Trabajo Practico de Simulacion
Codificacion de fuente y canal

Bautista Garcia - 03269/8

May 27, 2025

Contents

1 Codificacion de Canal

1.1 Introduccion

El objetivo detras de la **codificacion de canal** esta en usar sistemas de **gran dimension** para reducir la probabilidad de error (confundirse entre bits enviados y recibidos debido al ruido en el canal). Debido a que **codificar en frecuencia** resulta costoso (mayor ancho de banda necesario), codificamos en el **tiempo**. *Pensamos como que cada uso del canal representa una dimension* $\therefore$ *n usos de canal significara una palabra n-dimensional.*

1.1.1 Codigos de bloque lineales

Los codigos a utilizar son **codigos lineales**, en donde a cada palabra de fuente $\bar{u}$ se le asigna una palabra de codigo $\bar{v}$. Ademas vamos a utilizar una forma de **codificacion sistematica**, en donde los k bits de la palabra $\bar{u}$ se ven reflejados en los primeros k bits de la palabra de codigo v. Esta forma de codificacion permite una **decodificacion** mas sencilla (extraer k bits de palabra corregida).

$$\bar{u}G_{kxn} = \bar{u}\left[I_{kxk} \vdots P_{kx(n-k)}\right] = \bar{v}$$

A partir de la matriz **generadora** de codigos G definimos la **matriz de chequeo de paridad** H_T:

$$H^T = \begin{pmatrix} P_{kx(n-k)} \\ I_{(n-k)X(n-k)} \end{pmatrix}$$

1.1.2 Cota de Hamming

En el trabajo se menciona un codigo $(14, 10)$ $\therefore$ debemos encontrar las propiedades del mismo como $t_c \wedge d_{\text{min}}$. *En este caso asumi que se trabaja con el mejor codigo* $14, 10$.

El algoritmo para hallar sus propiedades consiste en probar iterativamente, para cada $t_c \in \mathbb{Z}$, la **cota de hamming**, hasta que la misma se deje de cumplir (ver Figure 1). Una vez que esta se deja de cumplir nos quedamos con el ultimo t_c que la cumplio:

$$\sum_{i=0}^{t_c} \binom{n}{i} \leq 2^{n-k}$$

```
LIM = 100
tc = 0

while(tc < LIM):
    lhs = [comb(n, idx) for idx in range(tc + 1)]
    if((res1:=np.sum(lhs)) > (res2:=2**(n - k))):
        tc -= 1
        break
    tc += 1

return tc, (2 * tc + 1)
```

Figure 1: Cota de Hamming

Conociendo t_c obtenemos $d_{\text{min}} = 2t_c + 1$.

1.1.3 Matriz Generadora y Matriz de Paridad

En la practica observamos que para armar ambas matrices necesarias para la codificacion y decodificacion del canal. Necesitamos que la matriz $P_{kx(n-k)}$ presente en $G \wedge H^T$, tenga k filas:

- Unicas: $P_i \neq P_j \ \forall \ (i, j \in [0, k-1])$
- $w_i = d_{\min} - 1$

Con estas consideraciones realize un algoritmo que a partir de $n, k \wedge d_{\min}$ dados, cree una matriz P, y consecuentemente $G \wedge H^T$.

Nota: *El algoritmo consiste en formar las primeras k combinaciones de palabras cuyo peso sea mayor o igual al $w_{\min}$ requerido. Implementado en* `comunicacion_digital/codificacion.py` *(ver Figure 2).*

```python
# Crear matriz generadora vacía
H = np.zeros((n, (n-k)), dtype=int)
P = np.zeros((k, (n-k)), dtype=int)
G = np.zeros((k, n), dtype=int)
# k filas de P con peso de hamming >= dmin-1
filas = []
min_w = dmin - 1
# Generar filas sistemáticamente por peso de hamming
for num_ones in range(min_w, (n - k) + 1):
    for positions in combinations(range(n - k), num_ones):
        row = np.zeros((n - k), dtype=int)
        row[list(positions)] = 1
        filas.append(row)
        if len(filas) == k:
            break
    if len(filas) == k:
        break
for i, row in enumerate(filas):
    H[i] = P[i] = row
# Resto de filas de H forman la matriz identidad
H[k:] = np.eye((n - k))
# Generamos la matriz G a partir de P
G[:, :k] = np.eye(k)
G[:, k:] = P
return H, G
```

