Trabajo Práctico de Simulación
Codificación de fuente y de canal

Universidad Nacional de La Plata

Facultad de Ingeniería

Valeria Micol García

N°Alumno: 03404/7

Índice

1. Introducción

El presente trabajo práctico tiene como objetivo principal la simulación de un sistema de comunicación digital sobre un canal ruidoso, implementando un código de bloque lineal sistemático con detección dura. Se busca analizar y comparar el desempeño del sistema codificado frente a un sistema sin codificación, utilizando modulación BPSK y un canal aditivo blanco gaussiano (AWGN).

Para ello, se desarrolló un código (14,10) con distancia mínima dmin = 3, capaz de corregir errores simples. La transmisión se simula mediante la generación de palabras de información aleatorias, codificación lineal, modulación BPSK, adición de ruido gaussiano, demodulación y decodificación. Posteriormente, se calculan las tasas de error de palabra y de bit para distintos valores de Eb/N0

Toda la simulación se implementó de forma matricial, optimizando el procesamiento y siguiendo la estructura propuesta por la cátedra. Se compararon los resultados con las curvas teóricas correspondientes (sistema sin codificación) y se analizó la ganancia de codificación obtenida, comparando tanto el sistema con corrección de errores como el sistema detector.

Además, como parte del segundo ejercicio, se implementó un algoritmo de compresión de fuente mediante la codificación de Huffman, aplicado a un archivo binario (imagen en formato TIFF). Se calcularon las probabilidades de los símbolos, se construyó el árbol de Huffman y se determinaron el largo promedio y la tasa de compresión del mensaje comprimido.

2. Construcción del código lineal (14,10)

En esta sección se diseñó e implementó un **código de bloque lineal sistemático** con las siguientes características:

- Longitud total de la palabra de código: $n = 14\ bits$
- Longitud del mensaje de información: $k = 10\ bits$
- Bits de paridad: $n - k = 4\ bits$

Para construir el código, se diseñó una matriz generadora G en forma sistemática:

$$G = [I_k \mid P]$$

donde:

- I_k es la matriz identidad 10×10, que representa la parte informativa.
- P es una matriz de paridad 10×4. La matriz P fue construida de forma tal que sus filas son linealmente independientes, distintas de cero y con peso de Hamming mayor o igual a 2 (es decir, $wH > dmin - 1 = 2$). Esto asegura distancia mínima = 3, ya que las combinaciones lineales de filas generan vectores con peso al menos 3.

Dicho de otra forma, las filas de G están construidas de tal manera que sus combinaciones lineales (suma módulo 2) nunca dan un vector de peso menor que 3, excepto el nulo. Esta propiedad también se traduce en que la suma (XOR) de las filas de G es el vector nulo, es decir:

$$\sum_{i=1}^{k} G_i = 0$$

Con esta configuración se encontró que $tc = 1$,

Por otra parte, se encontró $td = dmin - 1 = 2$

Este código fue seleccionado por su capacidad para detectar y corregir errores, cumpliendo con los requerimientos del ejercicio.

La matriz generadora completa se muestra a continuación:

$$G = [I_k \mid P]$$

```
n = 14
k = 10
dmin = 3

I = np.eye(k, dtype=int)
print ("Matriz identidad I[k*k]")
print(" ")
print (I)
```

✓ 0.0s

```
Matriz identidad I[k*k]

[[1 0 0 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 0 1]]
```

Figura 1: Creación de la matriz identidad

```
P = np.array([
    [1, 0, 1, 1],
    [1, 1, 0, 1],
    [0, 1, 1, 1],
    [1, 1, 1, 0],
    [1, 1, 0, 0],
    [0, 1, 1, 0],
    [1, 0, 1, 0],
    [1, 0, 0, 1],
    [0, 1, 0, 1],
    [0, 0, 1, 1]
], dtype=int)


print("Matriz de paridad P [ k * (n-k) ]")
print(" ")
print(P)
```

✓ 0.0s

```
Matriz de paridad P [ k * (n-k) ]

[[1 0 1 1]
 [1 1 0 1]
 [0 1 1 1]
 [1 1 1 0]
 [1 1 0 0]
 [0 1 1 0]
 [1 0 1 0]
 [1 0 0 1]
 [0 1 0 1]
 [0 0 1 1]]
```

Figura 2: Creación de la matriz de paridad

```
G = np.hstack((I, P))
print("Matriz G --> G[k*n] = [ I k*k | P k*(n-k) ]")
print(" ")
print(G)
```

