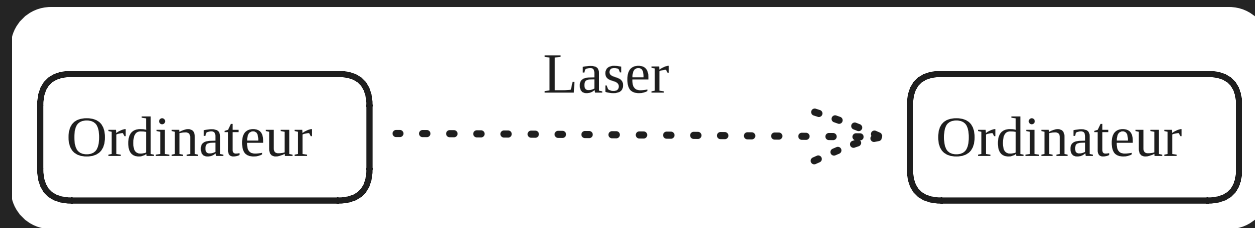


COMMUNICATION PAR LASER À GRANDE DISTANCE

CAYZAC Erhel 32242





Principe

POURQUOI ?

- Haut débit de transmission
- Liaison sans fil sécurisée
- Lien analogique / numérique

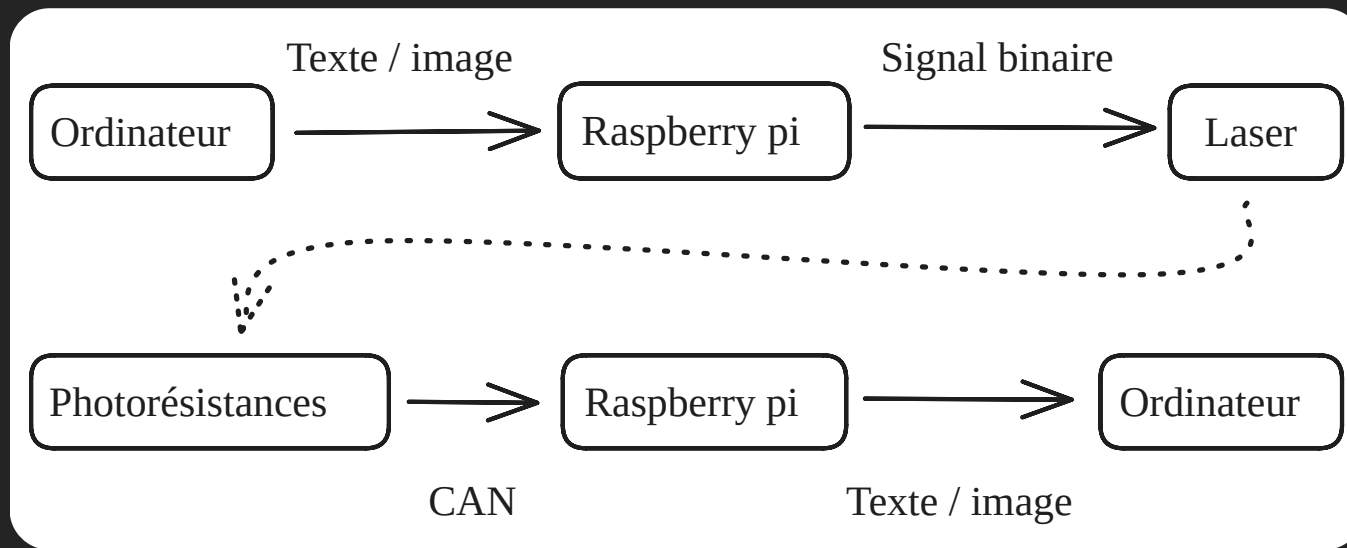


Schéma de principe

PLAN

I PREMIER PROTOTYPE

Électronique, transmission,
blocages, limitations

II AMÉLIORATION DU MODÈLE

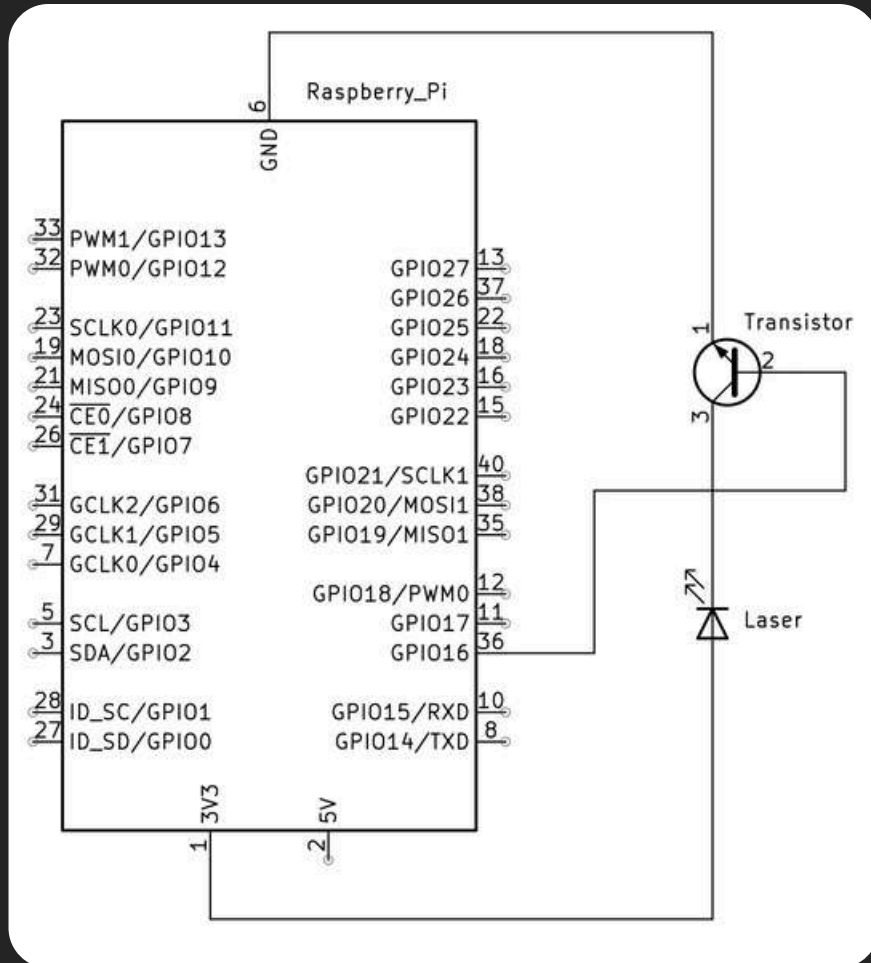
Capteurs, codes correcteurs,
vitesse d'émission

III LIMITES, ÉTUDE THÉORIQUE

Analyse théorique du projet
proposé

UN PREMIER PROTOTYPE

1. Émission
2. Protocole de communication
3. Réception
4. Problèmes rencontrés



- Envoi de bits
- Transistor
- Laser 10mW, 532nm, classe III

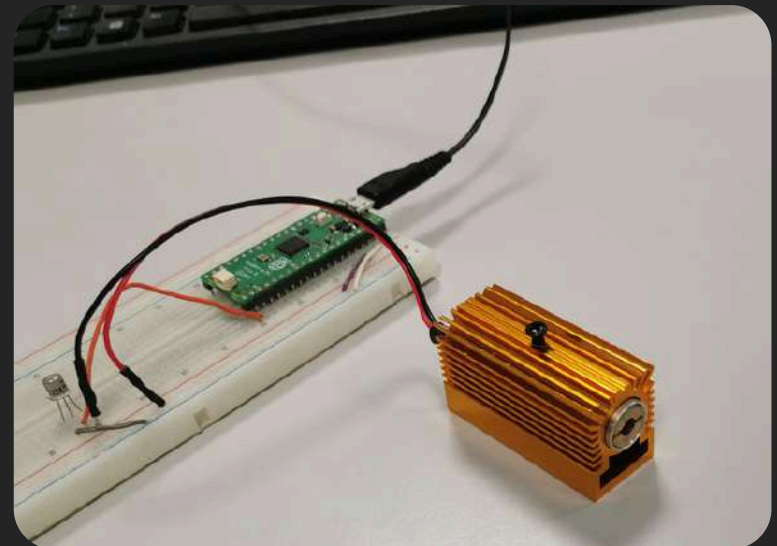


Schéma électrique émission

DÉTECTER L'ÉMISSION

Émission : “10110”

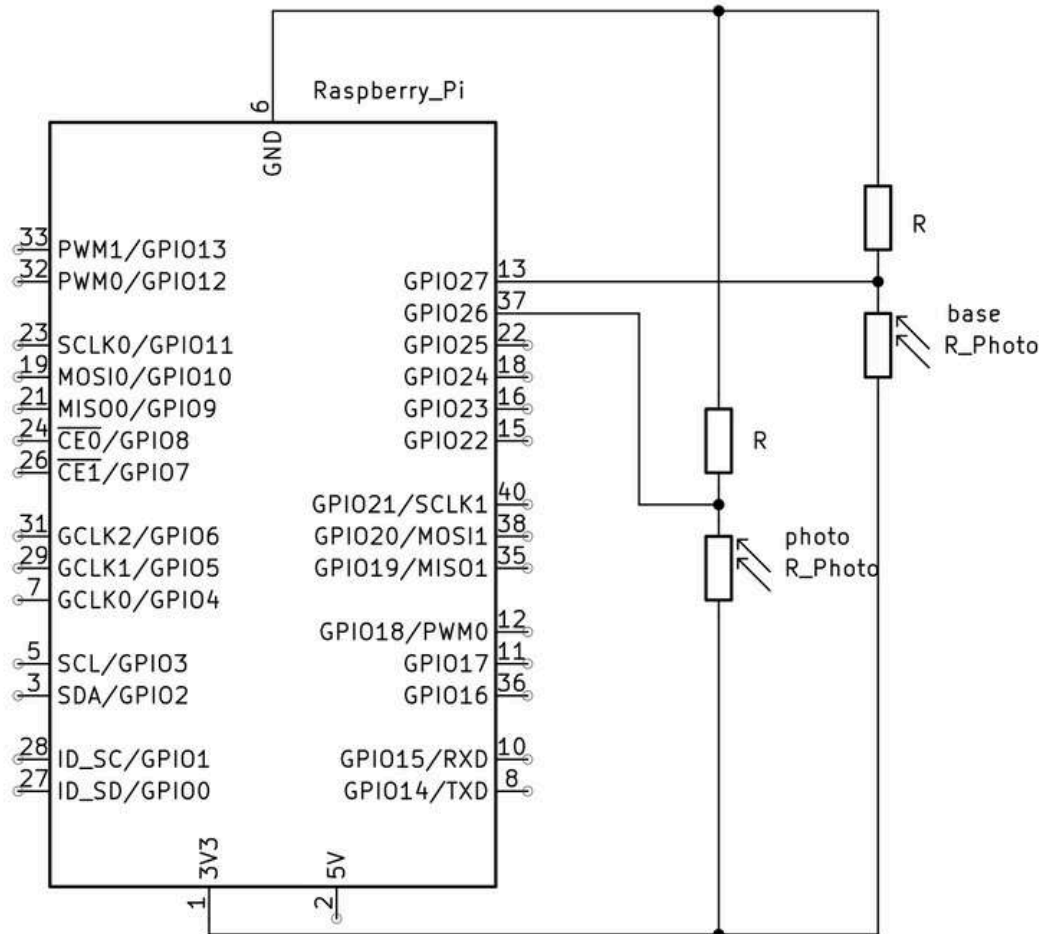
Réception : “...00010110000...”