Figure 2: Matriz Generadora

1.1.4 Canal simetrico binario

Se describe un **canal simetrico binario** (dado por la catedra), donde para una relacion $\frac{E_b}{N_0}$ dada y una **amplitud** A en BPSK, se determina cual debe ser la DEP N_0 a colocar en el canal, para que la simulacion tenga sentido (ver Figure 3).

```python
# Calcular energías
Es = A**2
Ebf = Es * n / k
N0 = Ebf / EbfN0
# Modulación BPSK (0 -> -A, 1 -> +A)
S = (2 * V - 1) * A
# Ruido AWGN para cada palabra código
noise = np.sqrt(N0 / 2) * (np.random.randn(*S.shape) + 1j * np.random.randn(*S.shape))
# Señal recibida
R = S + noise
# Demodulación (detección dura)
Rd = (np.real(R) > 0).astype(int)
return Rd
```

Figure 3: CSB

1.1.5 Simulacion

Una vez armadas $G \wedge H^T$ con los $n, k, d_{\min} \wedge (t_c \vee t_d)$ del mejor codigo $14, 10$ posible, la simulacion consiste en:

(i) Codificar una palabra de fuente:

$$\bar{u}G = \bar{v}$$

(ii) Enviar la palabra de codigo a traves de un canal **CSB**, donde ante la existencia de ruido, existe probabilidad de error en la transmision:

$$\text{CSB}(\bar{v}) = \bar{r} = \bar{v} + \bar{e}$$

(iii) Recibir la palabra $\bar{r}$ y **corregir** o **detectar** errores.

$$rH^T = (vH^T) + (eH^T) = eH^T$$

A eH^T se lo llama **sindrome**, si es 0 significa que no hubo error o que el error genero otra palabra de codigo distinta a la enviada. De lo contrario este sera una combinacion lineal de las filas de H^T donde hubo error:

$$eH^T = e_i H_i^T$$

(iv)
- Si se desea **corregir** errores, buscamos a que combinacion lineal de filas i pertenece el sindrome. Para luego invertir los $i-$**esimos** bits de la palabra de codigo recibida.
- Si se desea **detectar** errores, es suficiente descartar las palabras cuyo sindrome sea $\neq 0$. En la vida real, se pediria una retransmision de las mismas.

Estos pasos mencionados, se encuentran desarrollados dentro `comunicacion_digital/main.py` (ver Figure 4). Estos se ejecutan para cada $\frac{E_b}{N_0}$ a simular.

```python
P_ep = np.zeros_like(EbN0_c, dtype=float)
P_eb = np.zeros_like(EbN0_c, dtype=float)
ITERACIONES = 10

for i, EbN0 in enumerate(EbN0_c):
    H_t, G = matrizGeneradora(n, k, dmin) # Generar matrices de codigo
    EbfN0 = 10**(EbN0/10)                 # Eb/N0 [veces]
    P_eb_t = Q(np.sqrt(2 * EbfN0))        # Tasa de error de bit teorica (estimada)
    PALABRAS = int((10**2) * (1/P_eb_t))
    PALABRAS = PALABRAS if (PALABRAS > 1000000) else 1000000

    # Arrays para almacenar resultados de cada iteración
    P_ep_iter = np.zeros(ITERACIONES)
    P_eb_iter = np.zeros(ITERACIONES)

    for iter in range(ITERACIONES):
        U = random_U(PALABRAS, k)             # Palabras fuente random
        V = np.dot(U, G) % 2                  # Codificar palabras fuente
        R = canalCSB(n, k, A, EbfN0, V)       # Simular canal con ruido
        if MODO:
            Ve = corregir(R, H_t)             # Decodificar palabras recibidas
            Ue = Ve[:, :k]
            E = U != Ue
        else:
            detectados = detectar(R, H_t)
            Ve = np.delete(R, detectados, axis=0)
            U = np.delete(U, detectados, axis=0)
            Ue = Ve[:, :k]
            E = U != Ue
        e_p = (E.sum(axis=1) > 0).sum()
        P_ep_iter[iter] = e_p / E.shape[0] # Tasa de error de palabra
        e_b = E.sum()
        P_eb_iter[iter] = e_b / (k * E.shape[0]) # Tasa de error de bit
    # Promedio de iteraciones
    P_ep[i] = np.mean(P_ep_iter)
    P_eb[i] = np.mean(P_eb_iter)

    print(f"Promedio final - Eb/N0: {EbN0:.2f} dB, Tasa de error de bit: {P_eb[i]:.6f}")
return P_ep, P_eb
```