✓ 0.0s

```
Matriz G --> G[k*n] = [ I k*k | P k*(n-k) ]

[[1 0 0 0 0 0 0 0 0 0 0 1 0 1 1]
 [0 1 0 0 0 0 0 0 0 0 1 1 0 1]
 [0 0 1 0 0 0 0 0 0 0 0 1 1 1]
 [0 0 0 1 0 0 0 0 0 0 1 1 1 0]
 [0 0 0 0 1 0 0 0 0 0 1 1 0 0]
 [0 0 0 0 0 1 0 0 0 0 0 1 1 0]
 [0 0 0 0 0 0 1 0 0 0 1 0 1 0]
 [0 0 0 0 0 0 0 1 0 0 1 0 0 1]
 [0 0 0 0 0 0 0 0 1 0 0 1 0 1]
 [0 0 0 0 0 0 0 0 0 1 0 0 1 1]]
```

Figura 3: Creación de la matriz Generadora

Posteriormente, se construyó la matriz de control de paridad H, utilizada para el cálculo del síndrome de las palabras recibidas:

$$H = [P^T \mid I_{n-k}]$$

```
I_2 = np.eye(n-k, dtype=int)
H = np.hstack((P.T, I_2))
print("\nMatriz HT:")
print ("HT = [ P  (k)  * (n-k)" )
print("       I (n-k) * (n-k) ]")
print(" ")
print(H.T)
```

✓ 0.0s

```
Matriz HT:
HT = [ P  (k)  * (n-k)
       I (n-k) * (n-k) ]

[[1 0 1 1]
 [1 1 0 1]
 [0 1 1 1]
 [1 1 1 0]
 [1 1 0 0]
 [0 1 1 0]
 [1 0 1 0]
 [1 0 0 1]
 [0 1 0 1]
 [0 0 1 1]
 [1 0 0 0]
 [0 1 0 0]
 [0 0 1 0]
 [0 0 0 1]]
```

Figura 4: Creación de la matriz de control de paridad

Esta matriz permite calcular el síndrome S mediante la expresión:

$$S = r \cdot H^T \mod 2$$

donde *r* es la palabra recibida. El síndrome *S* indica la presencia y localización de errores en el bloque recibido.

Finalmente, se calculó la ganancia asintótica de codificación para detección dura, dada por:

$$G_a = \frac{k}{n} \cdot \left\lfloor \frac{d_{min} + 1}{2} \right\rfloor$$

obteniéndose un valor aproximado de 1.4 para el código diseñado. Esta ganancia representa la mejora teórica máxima del sistema codificado respecto a un sistema sin codificación.

```
Ga = (k / n) * floor((dmin + 1) / 2)
print(f"La ganancia asintótica Ga (detección dura) es de ≈ {Ga:.3f}")

✓ 0.0s
La ganancia asintótica Ga (detección dura) es de ≈ 1.429
```

Figura 5: Cálculo de la ganancia asintótica

3. Simulación del sistema codificado (modo corrector)

En esta sección se realizó la simulación del sistema de comunicación digital con codificación de canal, considerando la capacidad del código de corregir errores. El esquema general consistió en transmitir bloques de información codificados, sometidos a un canal con ruido aditivo blanco gaussiano (AWGN), y aplicar un detector/corrector de errores. Se compararon los resultados con la curva teórica del sistema sin codificación.

La simulación del sistema codificado se implementó utilizando Python y se basó en una estructura matricial eficiente para representar la transmisión de M bloques de información.

El procedimiento consistió en los siguientes pasos:

- **Generación de bloques U**: Se crearon M palabras de información aleatorias de longitud k=10.
- **Codificación V**: Cada bloque fue codificado usando la matriz generadora G, obteniendo palabras de código de longitud n=14.
- **Modulación BPSK**: Las palabras codificadas se mapearon a símbolos BPSK ($0 \rightarrow -A$, $1 \rightarrow A$).
- **Canal con ruido AWGN**: Se sumó ruido gaussiano complejo a cada símbolo transmitido para simular el canal. Se utilizó la energía de bit Eb y densidad espectral $N0$correspondientes a cada $Eb/N0$.
- **Demodulación y detección**: Se realizó detección dura comparando el valor real recibido con un umbral 0, $(r > 0 \rightarrow 1 \text{ , } r < 0 \rightarrow 0)$
- **Cálculo del síndrome S**: Se utilizó la matriz de control H para determinar la presencia y localización de errores en cada bloque recibido.

