1 est egale a celle des 0 """
    n=len(ch)
    if(n>=100):
        s=0
        for i in range(n):
            s=s+(2*int(ch[i])-1)
        k=(abs(s)/sqrt(n))
        print('abs(S)/sqrt(n)  : ',k)
        a=(special.erfc(k/sqrt(2)))
        res=bool(a>0.01)
        return(a,' / le resultat du premier test est ',int(res)*'random'+(1-int(res))*'non_random')
    else:
        return('ERROR !! la longueur de la sequence est insuffisante ')
```

Rq: ce test est critique. S'il n'est pas passé, on ne vérifie pas les autres tests.

b) Localement: (Dans des sous-blocs de longueur M de la séquence binaire)

=> Test #2: **FrequencyWithinBlock**

Modèle mathématique: **0 ⇔ -1 et 1 ⇔ 1, lorsque 'M' est assez grand (contrainte du test), on trouve Si ≈ N(0,1) avec i ∈ [1..n/M]**
=> S = ∑ Si² suit la loi de χ^2 avec n/M degrés de liberté

- Implémentation sur python du 2éme test:

```
def frequencybleocktest(ch):
    """frequency within a block test:determiner si la frequence des 1 est 
egale a celle des 0 dans des sous-sequences disjointes"""
    L=[]
    n=len(ch)
    p=n//M
    if(n>=M*p)and(n>=100)and(M>0.01*n)and(M>=20)and(p<100):
        for i in range(p):
            s=0
            for j in range(M):
                s+=int(ch[i*M+j])
            L.append(s/float(M))
        s=0
        for i in range(p):
            s+=(L[i]-0.5)**2
        K=4*M*s
        print('chi squared , ',K)
        a=1-(special.gammainc(p/float(2),K/float(2)))
        res=bool(a>0.01)
        return(a,' / le resultat du deuxieme test est ',int(res)*'random'+(1-int(res))*'non_random')
    else:
        return('ERROR !! la longueur de la sequnece est insufisante')
```

2. Périodicité des oscillations entre 0 et 1

- Globalement: (Dans la séquence entière) => Test #3: **Runs**

Modèle mathématique: $\varepsilon_{K+1} = \varepsilon_K$ ⇔ 1 et $\varepsilon_{K+1} \mathrel{!}= \varepsilon_K$ ⇔ **0 puis on somme, ce qui nous donne une variable aléatoire V ≈ B(n, ½)**

Pour 'n' assez grand — **Théorème central limite** → **V ≈ N(2nϴ(1- ϴ) , 4n ϴ²(1- ϴ)²)**

avec **ϴ**=∑ *des '1'* **/n** **(= n/2 cas parfait)**

- Implémentation sur python:

```
def runstest(ch):
    """runs test:caracterise l'oscillation entre des 0 et 1 en comptant
le nombre des series non interrompues des 1 """
    n=len(ch)
    if(n>=100):
        s=0
        for i in range(n):
            s=s+int(ch[i])
        s=s/float(n)
        if(abs(s-0.5)<(2/sqrt(n))):
           v=1
           for i in range(n-1):
               v+=1-int(ch[i]==ch[i+1])
           print('Vn(obs)  : ',v)
           k=abs(v-2*n*s*(1-s))
           k1=2*sqrt(2*n)*s*(1-s)
           a=special.erfc(k/float(k1))
           res=bool(a>0.01)
        else:
            a=0.00
            res=False
        return(a,' / le resultat du troisieme test est ',int(res)*'random'+(1-int(res))*'non_random')
    else:
        return('ERROR !! la longueur de la sequence est insuffisante')
```

3. Absence d'un motif particulier

Déterminer le nombre d'occurrence d'un motif cible

a) une série de '1' (ou '0') de longueur -p-: => Test #4: **OverlappingTemplateMatch**

1) On divise la séquence sur N blocs de longueur M et on détermine le nombre d'occurrence 'p' du motif cible dans chaque blocs sans chevauchement même dans le cas où il ' y a correspondance

2) Incrémenter la fréquence F_p par 1 et puis la comparer à la valeur théorique

b) une séquence apériodique de 1 et 0 de longueur -m-: => Test #5: **Non-OverlappingTemplateMatch**

1) On divise la séquence sur N blocs de longueur ‘m’ et on détermine le nombre d’occurrence ‘p_i’ du motif cible dans chaque blocs avec chevauchement dans le cas où il’ y a correspondance.
2) On calcule l’espérance μ et la variance σ^2 .
3) On compare les valeurs empiriques aux paramètres théoriques (μ,σ) en utilisant une variable aléatoire qui suit la loi Khi-deux avec N degrés de liberté .

F_0	**0.364091**
F_1	**0.185659**
F_2	**0.139381**
F_3	***0.100571***
F_4	***0.070432***
F_5	***0.139865***

Valeurs théoriques

$$\mu = (M-m+1)/2^m.$$

$$\sigma^2 = M\left(\frac{1}{2^m} - \frac{2m-1}{2^{2m}}\right)$$

M=n/N

- Implémentation sur python du 4éme test:

```
def overlap_templ_match_test(ch,trgt='111111111',long_bloc=1032,nbr_bloc=968):
    n=len(ch)
    val_theor=[0.364091,0.185659,0.139381,0.100571,0.070432,0.139865]
    K=5#need to be 5
    M=long_bloc
    N=nbr_bloc
    b=len(trgt)
    lamda=(M-b+1)/float(2**b)
    nu=lamda*0.5
    if(N*0.070432>5)and(n>=10**4):
        L=[]
        for i in range(N):
            ch1=''
            for j in range(M):
                ch1=ch1+ch[i*M+j]
            L.append(ch1)
        L1=[]
        for i in range(N):
            s=0
            j=0
            while(j<=M):
                if(L[i][j:j+b]==trgt):
                    s+=1
                j+=1
            L1.append(s)
        T=[0]*6
        for i in range(N):
            if(L1[i]>=5):
                T[5]+=1
            else:
                T[L1[i]]+=1
        Xobs=0
        for i in range(K+1):
            Xobs+=((T[i]-N*val_theor[i])**2)/float(N*val_theor[i])
        print('Xobs : ',Xobs)
        a=1-(special.gammainc(K/float(2),Xobs/float(2)))
        res=bool(a>0.01)
        return(a,' / le resultat du huitieme test est ',int(res)*'random'+(1-int(res))*'non_random')
    else:
        return('ERROR !! la sequence est de longueur insuffisante')
```

- Implémentation sur python du 5éme test:

```python
def nonoverlap_templ_match_test(ch,trgt='00000001',nbr_bloc=8):
    """non-overlapping template match test: le but de ce test est de detecter la repitition d'un mot
    n=len(ch)
    N=nbr_bloc
    b=len(trgt)
    M=n//N
    var=M*((1/float(2**b))-((2*b-1)/float(2**(2*b))))
    moy=(M-b+1)/float(2**b)
    print(moy,var)#

    if(N<=100)and(b in range(2,11))and(M>0.01*n):
        L=[]
        for i in range(N):
            ch1=''
            for j in range(M):
                ch1=ch1+ch[i*M+j]
            L.append(ch1)
        L1=[]
        for i in range(N):
            s=0
            j=0
            while(j<=M):
                if(L[i][j:j+b]==trgt):
                    s+=1
                    j+=3
                else:
                    j+=1
            L1.append(s)
        Xobs=0
        for i in range(N):
            Xobs+=((L1[i]-moy)**2)/float(var)
        print('Xobs : ',Xobs)
        a=1-(special.gammainc(N/float(2),Xobs/float(2)))
        res=bool(a>0.01)
        return(a,' / le resultat du septieme test est ',int(res)*'random'+(1-int(res))*'non_random')
    else:
        return('ERROR !! la sequence est de longueur insuffisante ')
```

4. Entropie maximale

1) On détermine tous les motifs possibles de longueur M bits et on les note C^{M}_{mot}
2) On calcule la fréquence π_{mot} de chaque C^{M}_{mot}
3) On calcule l'entropie de Shanon $E(m)=\sum \pi_{mot} * \log(\pi_{mot})$
4) On calcule E(m+1), on aura ainsi ApEn(m)=E(m)-E(m+1)

Implémentation sur Python

```
def Approx_entrop_test(ch,m=3):
    n=len(ch)
    if(n>=1000)and(m<int(log(n,2))-5):
        L=[]
        for i in range(2):
            L.append(ch+ch[0:m+i-1])
        lis=[]
        for i in range(2):
            lis.append(toutes_combinaisons(m+i))
        T=[]
        for i in range(2):
            temp=[]
            for j in range(len(lis[i])):
                temp.append(occurence(lis[i][j],L[i]))
            T.append(temp)
        for i in range(2):
            s=0
            for j in range(len(T[i])):
                if(T[i][j] != 0):
                    T[i][j]=T[i][j]/float(n)
                    s+=(T[i][j])*log(T[i][j])
            T[i]=s
        Xobs1=2*n*(log(2)-T[0]+T[1])
        print('Xobs1 : ',Xobs1)
        a1=1-(special.gammainc(2**(m-1),Xobs1/float(2)))
        res=bool(a1>0.01)
        return(a1,' / le resultat du deuxieme test est ',int(res)*'random'+(1-int(res))*'non_random')
    else:
        return('ERROR !! la sequence est de longueur insuffisante ')
```

5. Indépendance

1) On construit des matrices binaires (32,32)(contrainte fixée par NIST) à partir de la séquence testée.
2) On détermine la probabilité que la matrice ait un rang=32, celle pour un rang=31 et celle pour un rang=30 (le reste est négligeable).
3) On compare les données empiriques aux valeurs théoriques (voir tableau) à l'aide d'une variable aléatoire qui suit la loi khi-deux ayant 2 degrés de liberté.

Implémentation sur Python