Figure 4: Simulacion de codificacion, CSB y decodificacion

1.1.6 Probabilidad de Error

Para calcular la probabilidad de error en los bits de fuente y canal, primero se extraian los k primeros bits de la matriz V_e (corregida o con errores detectados) obteniendo U_e, y se comparaba a esta misma con la matriz de palabras de fuente U. De esta forma obtenemos la matriz de error para calcular las **probabilidades de error** buscadas.

<u>Nota:</u> En el caso del codigo usado como **detector**, se eliminaban de $U \wedge U_e$ aquellas filas detectadas como error.

- ε_p: Aquellas filas en la matriz de errores con al menos 1 error.
- ε_b: Errores totales de la matriz de errores.

Esto fue implementado en `comunicacion_digital/main.py` (ver Figure 4 y Figure 5).

```
e_p = (E.sum(axis=1) > 0).sum()
P_ep[i] = e_p / E.shape[0] # Tasa de error de palabra
e_b = E.sum()
P_eb[i] = e_b / (k * E.shape[0]) # Tasa de error de bit
```

Figure 5: Probabilidad de error

1.2 Discusion

Habiendo realizado las simulaciones mencionadas se obtuvieron los siguientes resultados:

Propiedades del codigo:

$$t_c = 1 \wedge d_{\min} = 3$$

$$G_a = \left(\frac{k}{n}\right)\lfloor\frac{d_{\mathrm{hmin}}+1}{2}\rfloor = 1.54dB$$

1.2.1 Simulacion

Se simularon $\frac{E_{bf}}{N_0} \in [0, 10]$ con un paso de 0.33 (es decir 30 niveles discretos de $\frac{E_{bf}}{N_0}$), para los cuales, la cantidad de palabras a estimar fue variable dependiendo de la P_{ebf} teorica estimada. La regla fue:

$$\mathrm{PALABRAS} = \frac{10^2}{Q\left(\sqrt{2\frac{E_b}{N_0}}\right)}$$

Debido a que estas no fueron suficientes para el caso de el codigo usado como **detector**. Decidi hacer un promedio sobre 10 iteraciones con esta configuracion. Aun asi para $\frac{E_b}{N_0} \to 10$ en el **detector**, existieron casos donde no ocurrieror errores (es por esto que se ven saltos en el grafico sobre el final).