- **Corrección VRcorregida**: Si el síndrome era distinto de 0, se corrigió el bit correspondiente al error identificado.
- **Decodificación Uest**: Se extrajeron los primeros k bits de cada bloque corregido para obtener las palabras decodificadas.
- **Cálculo de errores**: Se contaron los errores de palabra (V≠VRcorregida) y de bit (U≠Uest).
- **Resultados Pe y Pb**: Se calcularon las probabilidades de error de palabra y de bit, y se compararon con la curva teórica de BPSK sin codificación.

```python
for EbN0 in EbN0_dB:
    # Conversión de Eb/N0 de dB a veces
    EbN0_veces = 10**(EbN0 / 10)

    # Energías
    Es = A**2
    Ebf = Es * n / k
    N0 = Ebf / EbN0_veces

    # Pb teórico (sin codificación, BPSK)
    Pb = 0.5 * erfc(np.sqrt(EbN0_veces))
    Pb_teorico.append(Pb)

    # Estimación teórica del error de bit con codificación (cota superior)
    Pb_cod = 0
    for i in range(2, n + 1):
        Pb_cod += i * comb(n, i) * (Pb**i) * ((1 - Pb)**(n - i))
    Pb_cod_teorico.append(Pb_cod / k)

    # Estimación de errores por bloques acumulados
    Q_teorica = 1 - (1 - Pb)**n
    if Q_teorica == 0:
        Q_teorica = 1e-12

    errores_palabra_total = 0
    errores_bit_total = 0
    total_palabras = 0
    total_bits = 0

    repeticiones = 100
    M_bloque = max(num_bloques, int(10 / Q_teorica)) // repeticiones

    for _ in range(repeticiones):
        # Generación de palabras
        U = np.random.randint(0, 2, (M_bloque, k))
        V = np.mod(U @ G, 2)

        # Modulación BPSK
        S = (2 * V - 1) * A

        # Canal con ruido AWGN complejo
        ruido = np.sqrt(N0 / 2) * (np.random.randn(M_bloque, n) + 1j * np.random.randn(M_bloque, n))
        R = S + ruido

        # Demodulación dura
        VR = (np.real(R) > 0).astype(int)

        # Síndrome
        SIND = np.mod(VR @ H.T, 2)

        # Corrección de errores
        VR_corregida = VR.copy()
        for i in range(M_bloque):
            S_i = SIND[i]
            if np.any(S_i):
                for j, row in enumerate(H.T):
                    if np.array_equal(S_i, row):
                        VR_corregida[i, j] ^= 1
                        break

        # Decodificación
        U_est = VR_corregida[:, :k]

        # Conteo de errores
        errores_palabra = np.sum(~np.all(V == VR_corregida, axis=1))
        errores_bit = np.sum(U != U_est)

        # Acumulación
        errores_palabra_total += errores_palabra
        errores_bit_total += errores_bit
        total_palabras += M_bloque
        total_bits += M_bloque * k

    # Resultados finales
    Pe_palabra_sim.append(errores_palabra_total / total_palabras)
    Pb_bit_sim.append(errores_bit_total / total_bits)
```

Figura 6: Código de simulación

En el último bloque del código, se calcularon y almacenaron las tres principales probabilidades de error que caracterizan el desempeño del sistema de comunicación digital:

1. Probabilidad de error de palabra simulada (P_e)

$$P_e = \frac{\text{errores\_palabra}}{M}$$

Se calcula como la proporción de palabras de código transmitidas que contienen errores. Interpreta el rendimiento del sistema a nivel de palabra (es decir, cuántas palabras fueron incorrectas).

2. Probabilidad de error de bit simulada (Pb)

$$P_b = \frac{\text{errores\_bit}}{M \cdot k}$$

Es la fracción de bits erróneos respecto al total de bits transmitidos. Mide el desempeño del sistema a nivel de bit, considerando la codificación y corrección aplicada.

3. Probabilidad de error teórica sin codificación (Pb, teórico)

$$P_{eb} = Q\left(\sqrt{\frac{2E_b}{N_0}}\right)$$

Representa el error teórico del sistema sin codificación, usando la expresión del error de bit para modulación BPSK en un canal AWGN. En el código, esta fórmula está implementada con la función *erfc* de Scipy.

En este caso, la energía de bit Eb coincide con:

- $Eb = Ebc = Ebf$, ya que cada bit de fuente se transmite directamente como bit de canal

En cambio, cuando se utiliza codificación, la relación entre las energías cambia:

- $Ebc = k/n * Ebf$, por lo tanto la amplitud de transmisión debe ajustarse en la simulación para mantener constante Ebf/N0 y permitir una comparación justa entre el sistema codificado y el no codificado.