```
def matrice_binaire_test(ch):
    val_theor=[0.2888,0.5776,0.1336] #
    n=len(ch)
    Q,M=32,32 #ligne
     #colonne   #valeurs theoriques due aux approximations considerees
    if(n>=38*M*Q):
        L=[]
        p=n//(M*Q)

        l=0
        for i in range(p):
            L1=[]
            for j in range(M):
                L2=[]
                for k in range(Q):
                    L2.append(int(ch[l]))
                    l+=1
                L1.append(L2)
            L.append(np.matrix(L1))
        L2=[np.linalg.matrix_rank(L[i]) for i in range(len(L))]
        L1=[0,0,0]
        for i in range(p):
            if (L2[i]==M):
                L1[0]+=1
            elif(L2[i]==M-1):
                L1[1]+=1
            else:
                L1[2]+=1
        Xobs=((L1[0]-p*val_theor[0])**2/float(p*val_theor[0]))+((L1[1]-p*val_theor[1])**2/float(p*val_theor[1]))+
        print('Xobs : ',Xobs)
                                                ((L1[2]-p*val_theor[2])**2/float(p*val_theor[2]))
        a=1-(special.gammainc(1,Xobs/float(2)))
        res=bool(a>0.01)
        return(a,' / le resultat du cinqueme test est ',int(res)*'random'+(1-int(res))*'non_random')
    else:
        return('ERROR !! longueur de sequence insuffisante ')
```

P_{32}	0.2888
P_{31}	0.5776
P_{30} (et <30)	0.1284

Valeurs théoriques

- Résultats obtenus à partir d'une séquence de 10^6 bits générée par le circuit:

Test	Valeur de Probabilité	Conclusion
Monobit Test	0.1323	Acceptée
FrequencyWithinBlock	0.8814	Acceptée
Runs Test	0.5461	Acceptée
Test de correspondance du motif '1111' sans chevauchement	0.1634	Acceptée
Test de correspondance du motif '0001' avec chevauchement	0.0582	Acceptée
Entropie Approximative	0.3994	Acceptée
Matrice Binaire	0.8399	Acceptée

=> Critère de la qualité de données est validé

Protection du GNA contre les attaques

Les GNA sont toujours en danger à cause des attaques externes qui puissent se produire.

Parmi ces attaques, on cite :

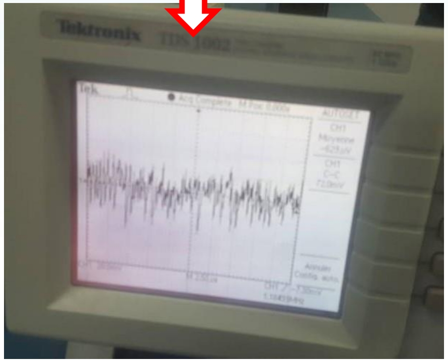
a) Injection de fréquence ou tension:

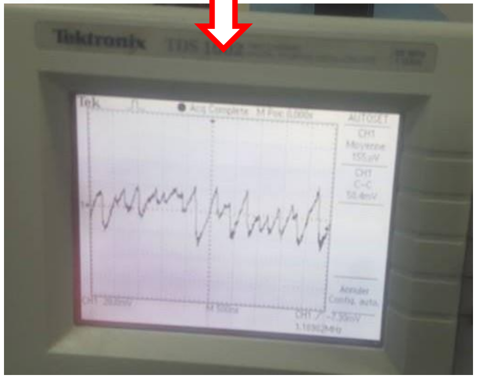
Le pirate essaie de saturer le système avec cette technique pour avoir une série de 1 ou 0 par exemple mais avec la comparaison différentielle toute fréquence ajoutée va être éliminée car elle affecte les deux sources de la même façon.

b) Variation de la température:

Cette technique menée par l'agent externe vise à changer les caractéristiques des composants en variant la température pour diminuer la qualité du bruit généré.

Echauffement	Refroidissement
Séchoir	Module à effet Peltier
80°C	10°C





=> On remarque un faible changement au niveau du signal du bruit, ainsi, le critère de l'immunité est validé

Conclusion:

On conclut à la fin que la mise en place d'un GNA de haute performance demande une démarche scientifique approfondie qui couvre plusieurs domaines.