<u>Codigo Corrector:</u>

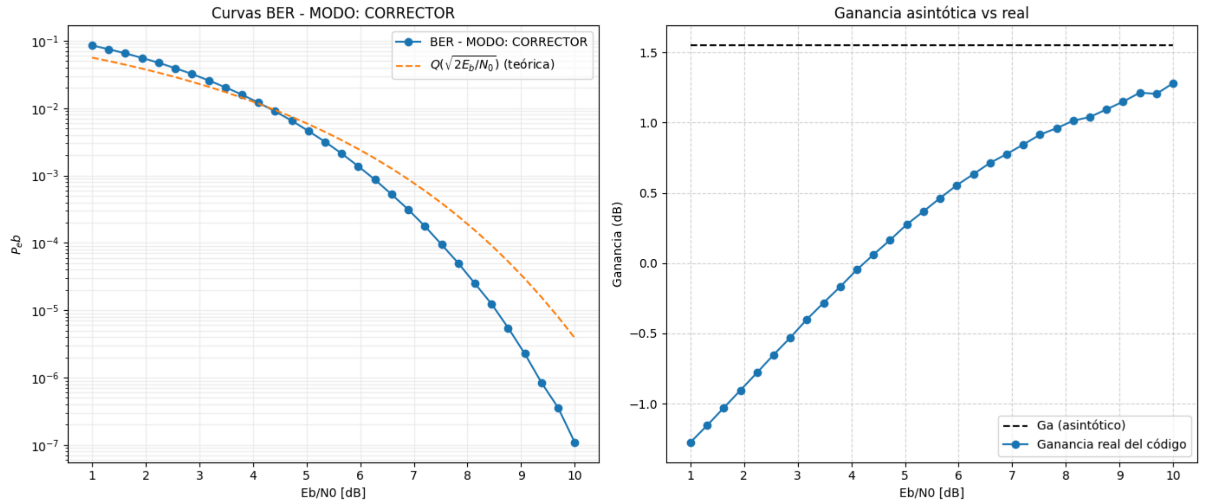

Figure 6: Curvas en MODO CORRECTOR

Eb/N0 [dB]	P_ep	P_eb	Gc (dB)	Ga (dB)
1.0	0.362212	0.0851482	-1.2679542086228262	1.5490195998574319
1.3103448275862069	0.322005	0.074908	-1.1522990372400805	1.5490195998574319
1.6206896551724137	0.282653	0.0651294	-1.0337396099366227	1.5490195998574319
1.9310344827586208	0.245114	0.0559099	-0.9130282908568415	1.5490195998574319
2.2413793103448274	0.208151	0.0470559	-0.7760738409543251	1.5490195998574319
2.5517241379310347	0.175011	0.0392216	-0.6528008052337988	1.5490195998574319
2.8620689655172415	0.144708	0.0321907	-0.5311439233901818	1.5490195998574319
3.1724137931034484	0.117225	0.0258406	-0.40084008537621774	1.5490195998574319
3.4827586206896552	0.093169	0.0203933	-0.2762500473432792	1.5490195998574319
3.793103448275862	0.073012	0.0158531	-0.16233324405114846	1.5490195998574319
4.103448275862069	0.055724	0.0120614	-0.05017110221988208	1.5490195998574319
4.413793103448276	0.041605	0.0089376	0.06424944528151322	1.5490195998574319
4.724137931034483	0.030243	0.0064703	0.17348740218132708	1.5490195998574319
5.0344827586206895	0.021427	0.0045701	0.27725574071646	1.5490195998574319
5.344827586206897	0.014775	0.0031324	0.37953999082580925	1.5490195998574319
5.655172413793103	0.010131	0.002141	0.4515277495653729	1.5490195998574319
5.9655172413793105	0.006577	0.0013946	0.5377893050432601	1.5490195998574319
6.275862068965518	0.004089	0.0008704	0.6281175685415601	1.5490195998574319
6.586206896551724	0.002501	0.000525	0.7124256218230647	1.5490195998574319
6.8965517241379315	0.001492	0.000311	0.7783301316081825	1.5490195998574319
7.206896551724138	0.000846	0.0001745	0.8505564164784358	1.5490195998574319
7.517241379310345	0.00044	9.16e-05	0.9324978619930011	1.5490195998574319
7.827586206896552	0.000245	4.92e-05	0.9711561970474847	1.5490195998574319
8.137931034482758	0.00013	2.94e-05	0.9310499389471847	1.5490195998574319
8.448275862068966	5.5e-05	1.08e-05	1.1044506999258807	1.5490195998574319
8.758620689655173	1.8e-05	4E-06	1.2278663869696906	1.5490195998574319
9.068965517241379	1.3e-05	2.8e-06	1.0636783779015762	1.5490195998574319
9.379310344827587	3E-06	7E-07	1.2803444259744623	1.5490195998574319
9.689655172413794	1E-06	2E-07	1.398418297936061	1.5490195998574319
10.0	1E-06	2E-07	1.0880734703498547	1.5490195998574319

