Universidad Técnica de Ambato

PROYECTO DE FUNDAMENTOS DE PROGRAMACIÓN

Título: Simulador Visual de Señales Digitales
Área: Telecomunicaciones / Educación
Autores: Lara Sebastián, Larrea Rodrigo, Paredes David
Fecha: 10 de noviembre de 2025

ÍNDICE

---

1. INTRODUCCIÓN

El objetivo de este proyecto es desarrollar una aplicación educativa que permita a estudiantes de telecomunicaciones y carreras afines visualizar cómo se codifican secuencias binarias en diferentes esquemas de línea digital, sin necesidad de hardware externo.

La codificación de línea es un proceso fundamental en las telecomunicaciones digitales que convierte datos binarios en señales eléctricas u ópticas apropiadas para su transmisión. Este simulador facilita la comprensión de conceptos clave como:

- **Codificación de bits:** Representación física de 0s y 1s
- **Sincronización de reloj:** Recuperación temporal de la señal
- **Densidad de transición:** Cambios de nivel que facilitan sincronización
- **Ancho de banda:** Eficiencia espectral de cada esquema
- **Detección de errores:** Capacidad de identificar inconsistencias

Este proyecto se justifica por la necesidad de herramientas visuales interactivas que complementen la teoría tradicional, permitiendo a los estudiantes experimentar con diferentes secuencias binarias y observar inmediatamente el comportamiento de cada esquema de codificación.

---

2. MARCO TEÓRICO

2.1 Fundamentos de Codificación de Línea

La codificación de línea es la técnica mediante la cual los datos digitales se convierten en señales digitales. Los principales objetivos son:

1. **Sincronización:** La señal debe permitir que el receptor sincronice su reloj con el transmisor
2. **Inmunidad al ruido:** Minimizar errores de transmisión
3. **Eficiencia espectral:** Utilizar el ancho de banda de manera óptima
4. **Componente DC nula:** Evitar acumulación de carga en transformadores y capacitores
5. **Detección de errores:** Capacidad de identificar bits erróneos

2.2 Esquemas de Codificación Implementados

2.2.1 Non-Return to Zero Level (NRZ-L)

Principio de funcionamiento:

- Bit '1' se representa con nivel alto (+V)
- Bit '0' se representa con nivel bajo (-V)
- No hay retorno a cero entre bits del mismo valor

Fórmulas:

- Ancho de banda mínimo: BW = R (donde R = tasa de bits)
- Eficiencia: 1 bit por nivel de señal

Ventajas:

- Simplicidad de implementación
- Eficiente en ancho de banda
- Fácil de generar y detectar

Desventajas:

- Componente DC presente
- Sincronización problemática con largas secuencias de 0s o 1s
- No auto-sincronizable
- Sin capacidad inherente de detección de errores

Aplicaciones:

- Comunicaciones de corta distancia
- Sistemas donde la sincronización se maneja externamente

2.2.2 Manchester Encoding (Codificación Bifase)

Principio de funcionamiento:

- Cada bit se divide en dos mitades
- Bit '1': transición de bajo a alto en la mitad del periodo
- Bit '0': transición de alto a bajo en la mitad del periodo
- Siempre hay transición en el centro de cada bit

Fórmulas:

- Ancho de banda mínimo: BW = 2R
- Eficiencia: 0.5 bits por nivel de señal
- Frecuencia de la señal: f = R (para datos alternantes)

Ventajas:

- Auto-sincronizable: transición garantizada en cada bit
- No tiene componente DC
- Detección de errores por ausencia de transición
- Inmune a inversión de polaridad

Desventajas:

- Requiere el doble de ancho de banda que NRZ
- Mayor complejidad de implementación
- Consume más energía

Aplicaciones:

- Ethernet (10BASE-T)
- Token Ring
- RFID y sistemas de identificación

2.3 Tabla Comparativa

Característica	NRZ-L	Manchester
Ancho de banda	R	2R
Sincronización	Pobre	Excelente
Componente DC	Presente	Ausente
Complejidad	Baja	Media
Inmunidad al ruido	Media	Alta
Eficiencia espectral	Alta	Media

Característica	NRZ-L	Manchester
Detección de errores	No	Limitada

2.4 Importancia en Telecomunicaciones Modernas

Aunque las tecnologías actuales utilizan esquemas más avanzados (PAM, QAM), entender NRZ y Manchester es fundamental porque:

1. Son la base conceptual de esquemas más complejos
2. Se siguen usando en sistemas legacy
3. Ilustran el compromiso entre eficiencia y robustez
4. Son ejemplos claros de cómo resolver problemas de sincronización

---

3. ALCANCE DEL PROYECTO

3.1 Funcionalidades Principales

El simulador permitirá:

1. **Entrada de datos:**
    - Ingresar secuencia binaria de hasta 32 bits
    - Validación de entrada (solo 0s y 1s)
    - Selección de secuencias predefinidas de ejemplo
2. **Selección de esquema:**
    - NRZ-L (Non-Return to Zero Level)
    - Manchester (Bifase)
    - Interfaz clara con descripción de cada esquema
3. **Visualización gráfica:**
    - Gráfica de la señal resultante
    - Eje temporal con marcadores de bits
    - Niveles de voltaje claramente etiquetados
    - Colores distintivos para cada esquema
    - Cuadrícula para facilitar lectura
4. **Información educativa:**
    - Descripción del esquema seleccionado
    - Ventajas y desventajas
    - Aplicaciones prácticas
    - Fórmulas relevantes
5. **Exportación:**
    - Guardar gráfica como imagen PNG
    - Exportar datos en formato CSV

3.2 Limitaciones Definidas

- No se implementará hardware real
- Máximo 32 bits por secuencia (para claridad visual)
- Solo esquemas NRZ-L y Manchester en esta versión

- No se incluye simulación de ruido o errores de canal
- Interfaz básica sin animaciones avanzadas

3.3 Versión Mínima Viable (MVP)

Para la primera entrega funcional:

- Script Python ejecutable
- Entrada por consola o interfaz simple
- Gráfica con matplotlib
- Al menos NRZ-L funcionando correctamente

---

4. TECNOLOGÍAS Y MATERIALES

4.1 Lenguaje de Programación

Python 3.9+

Justificación:

- Sintaxis clara, ideal para propósitos educativos
- Amplia comunidad y documentación
- Excelente soporte para librerías científicas
- Multiplataforma
- Curva de aprendizaje suave

4.2 Librerías Principales

NumPy 1.24+

Función: Manejo eficiente de arrays para señales digitales

Justificación:

- Operaciones vectorizadas (100x más rápidas que listas)
- Funciones matemáticas optimizadas
- Estándar de facto en computación científica

Alternativa descartada: Listas nativas de Python (lentas para operaciones masivas)

Matplotlib 3.7+

Función: Visualización de formas de onda

Justificación:

- Gráficas de calidad publicable

- Control fino sobre elementos visuales
- Integración perfecta con NumPy
- Ampliamente documentada

Alternativa considerada: Plotly (interactivo pero más complejo, sobrecarga para este caso)

Tkinter (integrado en Python)

Función: Interfaz gráfica de usuario

Justificación:

- Incluido en Python estándar
- No requiere instalación adicional
- Suficiente para interfaces simples
- Cross-platform

Alternativa considerada: PyQt5 (más robusto pero mayor complejidad y dependencias externas)

4.3 Herramientas de Desarrollo

Editor de código: Visual Studio Code 1.85+

- IntelliSense para Python
- Extensión Pylint para análisis de código
- Integración con Git

Control de versiones: Git + GitHub

- Repositorio: [URL será proporcionada]
- Branch strategy: main (producción), dev (desarrollo)

Gestión de dependencias: pip + requirements.txt

```
numpy==1.24.3
matplotlib==3.7.1
```

Testing: unittest (módulo estándar de Python)

Documentación: Docstrings + README.md

4.4 Requisitos del Sistema

Hardware mínimo:

- Procesador: 1 GHz o superior
- RAM: 2 GB
- Espacio en disco: 500 MB

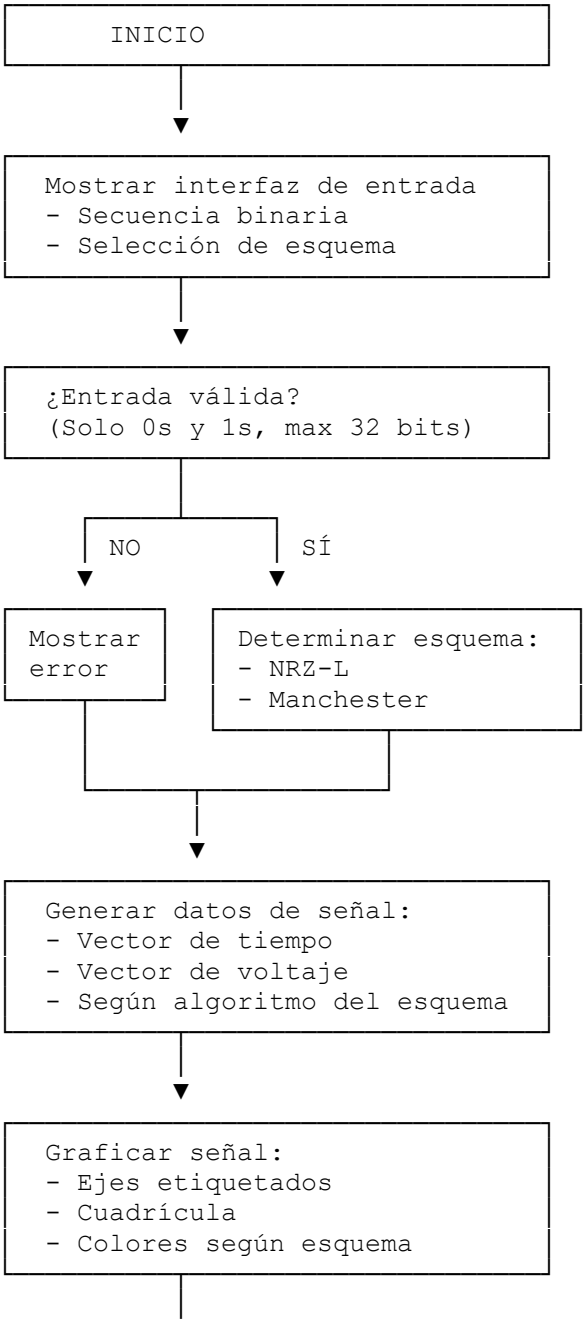
Software:

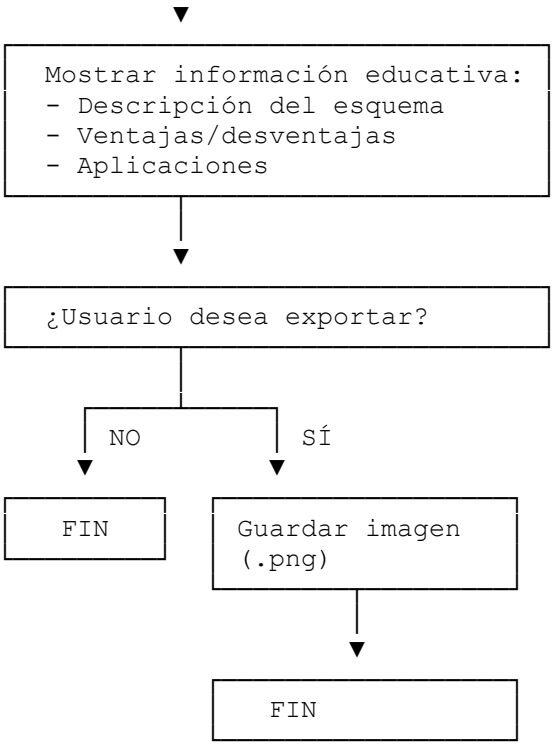
- Sistema operativo: Windows 10/11, macOS 10.15+, Linux (Ubuntu 20.04+)
- Python 3.9 o superior
- Conexión a internet (solo para instalación inicial)