Signal brut : “10110”

Émission : “1101101”

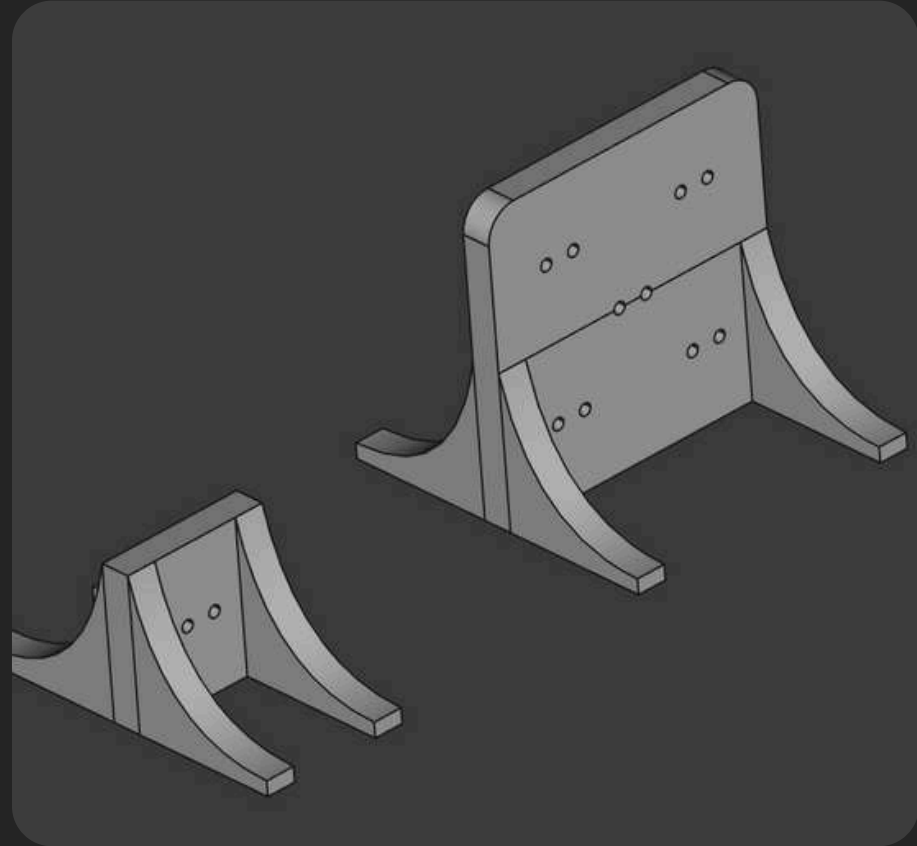
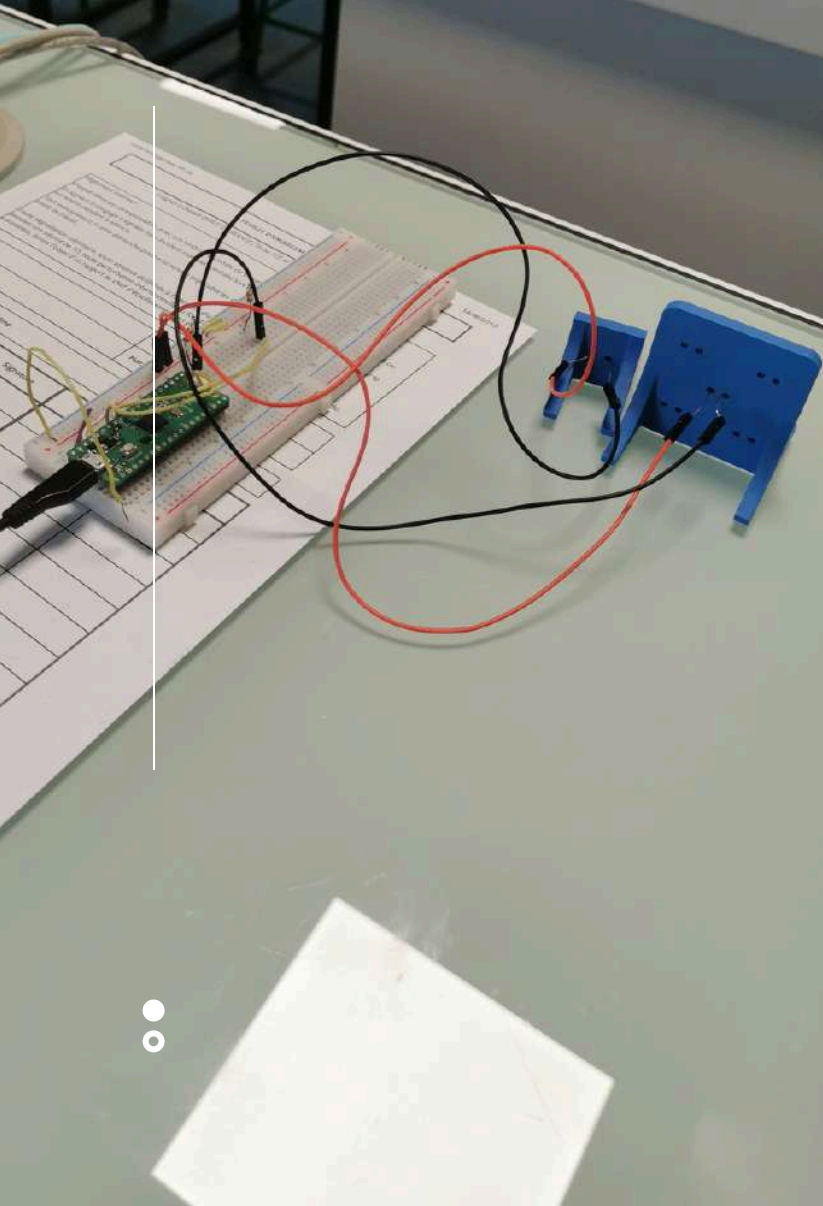
Réception : “...0001101101000...”



- Diviseurs de tension
- CAN
- Comparaison
- Seuil

Schéma électrique réception

10



Communication par laser à grande distance

Réception

- Création de supports pour les photorésistances

PREMIÈRES LIMITATIONS

- Non adaptabilité (seuil)
- Non linéarité (photorésistances)
- Manque de fiabilité (lenteur Raspberry)

SECOND PROTOTYPE

1. Phototransistors
2. Traitement différé
3. Seuil variable
4. Gestion des erreurs

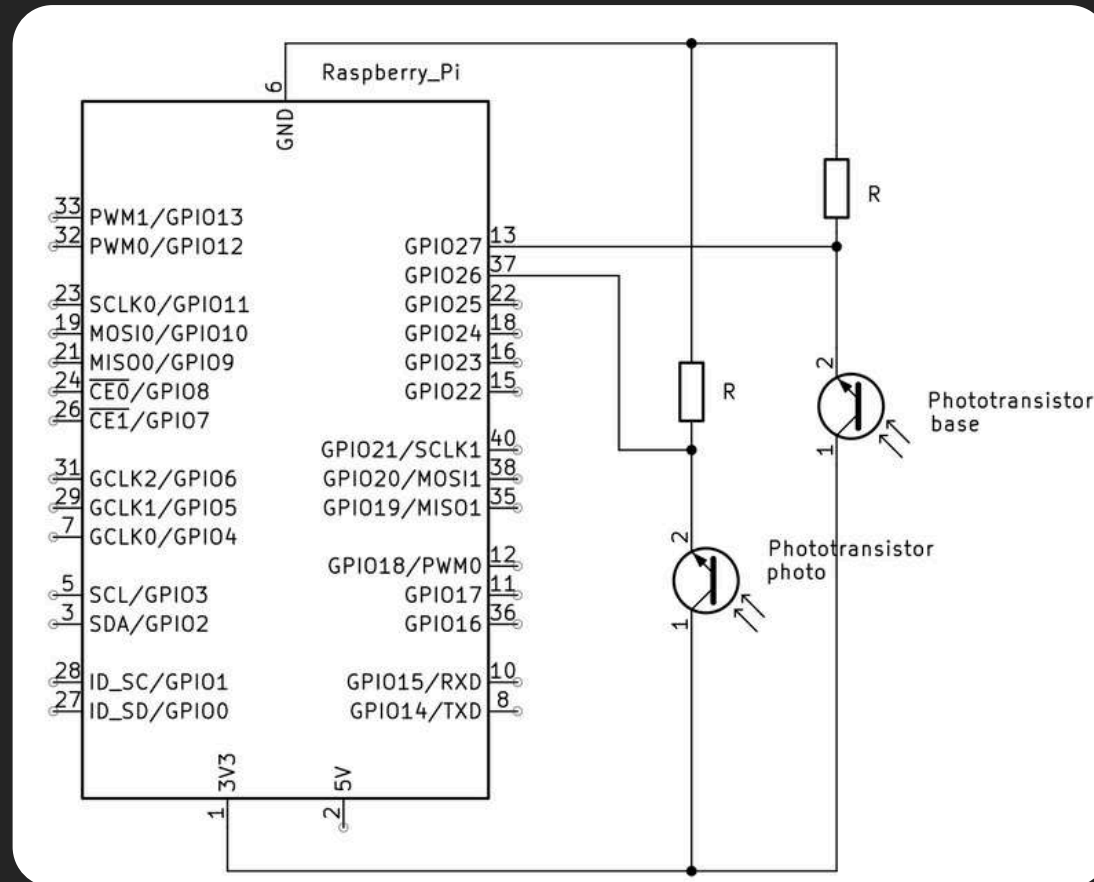
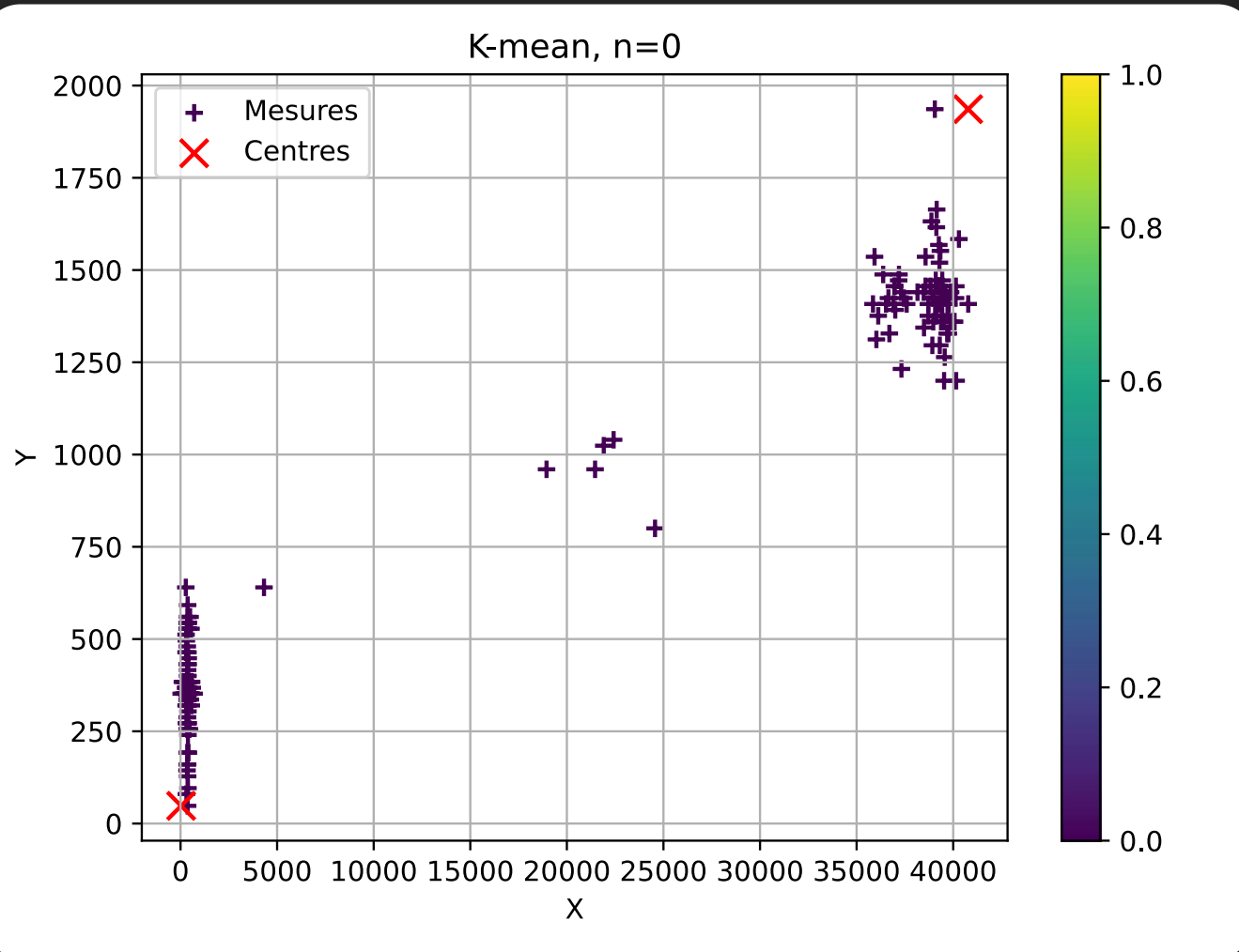