Figure 7: Resultados

<u>Codigo Detector:</u>

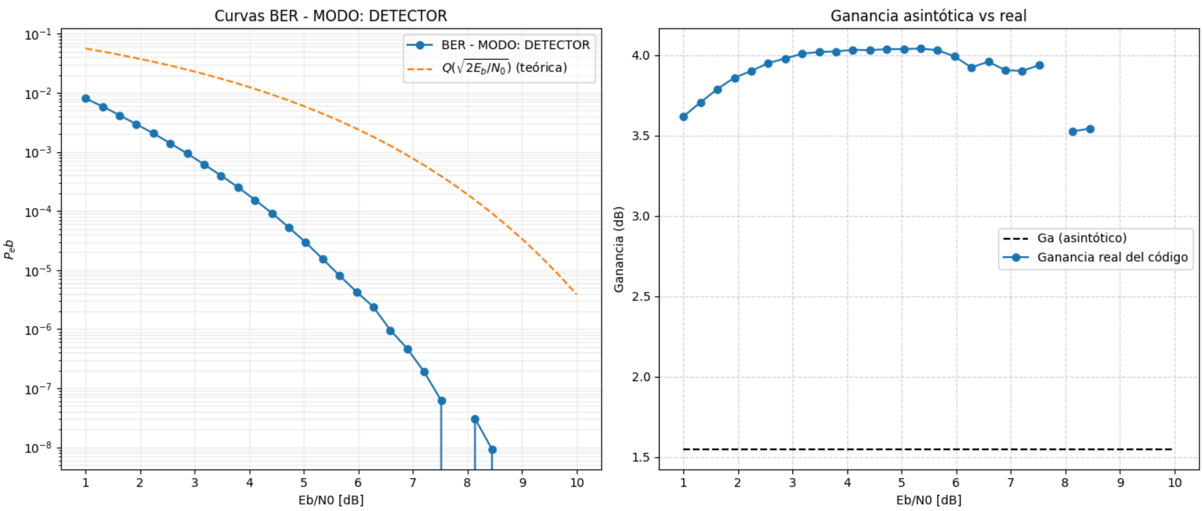

Figure 8: Curvas en MODO DETECTOR

Eb/N0 [dB]	P_ep	P_eb	Gc (dB)	Ga (dB)
1.0	0.034372358592150876	0.008041643562484768	3.619294178108511	1.5490195998574319
1.3103448275862069	0.025282056941221147	0.005875661095977394	3.705729411393571	1.5490195998574319
1.6206896551724137	0.018172660567868683	0.004189183503479417	3.789332813044455	1.5490195998574319
1.9310344827586208	0.012861692077895403	0.0029400236494833342	3.8591949210879095	1.5490195998574319
2.2413793103448274	0.009071304618395358	0.0020618494651671463	3.901541629990744	1.5490195998574319
2.5517241379310347	0.0061774126920368135	0.0013964346986886813	3.950415714929263	1.5490195998574319
2.8620689655172415	0.004175261130590606	0.0009397796965679498	3.9790161387318013	1.5490195998574319
3.1724137931034484	0.002755073678713589	0.0006135334179040517	4.008091033149712	1.5490195998574319
3.4827586206896552	0.001792595792530503	0.00039733285940985916	4.019875689588174	1.5490195998574319
3.793103448275862	0.0011431929975495446	0.00025275873273127735	4.0227277428494075	1.5490195998574319
4.103448275862069	0.0006958080837871581	0.0001538701463124587	4.033257073859824	1.5490195998574319
4.413793103448276	0.00042524193704581586	9.243575309948133e-05	4.030644013362606	1.5490195998574319
4.724137931034483	0.00024316995826064513	5.273074640267907e-05	4.036979771072484	1.5490195998574319
5.0344827586206895	0.00013419501428161545	2.9222766936445638e-05	4.0375786026680975	1.5490195998574319
5.344827586206897	7.172205078979338e-05	1.5445108890183347e-05	4.04093035111282	1.5490195998574319
5.655172413793103	3.686273806060955e-05	8.037473365133557e-06	4.030959417592953	1.5490195998574319
5.9655172413793105	1.9907793669418226e-05	4.287019643040732e-06	3.992017575200415	1.5490195998574319
6.275862068965518	1.1129255905431773e-05	2.3691124474405302e-06	3.923665651136531	1.5490195998574319
6.586206896551724	4.421395435917005e-06	9.597677949467058e-07	3.9587631947856607	1.5490195998574319
6.8965517241379315	2.1184110566044726e-06	4.6605421813211906e-07	3.9067348599915723	1.5490195998574319
7.206896551724138	8.352780021421518e-07	1.8793945831177554e-07	3.901419349638833	1.5490195998574319
7.517241379310345	3.096777419261219e-07	6.193396086795264e-08	3.937785052759164	1.5490195998574319
7.827586206896552	0.0	0.0	inf	1.5490195998574319
8.137931034482758	1.016232278179359e-07	3.0486968345380765e-08	3.5252158189599854	1.5490195998574319
8.448275862068966	9.28451658307507e-08	9.284516583075069e-09	3.5436128094978283	1.5490195998574319
8.758620689655173	0.0	0.0	inf	1.5490195998574319
9.068965517241379	0.0	0.0	inf	1.5490195998574319
9.379310344827587	0.0	0.0	inf	1.5490195998574319
9.689655172413794	0.0	0.0	inf	1.5490195998574319
10.0	0.0	0.0	inf	1.5490195998574319