A continuación se muestra un gráfico donde se trazaron estas probabilidades en escala logarítmica en función de Eb/N0, permitiendo comparar el rendimiento del sistema codificado (curvas Pb y Pe) con la teoría sin codificación (curva Pb, teórica) y con la curva teórica de error de palabra correspondiente al sistema codificado.

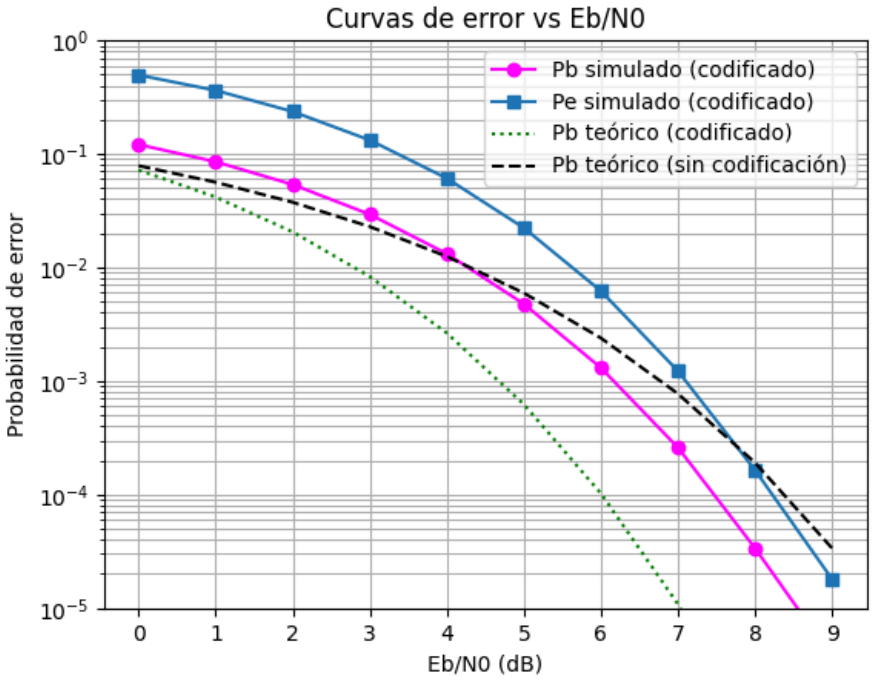

Figura 7: Curvas de tasa de error (Código corrector)

La ganancia de codificación se interpreta como la diferencia horizontal (en Eb/N0 en dB) necesaria para alcanzar un mismo nivel de probabilidad de error Pb en el sistema codificado respecto al sistema sin codificación.
Observamos que para un Pb específico (por ejemplo 10^{-3}), la curva violeta (sistema codificado) requiere un Eb/N0 menor que el sistema teórico sin codificación (curva negra), indicando una ganancia.
En este caso, la ganancia de codificación aproximada es de 1.4 dB, consistente con la ganancia asintótica calculada Ga.

3.1 Conclusiones

El código implementado permite reducir la energía necesaria por bit para alcanzar una determinada tasa de error, confirmando el efecto de la codificación.

La diferencia entre las curvas azul (Pe) y violeta (Pb) indica que aunque la probabilidad de palabra completa con error es relativamente alta, la probabilidad de error de bit se reduce significativamente gracias a la corrección.
La simulación evidencia que, para bajas probabilidades de error ($Pb \leq 10^{-3}$), el sistema codificado muestra una clara ventaja en términos de eficiencia energética al requerir un menor Eb/N0 para alcanzar la misma calidad de transmisión.

4. Simulación del sistema codificado (modo detector)

En esta sección se simula el desempeño del sistema digital usando el mismo código lineal propuesto anteriormente, pero empleándolo sólo como detector de errores, sin aplicar corrección. Esta aproximación permite analizar la capacidad del código para identificar errores, y cuantificar su impacto en la tasa de error en palabras y bits, comparado con el sistema sin codificación.

El procedimiento consistió en los siguientes pasos:

- **Generación de palabras aleatorias**: Se generan M bloques de longitud k, cada uno con bits equiprobables e independientes.
- **Codificación**: Se aplica la matriz generadora G para producir palabras codificadas de longitud n.
- **Modulación BPSK**: Los bits codificados se modulan mapeando 0 → -A y 1 → A, y se transmiten por el canal.
- **Ruido**: Se agrega ruido AWGN complejo a cada bloque simulado, modelando el canal realista.
- **Demodulación**: Se utiliza detección dura comparando el signo de la parte real de cada símbolo recibido.
- **Síndrome**: Se calcula el síndrome a partir de la matriz **H**, determinando si la palabra es válida o si contiene errores.
- **Contadores**:

  Si el síndrome es cero, se considera una palabra válida y se compara con la original para determinar errores de bit.