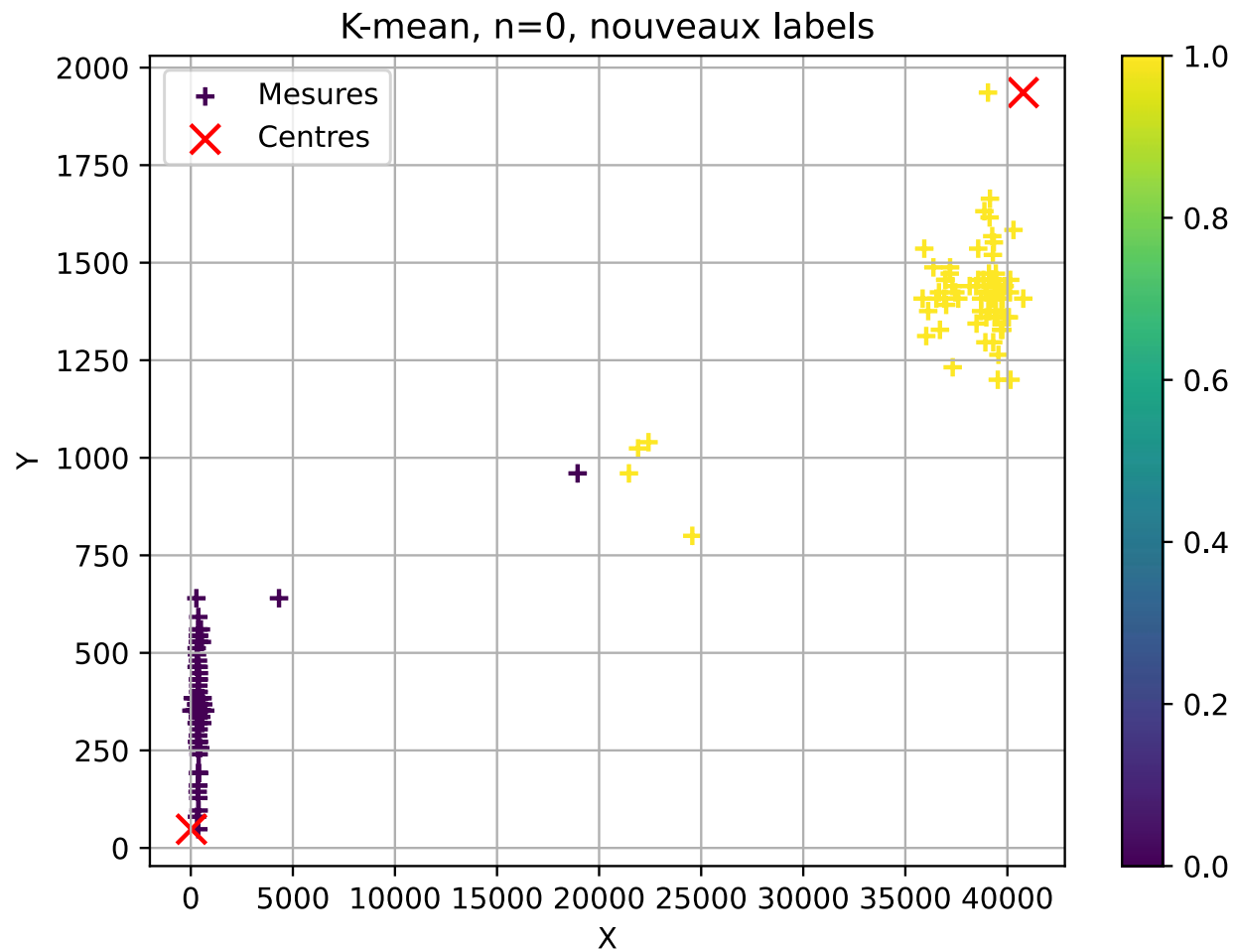
Schéma électrique réception

SEUIL DE DÉTECTION

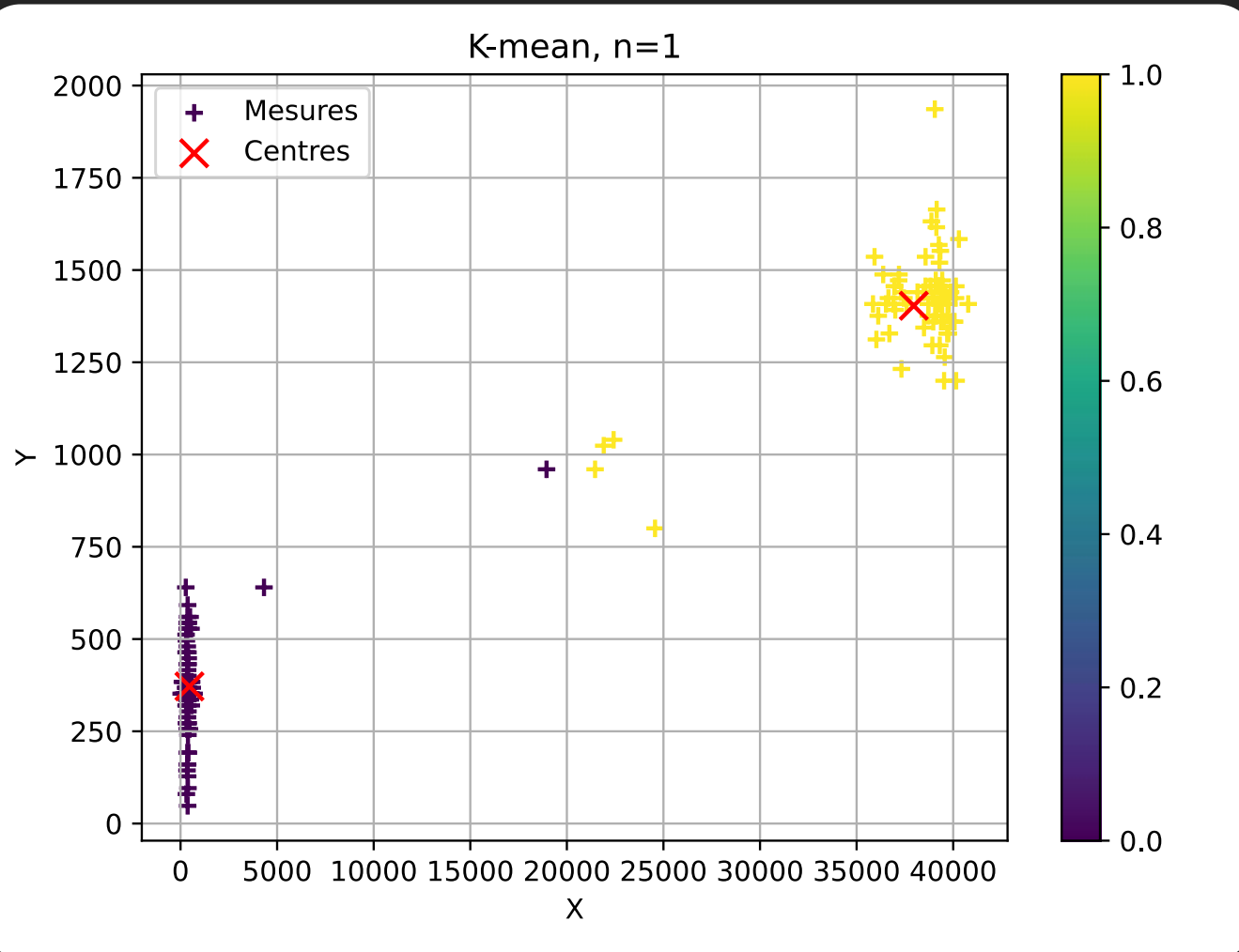
- Comparaison des données capteur
- Partitionnement des données



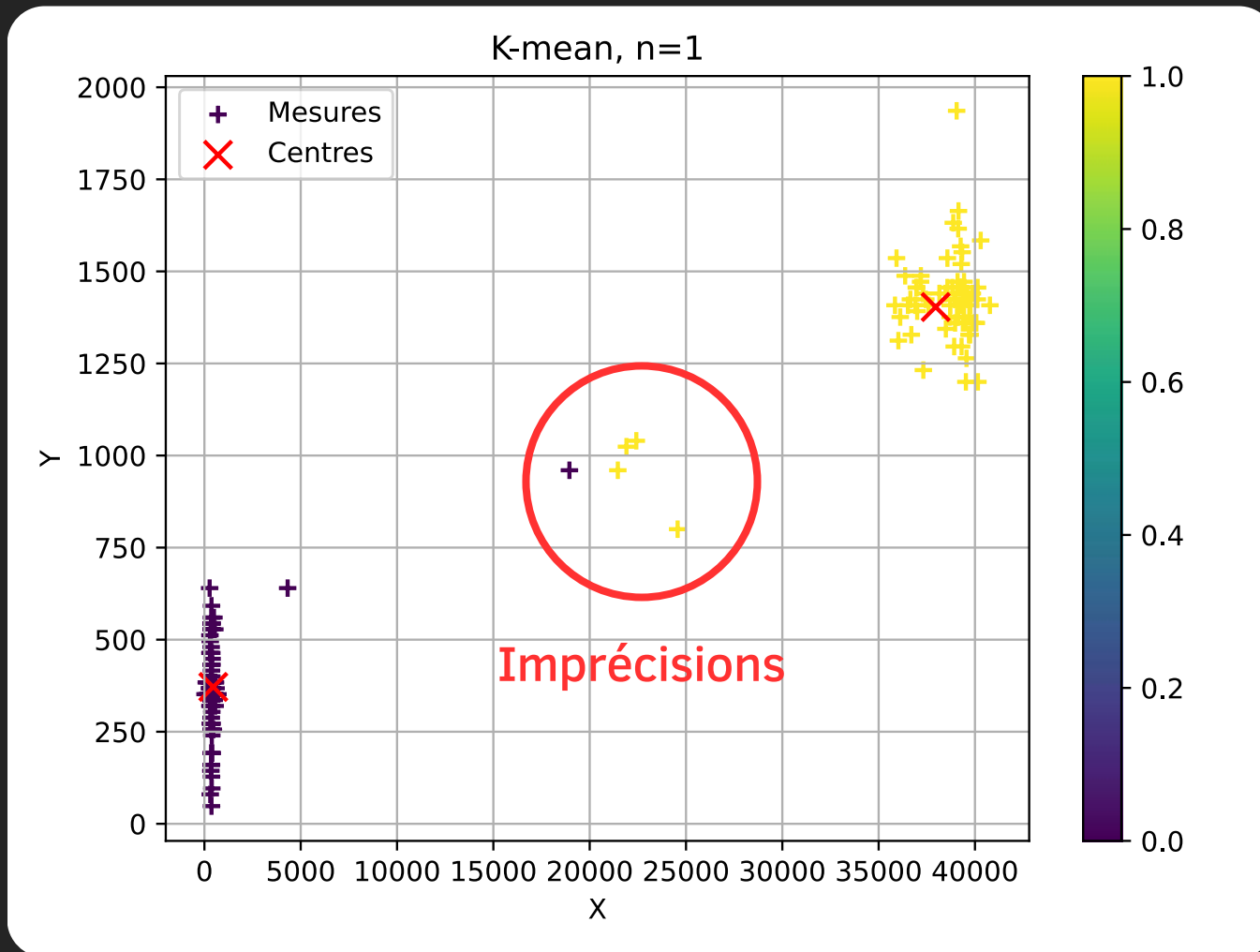
K-mean, 30m, 1



K-mean, 30m, 2



K-mean, 30m, 3



Présence d'imprécisions

TRAITEMENT DES ERREURS

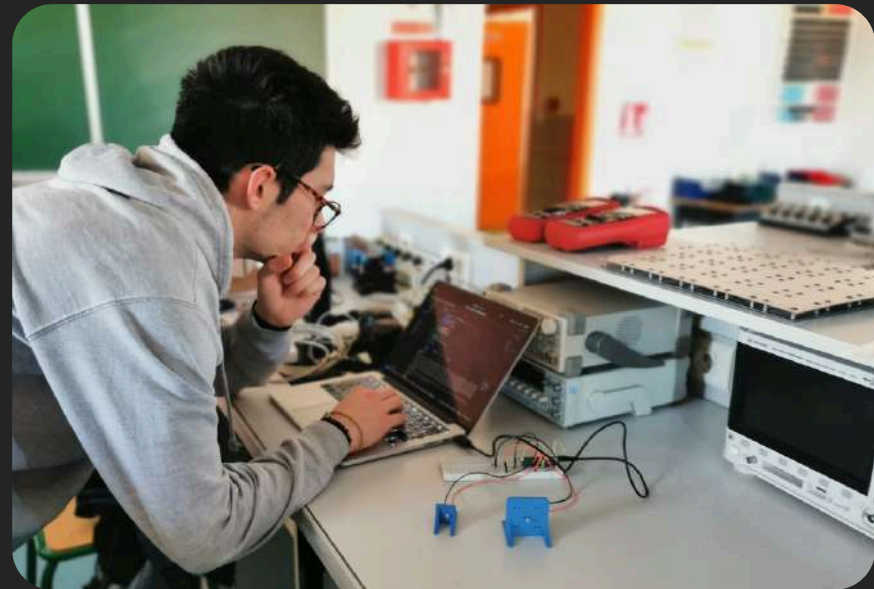
Critère de Shannon :

1. Filtrage par moyenne glissante

Code correcteur de Hamming :