Figure 9: Resultados

NOTA: En ambos casos se observan valores $0.0 \in P_{\text{ebf}} \wedge \infty \in G_c$. Esto se debe a que para $\frac{E_b}{N_0}$ grandes, donde la probabilidad de error es muy chica, no ocurrieron errores en la simulacion (ademas estos errores son aleatorios). Para evitar que esto suceda, se deben incluir aun mas palabras en la simulacion para evitar este tipo de valores y lograr una curva aun mas suave.

Ambos graficos y tablas se encuentran en el directorio `comunicacion_digital`*.*

1.3 Conclusiones

Las conclusiones obtenidas en base a la simulacion fueron:

- Ganancia Asintotica: Se comprueba que la **ganancia asintotica** resulta una aproximacion valida, unicamente para los casos donde $P_{\text{ebf}} \to 0$.
- Como vimos en la teoria, la aproximacion teorica de la probabilidad de error de bit (en BPSK) como

$$P_{\text{ebf}} \leq Q\left(\sqrt{2\frac{E_b}{N_0}}\right)$$

Es buena unicamente, en $\frac{E_b}{N_0}$ que no son cercanos. En nuestros graficos se puede observar que esta cota se empieza a cumplir desde $\frac{E_b}{N_0} \approx 4.2dB$

- En el caso del codigo usado como **detector** de errores, las probabilidades de error resultan muy bajas ya que estamos descartando aquellas palabras donde estos errores son detectados y no consideramos una posible retransmision, donde podrian ocurrir errores nuevamente.

2 Codificacion de Fuente

2.1 Introduccion

El objetivo de la **codificacion de fuente** es principalmente el de comprimir a la fuente lo maximo posible, restringido por un limite teorico que es $H(S)$ (la entropia de la fuente).

2.1.1 Codficacion de Huffman

Este algoritmo asigna las longitudes de palabra en forma inversa a las probabilidades de cada una (palabras mas probables, seran las mas cortas).

- Los codigos no son unicos (libertad en criterios de desempate o asignacion de simbolos en ramas).
- Se deben conocer probabilidades de antemano.

Para conocer las probabilidades de cada simbolo en nuestra imagen (ver Figure 10), debemos primero identificar a estos simbolos de acuerdo a un valor numerico. En nuestro caso hicimos una conversion a **escala de grises** para obtener dos simbolos posibles:

- $b : 0$
- $n : 255$

Figure 10: Imagen a comprimir

Luego se agruparon a los simbolos de la imagen en $n -$ tuplas para contar las ocurrencias de la **fuente extendida**. Por ultimo contamos cuantas veces aparece cada una de las 2^N tuplas posibles.

```
def frecuencias_imagen(image_path, n):
    # Convertimos a imagen a array en escala de grises
    img = np.array(Image.open(image_path).convert('L'))
    # Convertimos (2D) a (1D)
    flat = img.flatten()
    # Bloques de n simbolos consecutivos
    blocks = [tuple(flat[i:i+n]) for i in range(0, len(flat)-n+1, n)]
    # Frecuencias x bloque
    freq = Counter(blocks)
    return freq, flat.shape[0]
```

Figure 11: Frecuencias de fuente extendida

Una vez conseguidas las frecuencias, nos queda armar el arbol de **huffman**, en mi caso lo resolvi usando una **cola de prioridades**. Para luego, recorrer el mismo y asignarle $0 \vee 1$ dependiendo si es el hijo izquierdo o derecho (ver Figure 11).

```python
class Nodo:
    def __init__(self, simbolo, frecuencia):
        self.simbolo = simbolo
        self.frecuencia = frecuencia
        self.izq = None
        self.der = None
    def __lt__(self, otro):
        return self.frecuencia < otro.frecuencia

def construir_arbol_huffman(frecuencias):
    # Creamos un nodo x simbolo de fuente con su frecuencia
    heap = [Nodo(s, f) for s, f in frecuencias.items()]
    heapq.heapify(heap)

    # Creamos arbol de huffman
    while len(heap) > 1:
        nodo1 = heapq.heappop(heap)
        nodo2 = heapq.heappop(heap)
        nuevo_nodo = Nodo(None, nodo1.frecuencia + nodo2.frecuencia)
        nuevo_nodo.izq = nodo1
        nuevo_nodo.der = nodo2
        heapq.heappush(heap, nuevo_nodo)

    return heap[0]

def obtener_codigos(nodo, codigo_actual="", codigos={}):
    # Recorremos arbol recursivamente (0: izquierda, 1: derecha)
    if nodo is None:
        return
    if nodo.simbolo is not None:
        codigos[nodo.simbolo] = codigo_actual
    obtener_codigos(nodo.izq, codigo_actual + "0", codigos)
    obtener_codigos(nodo.der, codigo_actual + "1", codigos)
    return codigos
```