  Si el síndrome es distinto de cero, se cuenta la palabra como errónea, pero no se corrige.

```python
for EbN0 in EbN0_dB:
    EbN0_veces = 10**(EbN0 / 10)

    # Cálculo de energías
    Es = A**2
    Ebf = Es * n / k
    N0 = Ebf / EbN0_veces

    # Pb teórico sin codificación
    Pb = 0.5 * erfc(np.sqrt(2 * EbN0_veces) / np.sqrt(2))
    Pb_teorico.append(Pb)

    # Pe teórico con codificación (detector td = dmin - 1 = 2)
    Pe_cod_analitico = 0
    for i in range(3, n+1):  # desde td+1 = 3 hasta n
        Pe_cod_analitico += comb(n, i) * (Pb**i) * ((1 - Pb)**(n - i))
    Pe_palabra_teorica_cod.append(Pe_cod_analitico)

    # Estimación por bloques acumulados
    Q_teorica = 1 - (1 - Pb)**n
    if Q_teorica == 0:
        Q_teorica = 1e-12

    errores_palabra_total = 0
    errores_bit_total = 0
    total_palabras_validas = 0
    total_bits_validos = 0

    repeticiones = 100
    M_bloque = max(num_bloques, int(10 / Q_teorica)) // repeticiones

    for _ in range(repeticiones):
        # Generación y codificación
        U = np.random.randint(0, 2, (M_bloque, k))
        V = np.mod(U @ G, 2)

        # Modulación
        S = (2 * V - 1) * A

        # Canal AWGN complejo
        ruido = np.sqrt(N0 / 2) * (np.random.randn(M_bloque, n) + 1j * np.random.randn(M_bloque, n))
        R = S + ruido

        # Demodulación dura
        VR = (np.real(R) > 0).astype(int)

        # Cálculo de síndrome
        SIND = np.mod(VR @ H.T, 2)
        es_valida = ~np.any(SIND != 0, axis=1)

        # Solo decodificamos palabras válidas
        U_est = VR[:, :k]
        U_val = U[es_valida]
        U_est_val = U_est[es_valida]

        # Conteo de errores
        errores_en_validas = np.sum(np.any(U_val != U_est_val, axis=1))
        errores_bits_validos = np.sum(U_val != U_est_val)

        errores_palabra_total += errores_en_validas
        errores_bit_total += errores_bits_validos
        total_palabras_validas += np.sum(es_valida)
        total_bits_validos += np.sum(es_valida) * k

    # Guardamos resultados
    Pe_palabra_sim.append(errores_palabra_total / total_palabras_validas if total_palabras_validas > 0 else 1)
    Pb_bit_sim.append(errores_bit_total / total_bits_validos if total_bits_validos > 0 else 1)
```

Figura 10: Código de Simulación (Código detector)

En el último fragmento del código se calcula:

- Pe: tasa de error de palabra (simulada).
- Pb: tasa de error de bit (considerando solo palabras sin error detectado).
- Pb_teórico: tasa teórica de error de bit para BPSK sin codificación, usando:

$$P_{eb} = Q\left(\sqrt{\frac{2E_b}{N_0}}\right)$$

- Pe,teorico cod: cota teórica superior de la tasa de error de palabra para el sistema codificado (modo detector), calculada como:

$$P_e = \sum_{i=t_d+1}^{n} \binom{n}{i} P_b^i (1-P_b)^{n-i} \quad \text{con } t_d = d_{\min} - 1 = 2$$

A continuación se muestra el gráfico de las cuatro curvas resultantes: la probabilidad de error de bit simulada (Pb), la probabilidad de error de palabra simulada (Pe), la probabilidad de error de bit teórica sin codificación (Pb,teórico) y la probabilidad de error de palabra teórica del sistema codificado (Pe, teórico cod).