1. Ajout de bits de parité
2. Vérification de la cohérence
3. Correction

19



Communication par laser à grande distance

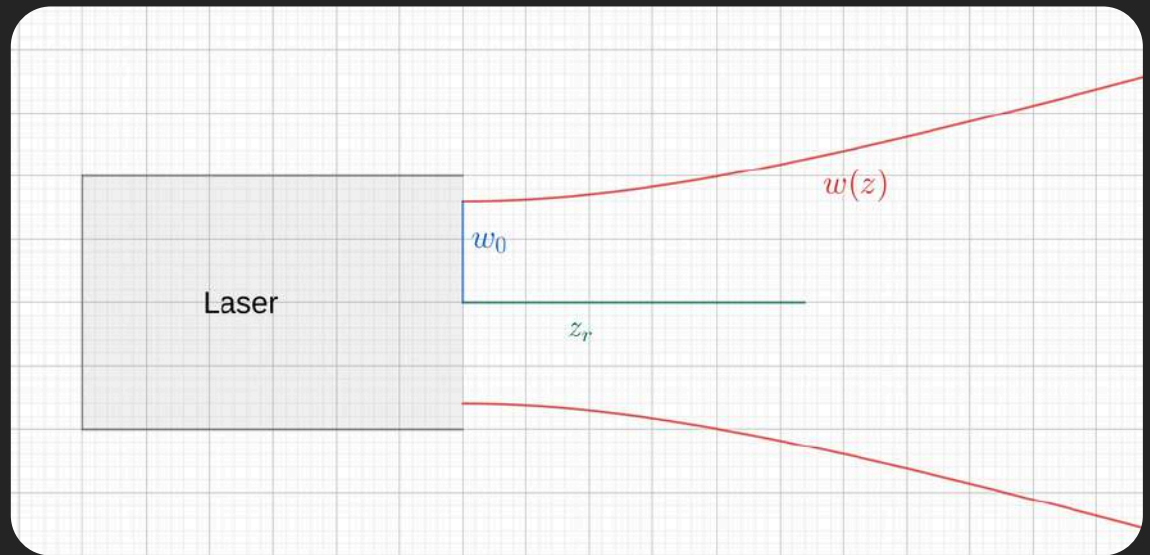
```

[336, 352], [352, 352], [384, 352], [352, 352],
[352, 288], [288, 352], [336, 352], [352, 336],
### K-mean :
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1,
0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1,
0, 0, 0, 0]
### Hamming :
[0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0,
0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0]
### Résultat :
tipe laser

```

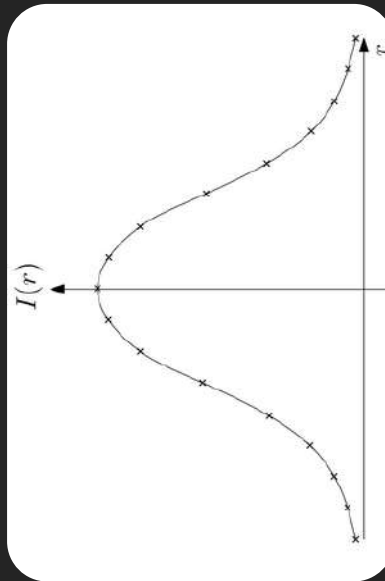
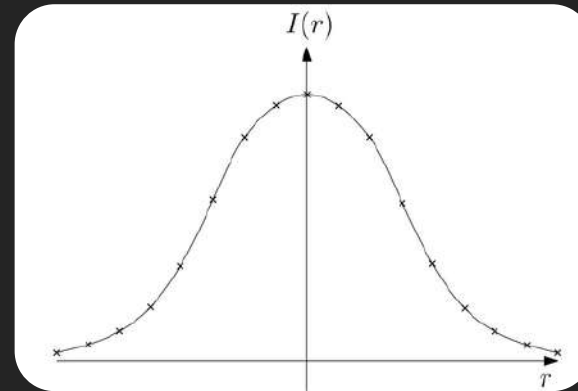
DIODE LASER

Modèle du faisceau gaussien
Étude de l'évolution de l'irradiance
lumineuse



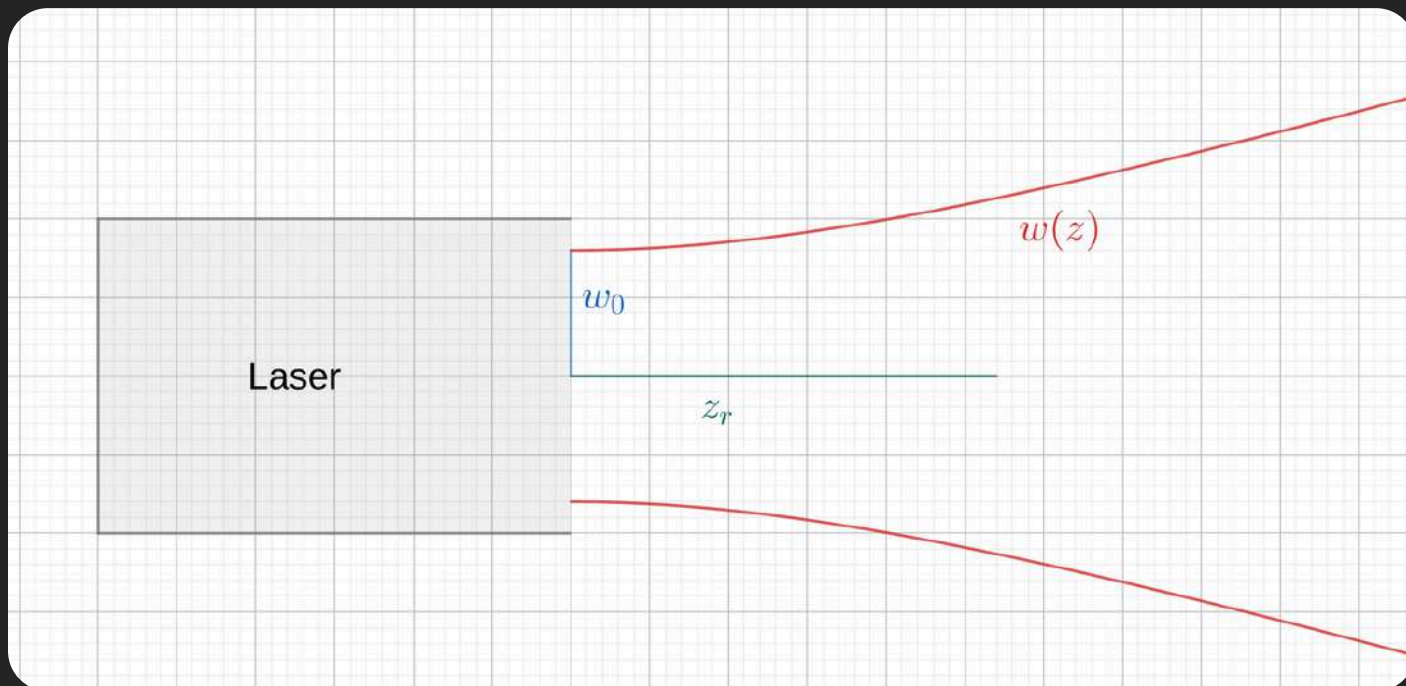
Laser dans l'air

$$I : (z, r) \mapsto I_0 \frac{w_0^2}{w^2(z)} \cdot \exp\left(\frac{-2r^2}{w^2(z)}\right)$$



Irradiance du faisceau Gaussien

$$w : z \mapsto w_0 \sqrt{1 + \left(\frac{z}{z_r}\right)^2}$$

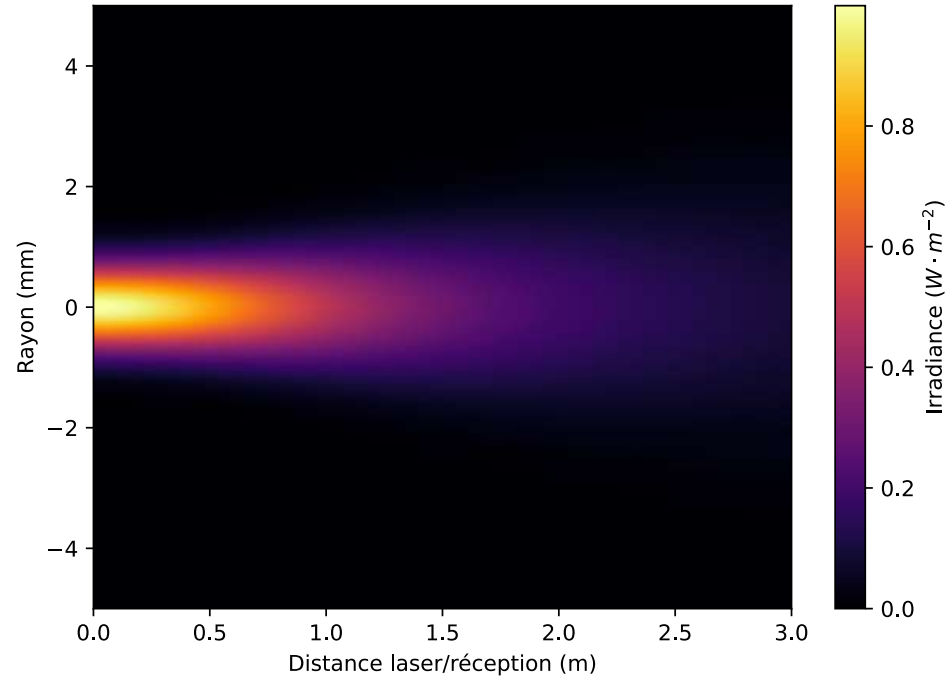


Divergence et longueur de Rayleigh

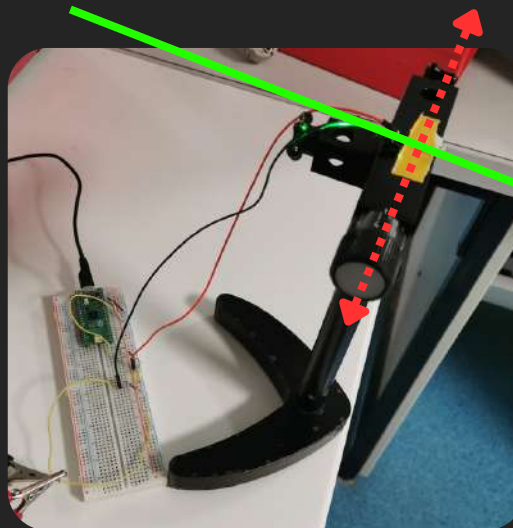
25



Irradiance du faisceau laser gaussien (valeurs arbitraires)

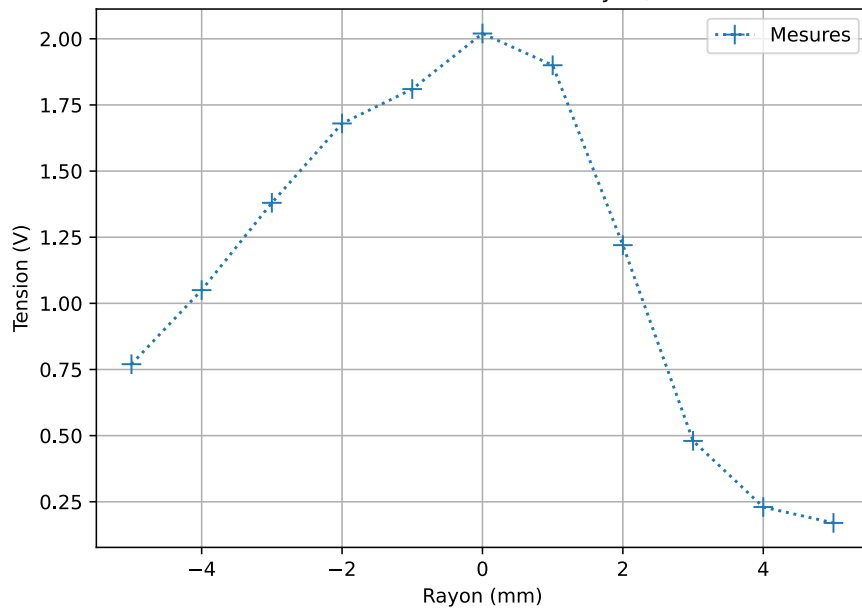
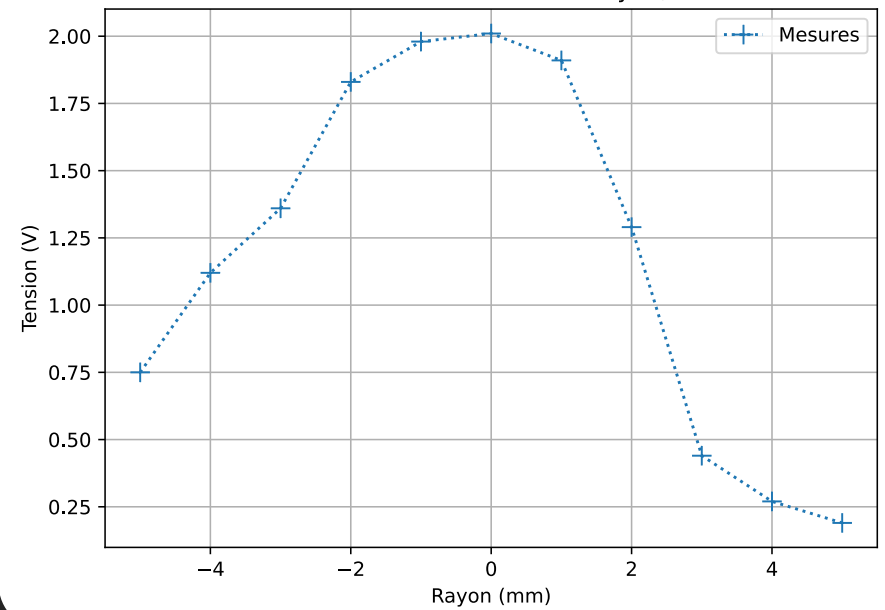