Figure 12: Arbol de Huffman

Por ultimo comprimimimos la imagen reemplazando cada simbolo de la fuente extendida, por su respectiva codificacion (ver Figure 13).

```python
def comprimir(codigos, image_path, n):
    # Convertimos a imagen a array en escala de grises
    img = np.array(Image.open(image_path).convert('L'))
    # Convertimos (2D) a (1D)
    flat_img = img.flatten()

    # Bloques de n simbolos consecutivos
    blocks = [tuple(flat_img[i:i+n]) for i in range(0, len(flat_img)-n+1, n)]

    # Reemplazamos cada bloque por su código correspondiente
    codigo_comprimido = []
    for bloque in blocks:
        codigo_comprimido.append(codigos[bloque])

    codigo_comprimido = np.array(codigo_comprimido).flatten()

    return codigo_comprimido
```

Figure 13: Compresion

2.2 Discusion

Los resultados obtenidos en base a la simulacion fueron los siguientes:

Fuente extendida ($n = 2$):

$$\bar{L_2} = 1.516$$

$$T_c = \frac{L_{\text{promedio original}}}{\frac{L_n}{n}} = 1.318$$

Fuente extendida ($n = 3$):

$$\bar{L_3} = 1.6$$

$$T_c = \frac{L_{\text{promedio original}}}{\frac{L_n}{n}} = 1.88$$

2.3 Conclusiones

El motivo detras de estos resultados esta en el **limite teorico**:

$$\lim_{n\to\infty} \frac{\bar{L_n}}{n} = H(s)$$

Es decir, al aumentar la extension de la fuente, nos acercamos a la compresion perfecta teorica dada por la **entropia de la fuente**. Es por esto que de $n = 2$ a $n = 3$, se ve una mejora la **tasa de compresion** (ver Figure 14).

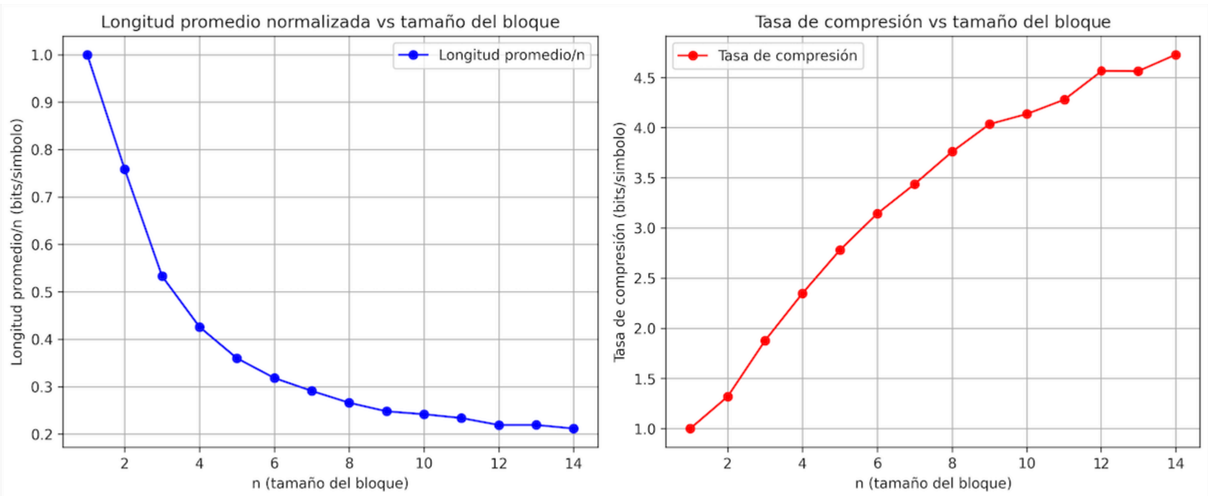

Figure 14: Longitudes y tasas (n)