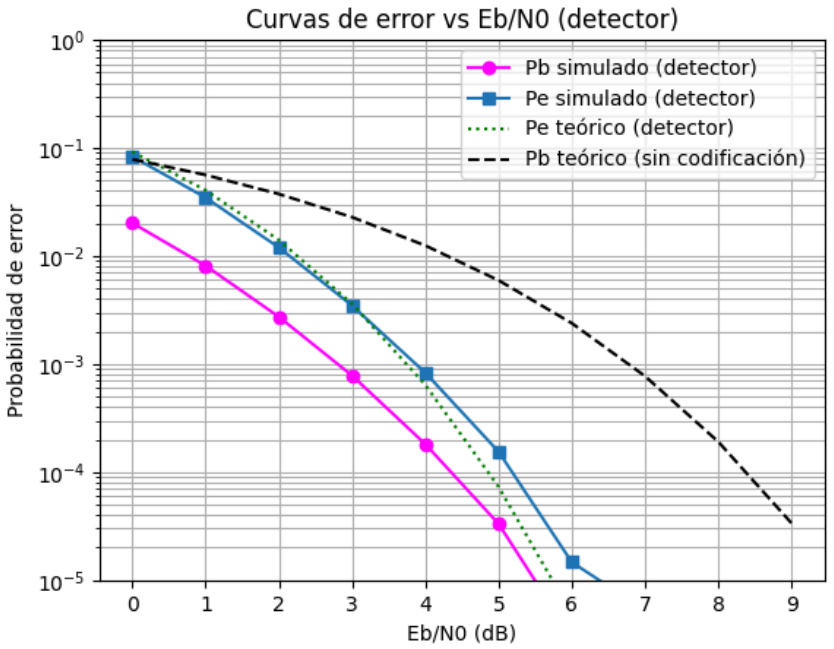

Figura 9: Curvas de tasa de error (Código detector)

En la Figura 9 se presentan las curvas del sistema detector, ya corregidas según las indicaciones recibidas. En versiones anteriores del gráfico, la tasa de error de palabra simulada (Pe) estaba sobreestimada porque se contaban como errores todas las palabras con síndrome no nulo. Sin embargo, esto no es correcto en el caso detector: el código está diseñado para descartar esas palabras, no para contabilizarlas como fallas.

En esta nueva versión, Pe se calcula únicamente sobre las palabras aceptadas como válidas, es decir, aquellas con síndrome nulo. Esto permite estimar correctamente el rendimiento real del sistema detector.

Se observa que las curvas simuladas se alinean bien con las curvas teóricas correspondientes. En particular, la curva simulada de Pe queda por debajo de la curva teórica del detector, lo cual es coherente con que esta última representa una cota superior.

Si bien la mejora en Pb es significativa, debe tenerse en cuenta que el detector descarta bloques con errores detectados, lo cual reduce el volumen de datos útiles transmitidos. Este es el costo asociado a obtener una menor tasa de error: pérdida de throughput o necesidad de retransmisión.

4.1 Conclusiones

El sistema detector implementado permite reducir significativamente la probabilidad de error de bit, ya que descarta aquellas palabras en las que se detectan errores mediante el cálculo del síndrome. De este modo, sólo se consideran válidas las palabras que pasan el control de paridad, lo que mejora la calidad de los datos aceptados.

La simulación muestra que la curva de error de palabra estimada se encuentra por debajo de la curva teórica correspondiente al sistema codificado, lo que indica que el desempeño obtenido es coherente con lo esperado. La curva teórica utilizada actúa como una cota superior basada en la capacidad de detección del código, con td=2.

Esta estrategia permite alcanzar bajas tasas de error, pero implica un compromiso: al descartar bloques con errores detectados, se reduce el volumen de datos útiles transmitidos. Por lo tanto, el uso de un código como detector mejora la confiabilidad, aunque puede requerir retransmisión o aceptación de pérdidas, dependiendo de la aplicación.

5. Compresión de Imagen Utilizando el Algoritmo de Huffman

En esta sección se implementó el algoritmo de Huffman para comprimir una imagen binaria (logo FI.tif), considerando la fuente extendida de orden 2 y orden 3. Se estimaron las probabilidades de cada bloque de símbolos y se construyó el código Huffman correspondiente. El objetivo era obtener el largo promedio y la tasa de compresión logrados, analizando además la razón por la cual se logra compresión a pesar de que los símbolos (blanco y negro) son prácticamente equiprobables.