Communication par laser à grande distance

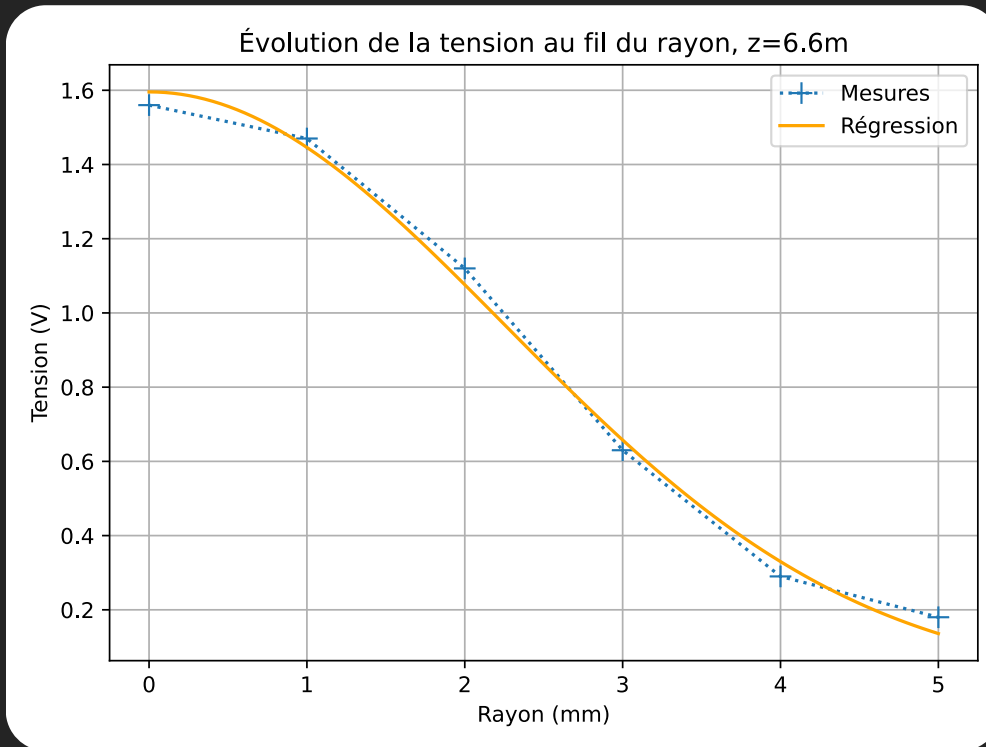


Étude du faisceau

- Distance z fixée
- Tension aux bornes d'une résistance

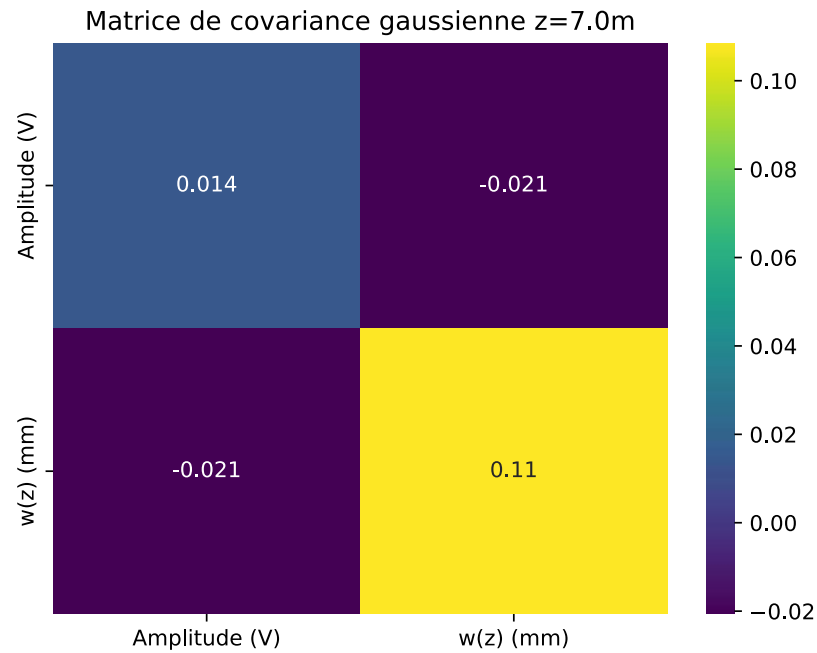
Évolution de la tension au fil du rayon, $z=3.93\text{m}$ Évolution de la tension au fil du rayon, $z=4.5\text{m}$ 

Première approche



$$U(z, r) = A(z) \cdot \exp\left(\frac{-2r^2}{B^2(z)}\right)$$

Seconde approche

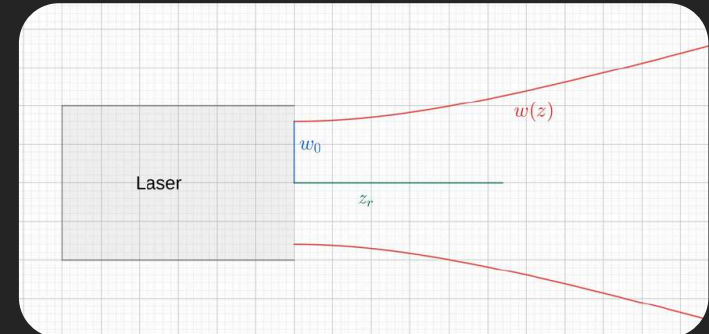
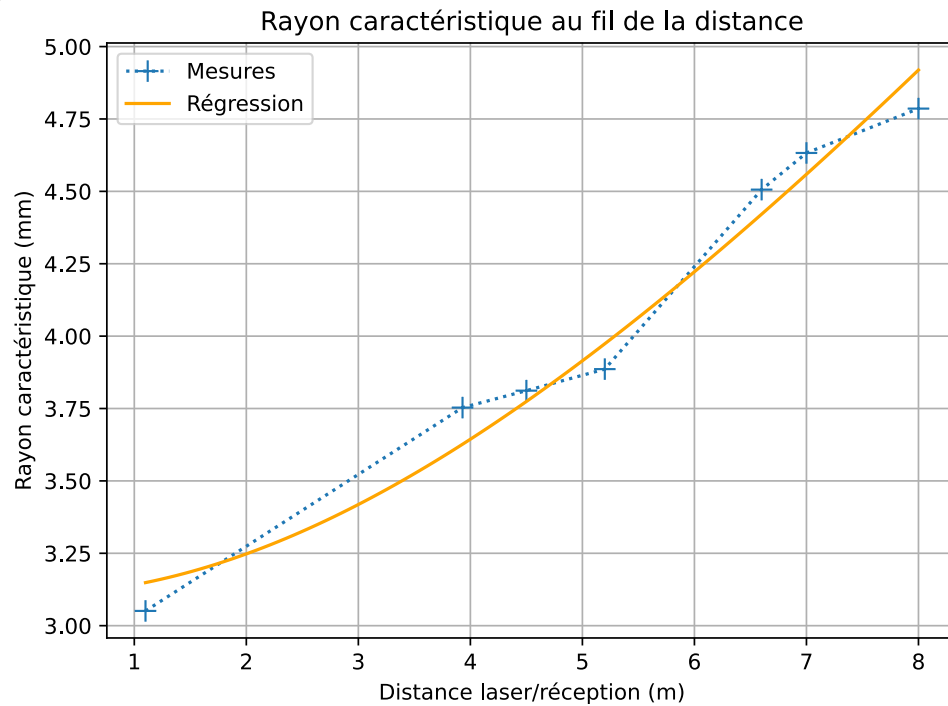


- $\sigma(A) \sim 10^{-1}\text{V}$
- $\sigma(B) \sim 10^{-1}\text{mm}$
- $\text{cov}(A, B) < 0$

$$I : (z, r) \mapsto I_0 \frac{w_0^2}{w^2(z)} \cdot \exp\left(\frac{-2r^2}{w^2(z)}\right)$$

$$A = \frac{cste}{B^2}$$

Matrice de covariance (pire cas)



- $w_0 = 3,1\text{mm}$, $\sigma(w_0) \simeq 0,1\text{mm}$
- $z_r = 6,5\text{m}$, $\sigma(z_r) \simeq 0,4\text{m}$

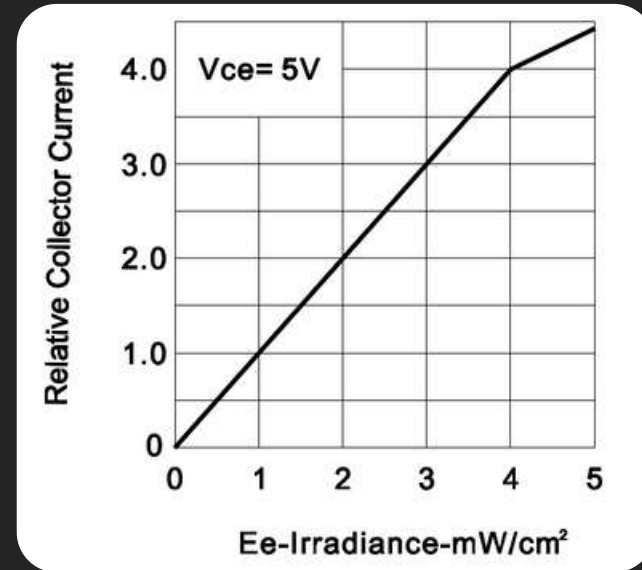
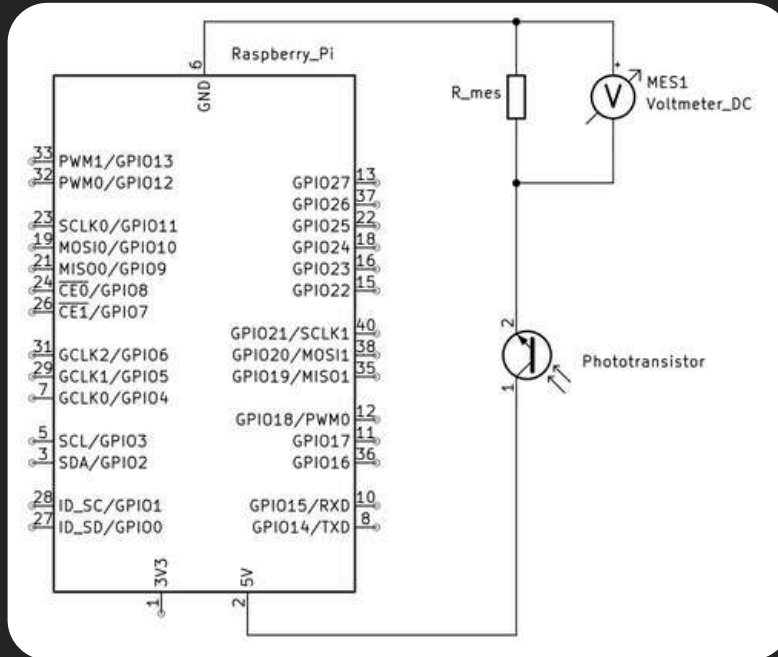
$$z_r = \frac{\pi w_0^2}{\lambda}$$

$$z_r = 56\text{m}$$

Distance de Rayleigh

CONCLUSION

- Système fonctionnel
- Adaptation à la luminosité
- Correction des erreurs
- Débit peu élevé (Python)