La **equiprobabilidad** de los simbolos blanco y negro, se da para la fuente de extension $n = 1$. Al extender la fuente a n mayores, se presentan patrones de simbolos que no siguen las mismas probabilidades que la fuente base. Son estos patrones los que aprovecha Huffman para comprimir aun mas la imagen. Ver histograma de probabilidades para $n = 4$ (Figure 15), ningun simbolo se encuentra cerca del nivel equiprobable.

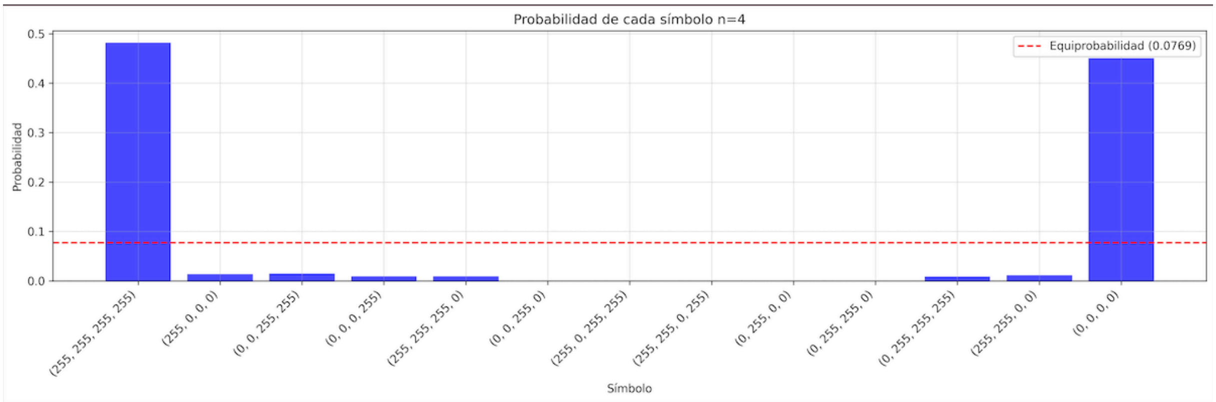

Figure 15: Equiprobabilidad (n=4)

3 Apendice

3.1 Estructura del Proyecto y Ejecucion

```
├── comunicacion_digital/
│   ├── resultados/          # Almacena resultados de simulaciones
│   ├── graficos/           # Almacena gráficos generados
│   ├── main.py             # Script principal de simulación
│   ├── helpers.py          # Funciones auxiliares
│   ├── csb.py              # Implementación del canal CSB
│   ├── codificacion.py     # Funciones de codificación
│   └── decodificacion.py   # Funciones de decodificación
│
└── compresion/
    ├── graficos/           # Almacena gráficos generados
    ├── main.py             # Script principal de compresión
    ├── helpers.py          # Funciones auxiliares
    ├── frecuencias.py      # Frecuencias de simbolos en imagen
    ├── huffman.py          # Implementación del algoritmo de Huffman (arbol y
codigos)
    └── logoFI.tif          # Imagen de prueba
```

Guia de uso:
El proyecto incluye dependencias basicas (numpy, pandas, PIL, scipy y matplotlib), para instalar las versiones con las que se realizo y testeo este proyecto, se recomienda crear un entorno de desarrollo virtual:

```
# En Windows
python -m venv venv

# En macOS/Linux
python -m venv venv
```

Activar el entorno virtual:

```
# En Windows
venv\Scripts\activate

# En macOS/Linux
source venv/bin/activate
```

Instalar las dependencias:

```
pip install -r requirements.txt
```

Comunicación Digital

(i) Ejecutar `main.py` en el directorio `comunicacion_digital/`
(ii) Los resultados se guardarán en el directorio `resultados/`
(iii) Los gráficos se generarán en el directorio `graficos/`

Compresión

(i) Ejecutar `main.py` en el directorio `compresion/`
(ii) Los resultados y gráficos se guardarán en el directorio `graficos/`