El algoritmo se implementó en Python y se basa en la construcción de un árbol binario mínimo de probabilidades, conocido como heap o montículo, para garantizar una codificación eficiente. Los primeros pasos previos a la implementación de Huffman son:

- **Lectura y Procesamiento de la Imagen:** Se carga la imagen binaria y se convierte a un vector de píxeles (0 y 1), obteniendo la frecuencia de aparición de cada símbolo.
- **Construcción de Bloques (Fuente Extendida):** Se forman bloques de longitud 2 y 3 para calcular las probabilidades extendidas, es decir, se consideran secuencias de símbolos para reflejar dependencias y aprovechar mejor la redundancia.

```
img = Image.open('logo FI.tif')     # Se abre la imagen
pixels = np.array(img).flatten()    # Conversión de la imagen a vector
pixels = pixels.astype(int)         # Conversión de booleanos a enteros

# Conteo de frecuencias de cada símbolo (0 y 1) en la imagen
freq = Counter(pixels)      # conteo de apariciones de cada símbolo
total = len(pixels)         # total de píxeles en la imagen

# Probabilidad de cada símbolo
prob = {sym: count/total for sym, count in freq.items()}
print("Probabilidades:", prob)
```

```
Probabilidades: {1: 0.5146142686465267, 0: 0.4853857313534733}
```

Figura 10: Lectura, conteo de frecuencias y probabilidades de cada símbolo

Implementación del código de Huffman

Se construye una heap donde cada elemento contiene la probabilidad, el símbolo (bloque) y el código binario asignado. En cada iteración, se extraen los dos nodos con menor probabilidad, se asignan los prefijos '0' y '1', se combinan y se reintegran a la heap. Este proceso continúa hasta que se obtiene el árbol completo con el código final para cada símbolo.

```python
def calcular_huffman(pixels, n):

    # Recorrido del vector de pixeles de n en n
    bloques  = [tuple(pixels[i:i+n]) for i in range(0, len(pixels)-n+1, n)]

    # Frecuencia de cada bloque de longitud n
    freq = Counter(bloques)

    # Total de bloques formados
    total = len(bloques)

    # Probabilidad de cada bloque
    prob = {sym: count/total for sym, count in freq.items()}

    print ("Las probabilidades de cada bloque son: " , prob)
    print(" ")

    # Construcción del árbol de Huffman usando heap

    # Cada elemento de la lista heap tiene [probabilidad, [simbolo, codigo binario]]
    heap = [[prob, [sym, ""]] for sym, prob in prob.items()]

    # El elemento de menor probabilidad queda en la raiz
    heapq.heapify(heap)


    while len(heap) > 1:

        #Extracción de los dos nodos con menor probabilidad de la heap
        nodo_izq = heapq.heappop(heap)
        nodo_der = heapq.heappop(heap)

        #nodo_izq/der = [probabilidad, [símbolo1, código1], [símbolo2, código2], ...]


        # Asignación de prefijo '0' al código del subárbol izquierdo
        for simbolo_codigo in nodo_izq[1:]:
            #ejemplo: simbolo_codigo = [ (1,0), '10']
            simbolo_codigo[1] = '0' + simbolo_codigo[1]
            #ejemplo: simbolo_codigo = [ (1,0), '010']

        # Asignación de prefijo '1' al código del subárbol derecho
        for simbolo_codigo in nodo_der[1:]:
            simbolo_codigo[1] = '1' + simbolo_codigo[1]

        #Probabilidad total
        probabilidad_total = nodo_izq[0] + nodo_der[0]

        # Combinación de ambos nodos en un nuevo nodo padre
        heapq.heappush(heap, [probabilidad_total] + nodo_izq[1:] + nodo_der[1:])

    codigo_huffman = {bloque: codigo for bloque, codigo in heap[0][1:]}

    Largo_promedio = sum(prob[sym] * len(code) for sym, code in codigo_huffman.items())
    Largo_promedio = Largo_promedio / n
    bits_originales = total * n
    bits_comprimidos = sum(freq[sym] * len(codigo_huffman[sym]) for sym in freq)
    tasa_compresion = bits_originales / bits_comprimidos

    return Largo_promedio, tasa_compresion, codigo_huffman
```

Figura 11: Implementación del código de Huffman

Ejemplo gráfico paso a paso, explicando el código:

Suponiendo que se tienen los siguientes bloques (versión simplificada):

Bloque	Probabilidad
A	0.1
B	0.15
C	0.25
D	0.5

Paso 1: Se crea una heap con los bloques y sus probabilidades:

[0.1 , [A, ""]]

[0.15, [B, ""]]

[0.25, [C, ""]]

[0.5 , [D, ""]]

Paso 2: Se extraen los dos bloques con menor probabilidad: A (0.1) y B (0.15). Se asigna ‘0’ a A y ‘1’ a B. Se crea un nuevo nodo con probabilidad 0.24 (A+B)

- Nuevo nodo AB: [0.25, [A, "0"], [B, "1"]]
- Heap Actualizado:
  [0.25, [C , ""]]
  [0.25, [AB, [A, "0"], [B, "1"]]]
  [0.5, [D , ""]]

Paso 3: Se extraen los dos bloques con menor probabilidad: C (0.25) y AB (0.25). Se asigna '0' a C y '1' al subárbol AB. Se crea un nuevo nodo CAB con probabilidad 0.5.