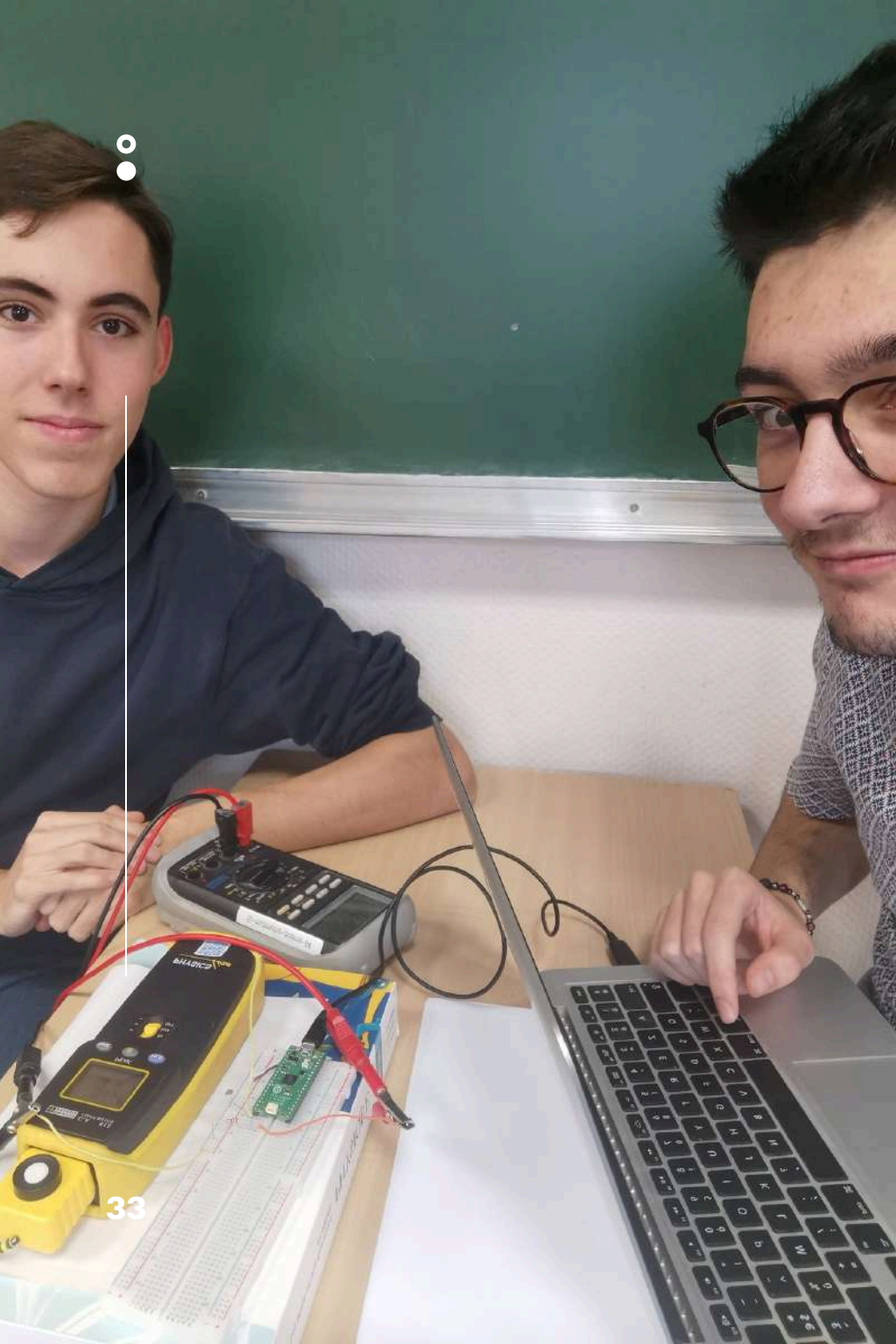
$$i = k_{capt} \cdot I$$

$$U = Rk_{capt} \cdot I$$

Annexe : Protocole faisceau Gaussien

ANNEXE : PHOTORÉSISTANCE

- Facile d'utilisation
- Non linéaire



Tracé de la caractéristique

PROTOCOLE :

- Pont diviseur de tension
- Éclairement constant

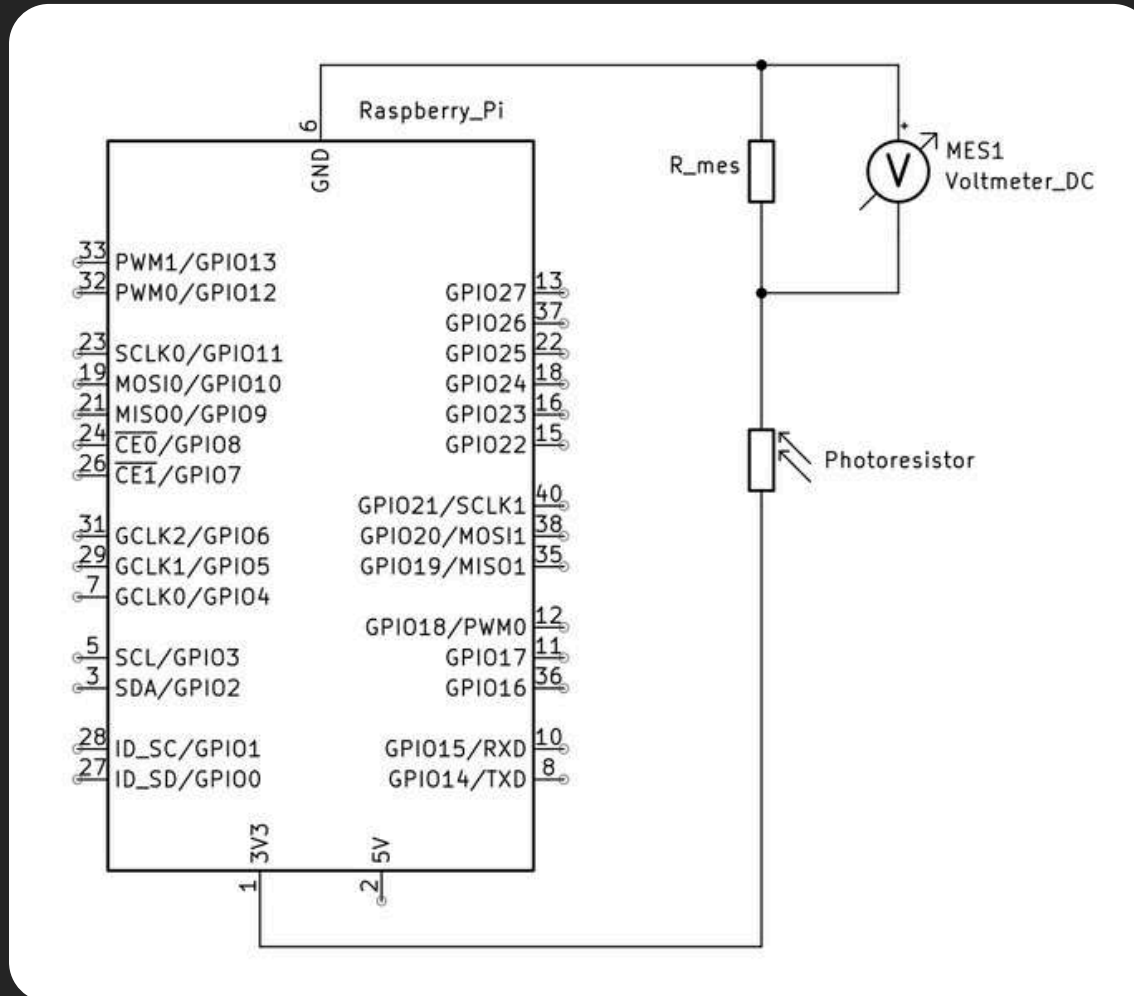
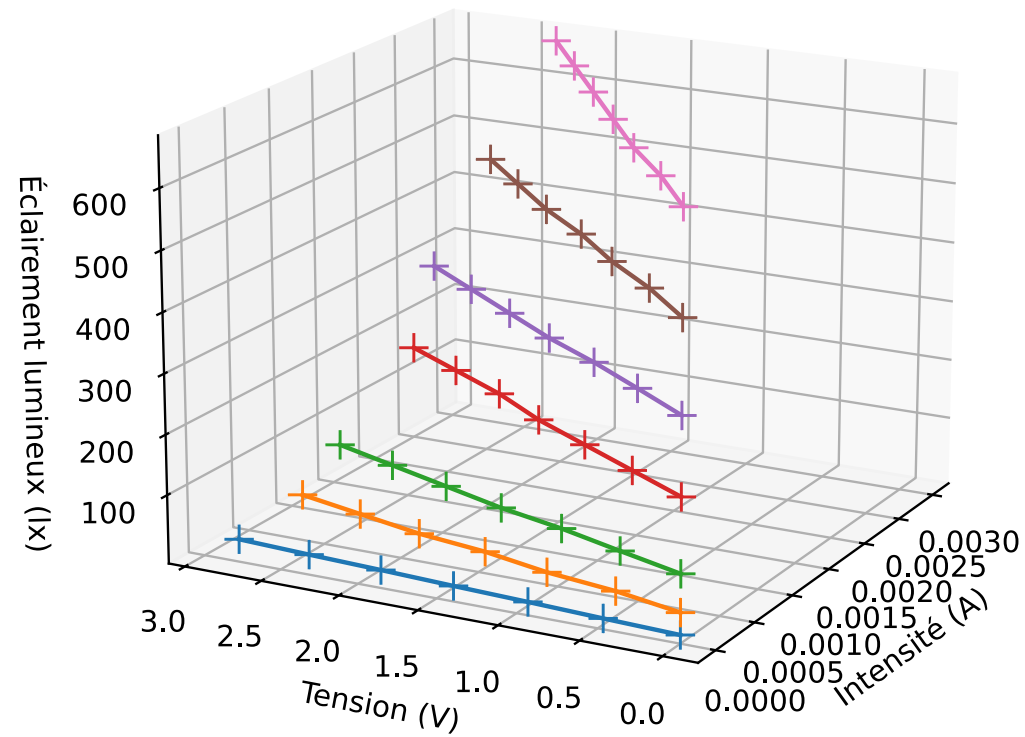
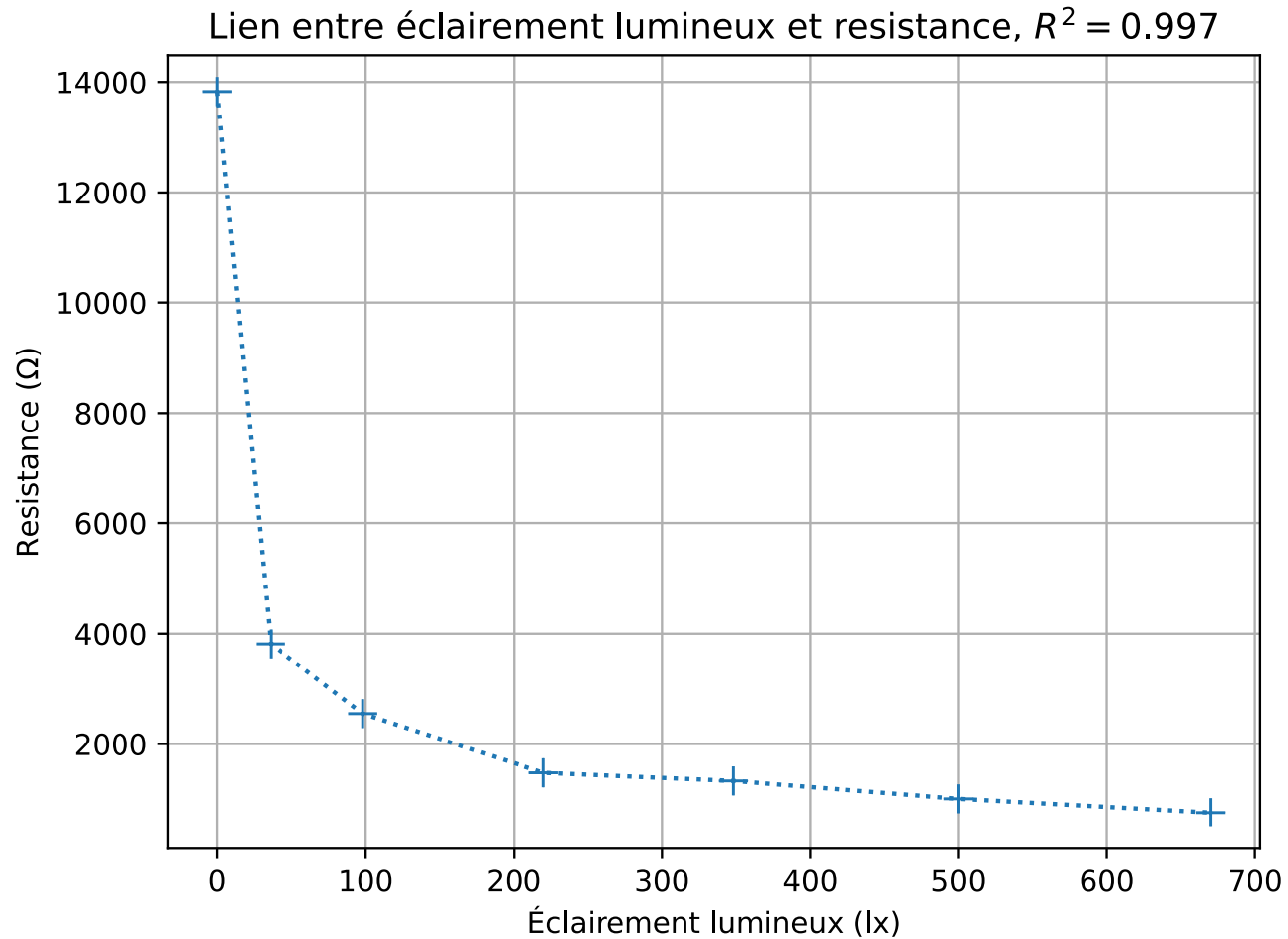