---

5. METODOLOGÍA Y ARQUITECTURA DEL SISTEMA

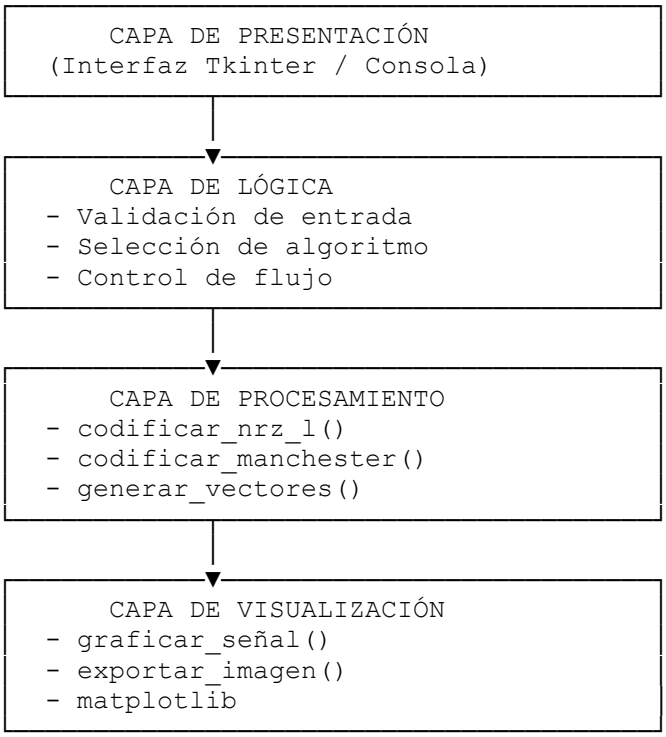
5.1 Diagrama de Flujo Principal





5.2 Arquitectura del Software

Modelo de capas:



5.3 Estructura de Archivos

```
simulador_señales/
```

```
│
├── src/
│   ├── main.py              # Punto de entrada
│   ├── codificadores.py     # Algoritmos de codificación
│   ├── graficador.py        # Funciones de visualización
│   ├── validador.py         # Validación de entrada
│   └── interfaz.py          # GUI con Tkinter
│
├── tests/
│   ├── test_nrz.py
│   ├── test_manchester.py
│   └── test_validador.py
│
├── docs/
│   ├── manual_usuario.pdf
│   ├── diagramas/
│   └── capturas/
│
├── ejemplos/
│   └── secuencias_prueba.txt
│
├── requirements.txt
├── README.md
└── .gitignore
```

5.4 Pseudocódigo de Funciones Principales

Función: codificar_nrz_l()

```
FUNCIÓN codificar_nrz_l(secuencia_bits):
    tiempo = []
    señal = []

    PARA cada bit EN secuencia_bits:
        índice = posición_del_bit

        // Cada bit dura 1 unidad de tiempo
        tiempo.agregar(índice)
        tiempo.agregar(índice + 1)

        // Determinar nivel de voltaje
        SI bit == '1':
            voltaje = +1
        SINO:
            voltaje = -1

        señal.agregar(voltaje)
        señal.agregar(voltaje)

    RETORNAR tiempo, señal
FIN FUNCIÓN
```

Función: codificar_manchester()

```
FUNCIÓN codificar_manchester(secuencia_bits):
    tiempo = []
    señal = []

    PARA cada bit EN secuencia_bits:
```

```
        índice = posición_del_bit
        mitad = índice + 0.5

        SI bit == '1':
            // Transición de bajo a alto
            tiempo.agregar(índice, mitad, mitad, índice+1)
            señal.agregar(-1, -1, +1, +1)
        SINO:
            // Transición de alto a bajo
            tiempo.agregar(índice, mitad, mitad, índice+1)
            señal.agregar(+1, +1, -1, -1)

    RETORNAR tiempo, señal
FIN FUNCIÓN
```

---

6. PLAN DE TRABAJO Y CRONOGRAMA

Fase	Actividad	Responsable	Meta	Fecha	Estado
Fase 1	Investigación y definición de requisitos	Lara	Documentación teórica completa, requisitos claros	Semana 1	✅ Completo
Fase 2	Diseño de arquitectura del software	Larrea	Diagramas UML, estructura de archivos, pseudocódigo	Semana 2	🔄 En progreso
Fase 3	Implementación NRZ-L	Larrea	Código funcional de NRZ-L con pruebas básicas	Semana 3	⬜ Planificado
Fase 4	Implementación Manchester	Larrea	Código funcional de Manchester, comparativa con NRZ	Semana 4	⬜ Planificado
Fase 5	Desarrollo de interfaz gráfica	Paredes	GUI funcional con Tkinter, integración completa	Semana 5	⬜ Planificado
Fase 6	Pruebas y validación	Todos	Suite completa de tests, corrección de bugs	Semana 6	⬜ Planificado
Fase 7	Documentación final	Lara	Manual de usuario, README, comentarios en código	Semana 7	⬜ Planificado
Fase 8	Presentación y entrega	Todos	Demo funcional, informe final, repositorio limpio	Semana 8	⬜ Planificado