- Nuevo nodo CAB: [0.5, [C, "0"], [A, "10"], [B, "11"]]

- Heap Actualizado:
  [0.5, [D, ""]]
  [0.5, [CAB, [C, "0"], [A, "10"], [B, "11"]]]

Paso 4: Se combinan D (0.5) y CAB (0.5). Se asigna '0' a D y '1' a CAB (agrega "1" a los códigos).

- Combino D y CAB:
  D: código "0"
  C: agrega "1" → "10"
  A: agrega "1" → "110"
  B: agrega "1" → "111"

- Árbol Final:
  D: "0"
  C: "10"
  A: "110"
  B: "111"

Resultado Final

Bloque	Probabilidad	Código
D	0.5	0
C	0.25	10
A	0.1	110
B	0.15	111

Resultados obtenidos con la implementación de Huffman

Por último, se calcula el largo promedio L ponderado por las probabilidades de los bloques en la fuente extendida de orden 2 y la de orden 3. También se calcula la tasa de compresión comparando los bits originales (sin compresión) con los bits comprimidos tras aplicar el código Huffman.

```
L2, tasa2, huff2 = calcular_huffman(pixels, 2)
print(f"Orden 2: L={L2:.2f}, tasa={tasa2:.2f}")

Las probabilidades de cada bloque son:  {(1, 1): 0.5042136030039256, (1, 0): 0.011627410820959208, (0, 0): 0.4749850657108722, (0, 1): 0.009173920464243045}

Orden 2: L=0.76, tasa=1.32
```

Figura 12: Resultados obtenidos aplicando huffman, fuente extendida orden 2

```
L3, tasa3, huff3 = calcular_huffman(pixels, 3)
print(f"Orden 3: L={L3:.2f}, tasa={tasa3:.2f}")

Las probabilidades de cada bloque son:  {(1, 1, 1): 0.493423579109063, (1, 1, 0): 0.010016641065028161, (0, 0, 0): 0.46542178699436765,

Orden 3: L=0.53, tasa=1.88

(0, 0, 1): 0.009072580645161291, (1, 0, 0): 0.009504608294930876, (0, 1, 1): 0.01238479262672811, (1, 0, 1): 1.600102406554019 5e-05, (0, 1, 0): 0.0001600102406554019 5}
```

Figura 13: Resultados obtenidos aplicando huffman, fuente extendida orden 2

Fuente Extendida de Orden 2:

- Largo promedio L=0.76 bits por símbolo
- Tasa de compresión ≈1.32

Fuente Extendida de Orden 3:

- Largo promedio L=0.53 bits por símbolo
- Tasa de compresión ≈1.88

5.1 Conclusiones

La compresión de imagen mediante el algoritmo de Huffman, implementado con la fuente extendida de orden 2 y 3, demostró ser una estrategia eficaz para reducir el tamaño de los datos codificados. Aunque la imagen binaria utilizada tiene una distribución casi equiprobable de píxeles blancos y negros, la extensión de la fuente permitió explotar las dependencias entre símbolos, logrando una codificación más eficiente.

Al analizar los resultados, se observa que el largo promedio se redujo de manera significativa al utilizar la fuente extendida de orden 3 respecto a la de orden 2, lo que implica que se necesitan menos bits promedio por símbolo para representar la información comprimida. Además, la tasa de compresión aumentó con la fuente extendida de orden 3, reflejando una mayor eficiencia en la reducción del tamaño de los datos transmitidos o almacenados.

Estos resultados evidencian que la compresión de datos no depende exclusivamente de la equiprobabilidad de los símbolos individuales, sino que también puede aprovechar las correlaciones entre ellos. La implementación de la estructura heap permitió construir de forma eficiente el árbol binario de Huffman, asegurando que los bloques con mayor probabilidad obtuvieran códigos más cortos, mientras que los de menor probabilidad obtuvieron códigos más largos, optimizando así la codificación global.

En resumen, el uso del algoritmo de Huffman con fuentes extendidas no sólo mejora la eficiencia de la compresión de datos, sino que también reduce la redundancia y el tamaño del archivo comprimido, contribuyendo a una transmisión y almacenamiento más eficiente.