Schéma électrique

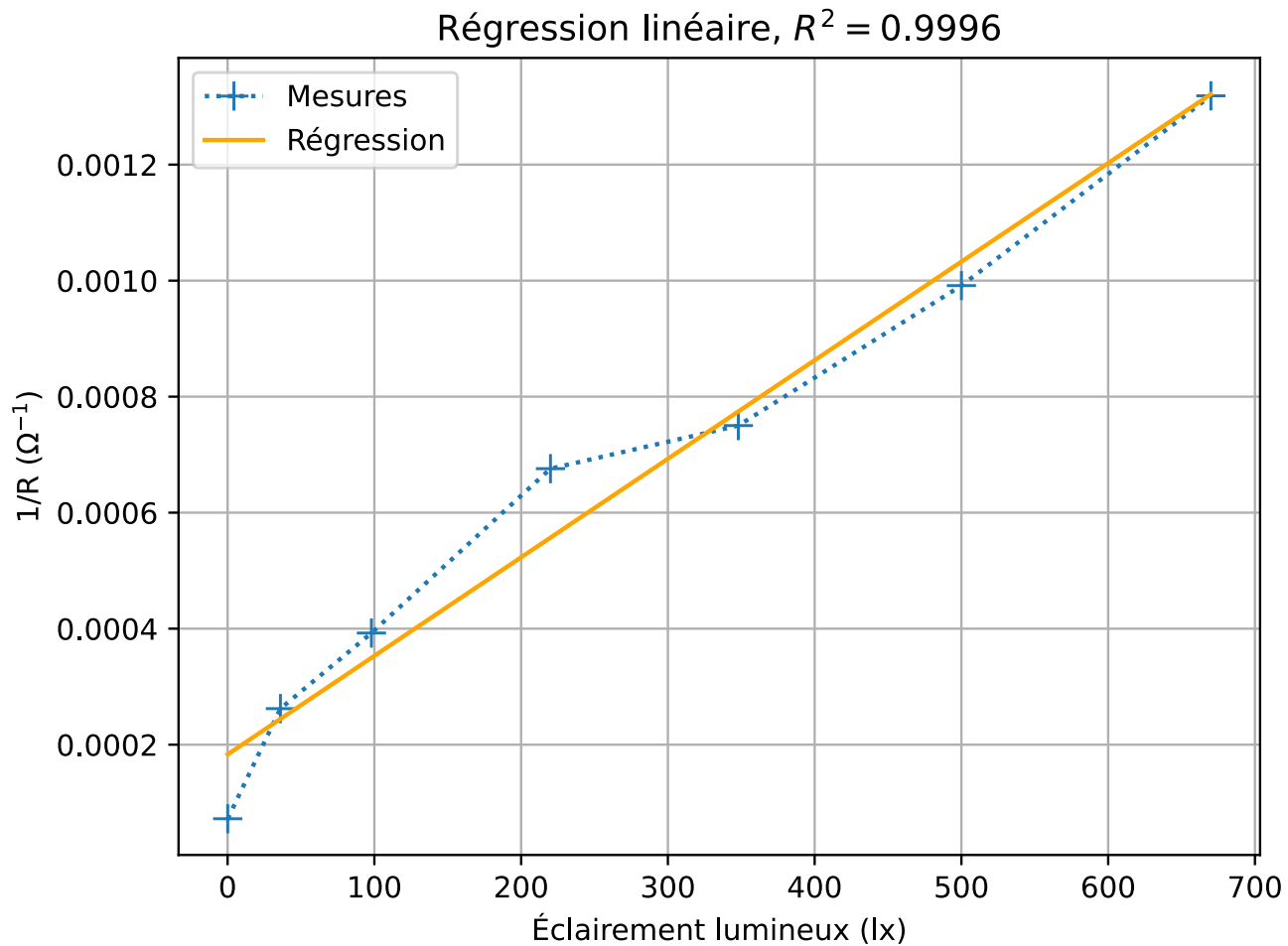
Lien entre intensité, tension et éclairement lumineux



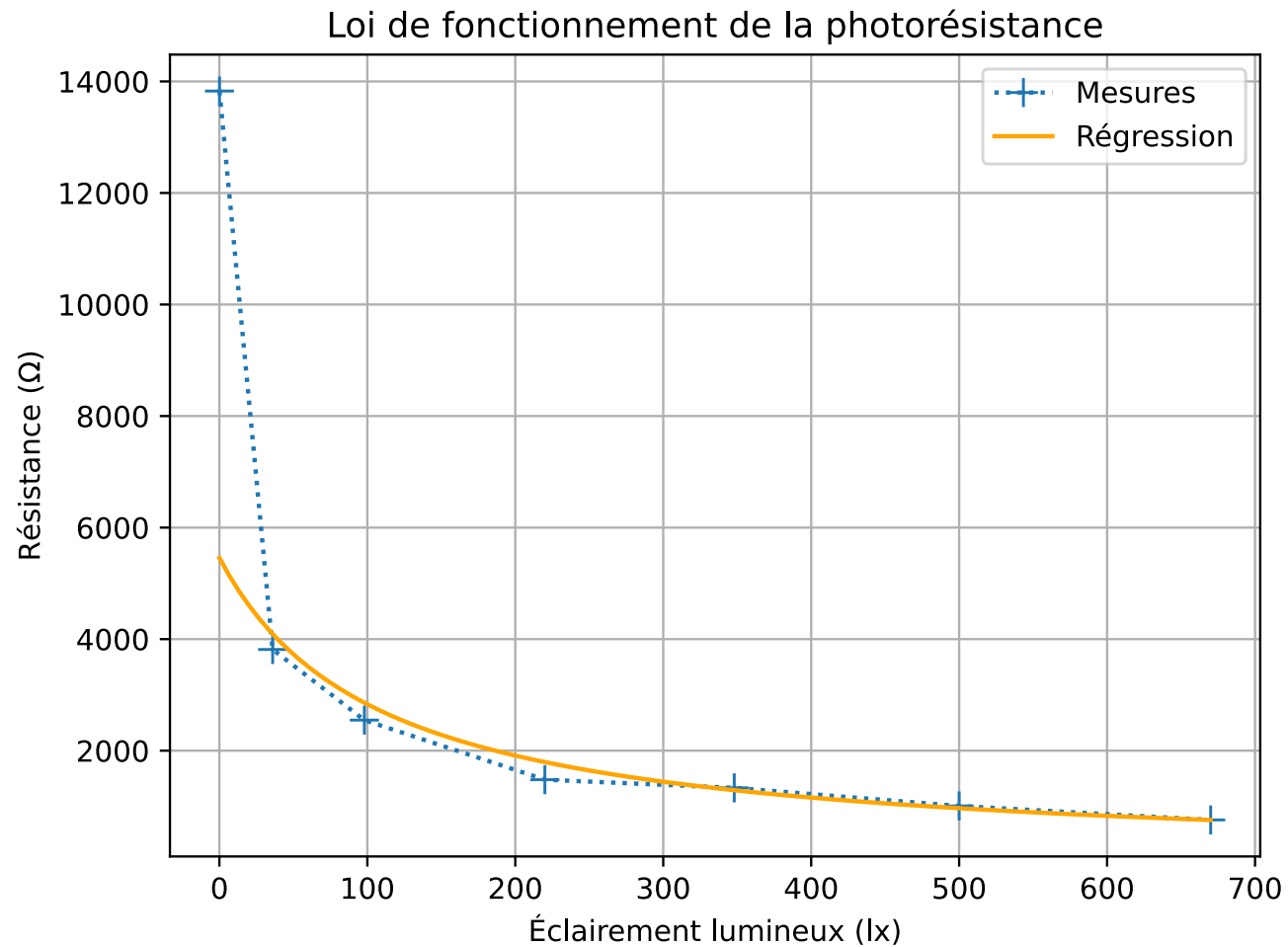
Données brutes



Régressions linéaires



Fonction inverse + régression linéaire



Transformée inverse

ANNEXE : HAMMING

ENCODAGE

1. Détection des “1”
2. Encodage de leur position
3. Ajout de bits

Émission : “1001”

Position	7	6	5	4	3	2	1
Message	1	0	0	p2	1	p1	p0

ENCODAGE

Position	7	6	5	4	3	2	1
Message	1	0	0	p2	1	p1	p0

Emplacements des 1	4	2	1
3	0	1	1
7	1	1	1

ENCODAGE

Emplacements des 1	4	2	1
3	0	1	1
7	1	1	1
	p4 = 1	p2 = 0	p1 = 0

Émission : “100**1100**”

DÉCODAGE

1. Détection des “1”
2. Encodage de leur position
3. Cohérence des bits de parité

Émission : “1001100”

Réception : “1101100”

Position	7	6	5	4	3	2	1
Message	1	1	0	1	1	0	0

DÉCODAGE

Emplacements des 1	4	2	1
3	0	1	1
4	1	0	0
6	1	1	0
7	1	1	1
Parité ?	x	x	v

Erreur détectée, position $2+4 = 6$

DÉCODAGE

Position	7	6	5	4	3	2	1
Message envoyé	1	0	0	1	1	0	0
Message reçu	1	1	0	1	1	0	0
Message corrigé	1	0	0	1	1	0	0

ANNEXES : CODES

```
def format_bis(entier, nb_bits):  
    """  
    Convertit un entier en binaire sur un nombre fixe de bits.  
  
    Args:  
        entier (int): L'entier à convertir.  
        nb_bits (int): Le nombre de bits pour la représentation binaire.  
  
    Returns:  
        str: La représentation binaire de l'entier sur le nombre de bits spécifié.  
    """  
    # Convertit l'entier en binaire  
    binaire = ""  
    while entier > 0:  
        binaire = str(entier % 2) + binaire  
        entier //= 2  
  
    # Ajoute des zéros de remplissage si nécessaire  
    while len(binaire) < nb_bits:  
        binaire = "0" + binaire  
  
    # Tronque la chaîne si elle est trop longue  
    if len(binaire) > nb_bits:  
        binaire = binaire[-nb_bits:]  
  
    return binaire  
  
led = Pin('LED', Pin.OUT)  
laser = Pin(16, Pin.OUT)
```

Annexe : Émission, 1

```
alphabet = {'a':1,'b':2,'c':3,'d':4,'e':5,'f':6,'g':7,'h':8,'i':9,'j':10,'k':11,'l':12,
            'm':13,'n':14,'o':15,'p':16,'q':17,'r':18,'s':19,'t':20,'u':21,'v':22,'w':23,
            'x':24,'y':25,'z':26,' ':27,',':28,'.':29,'!':30,'?':31,'"':32}

# Inversion du dictionnaire pour obtenir une correspondance valeur -> lettre
tebahpla = {i: j for j, i in alphabet.items()}

def text_to_binary(message): #Entrée : str
    # Stocker le résultat
    binary_message = ""
    # Parcourir chaque caractère
    for c in message:
        if c in alphabet:
            # On attribue une valeur au caractère
            value = alphabet[c]
            # Convertir la valeur en binaire sur 5 bits
            binary_c = format_bis(value, 5)
            binary_message += binary_c
        else:
            print("Caractère '{c}' non supporté dans l'alphabet personnalisé.")
    return binary_message
```

Annexe : Émission, 2


```

def hamming_encode(str):
    reverse_str=str[::-1]
    n=len(str)
    k = 0
    while (2 ** k) < (n + k + 1): #Calcul du nombre k de bits de contrôle
        k += 1
    result = [0 for u in range (n+k)]
    i = 0
    for x in range(1,n+k+1): #Ajoute les bits d'informations aux bons emplacements
        if x & (x-1) != 0: #Vérifie si x n'est pas une puissance de 2, & est l'opérateur ET logique qui compare en binaire x et x-1.
            #L'idée avec x et x-1 est que si x est une puissance de 2, son binaire ne possède qu'un 1, x-1 inverse alors tous les bits.
            #Donc aucun des bits ne sera commun à x et x-1 donc "l'indicatrice" & ne s'allume jamais et on a 0.
            result[x-1]=reverse_str[i]
            i+=1
    bits_1=[]
    bin_bits_1=[]
    for x in range(n+k):
        if result[x] == '1':
            bits_1.append(x+1) #stock la position de tous les bits valant 1
    for x in bits_1:
        bin_bits_1.append(format_bis(x,k))
    parity=[0 for i in range(k)]
    for x in bin_bits_1:
        for i in range(len(x)):
            parity[i]+=int(x[::-1][i]) #Calcule la parité de chaque bit de contrôle
    for u in range(len(parity)):
        if parity[u]%2==1:
            result[(2**u)-1]='1' #Ajuste la valeur du bit de contrôle selon sa parité
    final=''
    for y in result[::-1]:
        final+=y
    return(final)

```

Annexe : Émission, 3

```
def emission(binary,f):  
    binary2="1"+str(binary)+"1"  
    for x in binary2:  
        if x == "1":  
            laser.high()  
            sleep(1/f)  
            laser.low()  
        else:  
            sleep(1/f)  
  
def text_to_light(message,f):  
    return(emission(text_to_binary(message),f))  
  
def text_to_light_hamming(message,f):  
    return(emission(hamming_encode(text_to_binary(message)),f))
```

Annexe : Émission, 4

```
photo = ADC(Pin(26))
base = ADC(Pin(27))
switch = Pin(0, Pin.IN)

f_emit = 7
bit_duration = 3
f_recep = f_emit*bit_duration

data = []

while switch.value() == 0:
    pass

while switch.value() == 1:
    data.append([photo.read_u16(), base.read_u16()])
    sleep(1/f_recep)
```

```
centers = [[min(point[0] for point in data), min(point[1] for point in data)], [max(point[0] for point in data), max(point[1] for point in data)]]

centers, labels = knn.k_means(data, centers, n = 100)
bits = error_detection.clean_signal(labels, bit_duration)
bits = converters.list_to_str(bits)

### sans hamming
message = converters.binary_to_text(bits)

### avec hamming
hamminged_message = hamming.hamming_decode(bits)

message = converters.binary_to_text(hamminged_message)
print(message)
```

Annexe : Réception, 1

```
from math import sqrt

def distance(p1, p2):
    return sqrt((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2)

def k_means(data, centers, n=100):
    n_points = len(data)
    n_clusters = len(centers)

    labels = [0]*n_points

    for _ in range(n):
        # update labels
        new_labels = []
        for i in range(n_points):
            # compute all distances, attribute new centers
            distances = [distance(data[i], center) for center in centers]
            new_labels.append(distances.index(min(distances)))

        # update clusters
        new_centers = []
        for k in range(n_clusters):
            cluster_points = [data[i] for i in range(n_points) if new_labels[i] == k]
            if cluster_points != []: # for not empty clusters
                mean_x = sum(point[0] for point in cluster_points) / len(cluster_points)
                mean_y = sum(point[1] for point in cluster_points) / len(cluster_points)
                new_centers.append([mean_x, mean_y])
            else: # for empty clusters
                new_centers.append(centers[k])

        # check for convergence
        if new_centers == centers:
            break
        centers = new_centers
        labels = new_labels

    return centers, labels
```

Annexe : Réception, 2

```
def clean_signal(bits, bit_duration):
    output = rmv_first_zeros(bits, bit_duration)
    output = filtering(output, bit_duration)
    output = rmv_last_zeros(output)

    output = output[1:-1]

    print("clean_signal")
    print(output)

    return output

def rmv_first_zeros(bits, bit_duration):
    init = True

    while init :
        while bits[0] == 0:
            bits.pop(0)

        mean = round(sum(bits[:bit_duration])/bit_duration)

        if mean == 1:
            init = False

        else:
            bits.pop(0)

    return bits
```

Annexe : Réception, 3

```
def filtering(bits, bit_duration):
    n = len(bits)
    groups = [bits[i:i+bit_duration] for i in range(0, n, bit_duration)]
    output = []

    for group in groups:
        mean = round(sum(group)/bit_duration)
        output.append(mean)

    return output

def filtering2(bits, bit_duration):
    n = len(bits)
    groups = [bits[i:i+bit_duration] for i in range(0, n, bit_duration)]
    output = []

    for group in groups:
        mean = round(sum(group[1:-1])/(bit_duration-2))
        output.append(mean)

    return output

def rmv_last_zeros(bits):
    while bits[-1] == 0:
        bits.pop(-1)

    return bits
```

```
def hamming_decode(stri:str)->str:
    reverse_str=list(stri[::-1])
    n=len(stri)
    k = 0
    while (2 ** k) < (n + 1): #Cherche le nombre de bits de contrôle
        k += 1
    result=[]
    bits_1=[]
    bin_bits_1=[]
    for x in range(n):
        if reverse_str[x] == '1':
            bits_1.append(x+1) #stock la position de tous les bits valant 1
    for x in bits_1:
        bin_bits_1.append(format_bis(x,k))
    parity=[0 for i in range(k)]
    for x in bin_bits_1:
        for i in range(len(x)):
            parity[i]+=int(x[::-1][i]) #Calcule la parité de chaque bit de contrôle
    ok=0 #Tant que ok=0, aucune erreur détectée
    bit_error = 0
    for u in range(len(parity)):
        if parity[u]%2==1:
            ok+=1
            bit_error+=2**u
            index_bit_error=bit_error-1
    if ok == 0:
        print("Aucune erreur détectée")
    else:
        print("Erreur détectée au bit",bit_error)
        reverse_str[index_bit_error] = '0' if reverse_str[index_bit_error] == '1' else '1'
    for u in range(n):
        if u+1 & u != 0:
            result.append(reverse_str[u])
    final=''
    for y in result[::-1]:
        final+=y
    return final
```

Annexe : Réception, 5


```
def list_to_str(list:list) -> str:
    output = ""
    for i in list :
        output += str(i)
    return output

def binary_to_text(binary_message, tebahpla:dict=tebahpla) -> str:
    """Convert a "01" string to a string of ASCII characters."""
    if type(binary_message) != str:
        binary_message = list_to_str(binary_message)

    message = ""

    # parse the binary string into groups of 5 bits
    for i in range(0, len(binary_message), 5):
        bit = binary_message[i:i+5]

        # convert the 5 bits to an integer, convert the integer to a character
        try :
            message += tebahpla[int(bit, 2)]

        # if wrong char
        except KeyError:
            message += "*"

    return message
```

Annexe : Réception, 6


```
import numpy as np
import matplotlib.pyplot as plt

z_max = 3
r_max = 5
z_r = 1
w_0 = 1
I_0 = 1

z_list = np.linspace(0, z_max, 300)
r_list = np.linspace(-r_max, r_max, 300)
Z, R = np.meshgrid(z_list, r_list)

def div(z, A, B):
    return A * np.sqrt(1 + (z / B) ** 2)

def gaussian(r, A, B):
    return A * np.exp(-2 * r**2 / B**2)

def radiance(z, r):
    w = div(z, w_0, z_r)
    return gaussian(r, I_0 * (w_0 / w)**2, w)

I = radiance(Z, R)

plt.figure('Irradiance faisceau laser gaussien')
extent = [z_list.min(), z_list.max(), r_list.min(), r_list.max()]
plt.imshow(I, extent=extent, aspect='auto', cmap='inferno')
plt.xlabel('Distance laser/réception (m)')
plt.ylabel('Rayon (mm)')
plt.title('Irradiance du faisceau laser gaussien (valeurs arbitraires)')
plt.colorbar(label=r'Irradiance ($W \cdot m^{-2}$)')
plt.tight_layout()
plt.show()
```

Annexe : Topographie de l'irradiance

```
def gauss(x, A, B):  
    y = A*np.exp(-2*x**2/(B**2))  
    return y  
  
def div(x,A,B):  
    y = A*np.sqrt(1+(x/B)**2)  
    return y  
  
def get_r_v(z):  
    global cur  
    cur.execute(f'SELECT data.Radius, data.Voltage FROM data WHERE data.distance = {z} AND data.radius >= {0} ORDER BY data.Radius')  
    rows = cur.fetchall()  
    r = np.array([row[0] for row in rows])  
    v = np.array([row[1] for row in rows])  
    return np.array(r),np.array(v)
```

```
conn = sqlite3.connect('V2/data2.db')  
cur = conn.cursor()  
  
z_list = cur.execute('SELECT DISTINCT data.distance FROM data ORDER BY data.distance').fetchall()  
z_list = [z[0] for z in z_list]  
W_list = []
```

Annexe : Caractéristique laser, 1

```
for z in z_list:
    R,V = get_r_v(z)
    parameters,covariance = curve_fit(gauss, R, V)
    fit_A = parameters[0]
    fit_B = parameters[1]

    W_list.append(fit_B)

    fit_r = np.linspace(min(R), max(R), 100)
    fit_v = gauss(fit_r, fit_A, fit_B)

    plt.figure(f'Matrice de covariance gaussienne z={z}m')
    plt.title(f'Matrice de covariance gaussienne z={z}m')
    sns.heatmap(covariance, annot=True, xticklabels=['Amplitude (V)', 'w(z) (nm)'], yticklabels=['Amplitude (V)', 'w(z) (nm)'], cmap='viridis')
    plt.figure(f'Tension au fil du rayon (z={z})')
    plt.plot(R,V, marker='+', linestyle='dotted', markersize=10, label='Mesures')
    plt.plot(fit_r, fit_v, color='orange', label='Régression')
    plt.grid()
    plt.xlabel('Rayon (mm)')
    plt.ylabel('Tension (V)')
    plt.title(f'Évolution de la tension au fil du rayon, z={z}m')
    plt.tight_layout()
    plt.legend()
```

```
z_list, W_list = np.array(z_list), np.array(W_list)

parameters, covariance = curve_fit(div, z_list, W_list)
fit_A = parameters[0]
fit_B = parameters[1]

plt.figure('Matrice de covariance rayon caracteristique')
plt.title('Matrice de covariance rayon caracteristique')
sns.heatmap(covariance, annot=True, xticklabels=[r'$W_0$ (mm)', r'$z_r$ (m)'], yticklabels=[r'$W_0$ (mm)', r'$z_r$ (m)'], cmap='viridis')

print(f'W0 = {fit_A}, zr = {fit_B}')

fit_z = np.linspace(min(z_list), max(z_list), 100)
fit_w = div(fit_z, fit_A, fit_B)

plt.figure('Rayon caracteristique distance')
plt.plot(z_list, W_list, marker='+', linestyle='dotted', markersize=10, label='Mesures')
plt.plot(fit_z, fit_w, color='orange', label='Régression')
plt.xlabel('Distance laser/réception (m)')
plt.ylabel('Rayon caracteristique (mm)')
plt.title('Rayon caracteristique au fil de la distance')
plt.legend()
plt.tight_layout()
plt.grid()
plt.show()
```

Annexe : Caractéristique laser, 3

```

R_mes = 215

I_global = []
U_photo_global = []
i_mes_global = []
E = np.array([0,0.49,0.98,1.47,1.96,2.45,2.94])

def store(I,U_R):
    U_photo = E - U_R
    i_mes = U_R / R_mes

    I_global.append(I)
    U_photo_global.append(U_photo)
    i_mes_global.append(i_mes)

I_global = np.array(I_global)
U_photo_global = np.array(U_photo_global)
i_mes_global = np.array(i_mes_global)

```

```

fig = plt.figure('Données brutes')
ax = fig.add_subplot(projection='3d')

for i in range(len(I_global)) :
    ax.plot(i_mes_global[i],U_photo_global[i], I_global[i], marker = '+', markersize=10)

plt.title('Lien entre intensité, tension et éclairement lumineux')
ax.set_xlabel('Intensité (A)')
ax.set_ylabel('Tension (V)')
ax.set_zlabel('Éclairement lumineux (lx)')
plt.tight_layout()
ax.grid()

```

```

R = []
R_square = []

for i in range(len(I_global)) :
    vec = np.polyfit(i_mes_global[i], U_photo_global[i],1)
    R.append(vec[0])
    U_reg = i_mes_global[i]*vec[0]
    err_quadratique = np.sum((U_photo_global[i] - U_reg)**2)
    ecart_type = np.sum((U_photo_global[i] - np.mean(U_photo_global[i]))**2)
    R_square.append(1-err_quadratique/ecart_type)

R = np.array(R)

min_R_square = min(R_square)
min_R_square = round(min_R_square,3)

fig = plt.figure('Régression linéaire 1')

plt.plot(I_global,R, marker='+', linestyle='dotted', markersize=10)
plt.title(r'Lien entre éclairement lumineux et resistance, $R^2 = $' + str(min_R_square))
plt.xlabel('Éclairement lumineux (lx)')
plt.ylabel(r'Resistance ($\Omega$)')
plt.tight_layout()
plt.grid()

```

Annexe : Caractéristique photorésistance, 1

```

new_R = 1/R

vec = np.polyfit(I_global,new_R,1)
I_lin = np.linspace(min(I_global),max(I_global),100)
reg = vec[0]*I_lin + vec[1]

err_quadratique = np.sum((U_photo_global[i] - U_reg)**2)
ecart_type = np.sum((U_photo_global[i] - np.mean(U_photo_global[i]))**2)

R_square = (1-err_quadratique/ecart_type)
R_square = round(R_square,4)

fig = plt.figure('Régression linéaire 2')
plt.plot(I_global,new_R, marker='+', linestyle='dotted', markersize=10, label='Mesures')
plt.plot(I_lin,reg, color='orange', label='Régression')
plt.title(r'Regression linéaire, $R^2 = $' + str(R_square))
plt.xlabel('Éclairement lumineux (lx)')
plt.ylabel(r'1/R ($\Omega^{-1}$)')
plt.legend()
plt.tight_layout()
plt.grid()

```

```

R_reg = 1/reg

fig = plt.figure('Loi fonctionnement photorésistance')
plt.plot(I_global,R, marker='+', linestyle='dotted', markersize=10, label='Mesures')
plt.plot(I_lin,R_reg, color='orange', label='Régression')
plt.title('Loi de fonctionnement de la photorésistance')
plt.xlabel('Éclairement lumineux (lx)')
plt.ylabel(r'Résistance ($\Omega$)')
plt.legend()
plt.tight_layout()
plt.grid()
plt.show()

```

Annexe : Caractéristique photorésistance, 2