---

7. DISTRIBUCIÓN DE RESPONSABILIDADES

7.1 Matriz de Responsabilidades

Integrante	Responsabilidad Principal	Tareas Específicas	Tiempo Estimado
Lara Sebastián	Investigación y Documentación	• Investigar esquemas de codificación • Redactar marco teórico • Escribir manual de usuario • Recopilar bibliografía • Documentar código	25 horas
Larrea Rodrigo	Desarrollo Backend	• Implementar algoritmos de codificación • Crear funciones con NumPy • Optimizar procesamiento • Escribir tests unitarios • Debugging	30 horas
Paredes David	Interfaz y Visualización	• Desarrollar GUI con Tkinter • Crear gráficas con Matplotlib • Diseñar experiencia de usuario • Implementar exportación • Testing de interfaz	28 horas

7.2 Tareas Compartidas

Tarea	Todos los Integrantes
Reuniones de seguimiento	Lunes y jueves 15:00-16:00
Code review	Pull requests revisados por al menos 1 compañero
Pruebas de integración	Validación conjunta de funcionalidades
Presentación final	Preparación y ensayo en equipo

7.3 Herramientas de Colaboración

- **GitHub:** Control de versiones y colaboración de código
- **Trello:** Gestión de tareas y progreso
- **WhatsApp:** Comunicación diaria
- **Google Meet:** Reuniones virtuales
- **Google Drive:** Documentos compartidos

7.4 Metodología de Trabajo

Enfoque: Desarrollo ágil adaptado

Sprints: Semanales (lunes a domingo)

Reuniones:

- **Lunes 15:00:** Planificación del sprint
- **Jueves 15:00:** Revisión de progreso y ajustes

Criterios de aceptación para cada tarea:

1. Código funcional y testeado
2. Documentación actualizada

3. Commits con mensajes descriptivos
4. Aprobación en code review

---

8. CASOS DE PRUEBA Y VALIDACIÓN

8.1 Casos de Prueba para NRZ-L

ID	Entrada	Resultado Esperado	Criterio de Éxito	Prioridad
NRZ-01	"1"	Señal constante en +1V	Nivel alto todo el tiempo	Alta
NRZ-02	"0"	Señal constante en -1V	Nivel bajo todo el tiempo	Alta
NRZ-03	"10"	Transición única de +1 a -1	Cambio en t=1	Alta
NRZ-04	"01"	Transición única de -1 a +1	Cambio en t=1	Alta
NRZ-05	"10101"	Alternancia constante	4 transiciones	Media
NRZ-06	"11111"	Señal constante en +1V	Sin transiciones	Media
NRZ-07	"00000"	Señal constante en -1V	Sin transiciones	Media
NRZ-08	"11001100"	Bloques de 2 bits	Transiciones en t=2,4,6	Alta
NRZ-09	(vacío)	Mensaje de error	"Secuencia no válida"	Alta
NRZ-10	"102"	Mensaje de error	"Solo 0s y 1s permitidos"	Alta
NRZ-11	33 bits	Mensaje de error	"Máximo 32 bits"	Media

8.2 Casos de Prueba para Manchester

ID	Entrada	Resultado Esperado	Criterio de Éxito	Prioridad
MAN-01	"1"	Transición bajo→alto en t=0.5	Forma de onda ∩	Alta
MAN-02	"0"	Transición alto→bajo en t=0.5	Forma de onda ∪	Alta
MAN-03	"10"	2 transiciones centrales	Cambios en t=0.5 y t=1.5	Alta
MAN-04	"01"	2 transiciones centrales	Cambios en t=0.5 y t=1.5	Alta
MAN-05	"10101"	5 transiciones centrales	Una por cada bit	Alta
MAN-06	"11111"	5 transiciones iguales	Todas bajo→alto	Media
MAN-07	"00000"	5 transiciones iguales	Todas alto→bajo	Media
MAN-08	"11001100"	Patrón repetitivo	Simetría visible	Media

8.3 Casos de Prueba de Integración

ID	Funcionalidad	Entrada	Acción	Resultado Esperado
INT-01	Cambio de esquema	"1010", NRZ→Manchester	Cambiar esquema sin reingresar bits	Gráfica se actualiza correctamente
INT-02	Exportación	"11001100", NRZ-L	Hacer clic en "Guardar"	Archivo PNG creado exitosamente
INT-03	Secuencias predefinidas	Seleccionar ejemplo	Cargar ejemplo	Bits y gráfica se cargan automáticamente
INT-04	Información educativa	Manchester	Hacer clic en "Info"	Ventana con descripción del esquema

8.4 Matriz de Trazabilidad

Requisito	Caso de Prueba	Estado
REQ-01: Validar entrada binaria	NRZ-09, NRZ-10, NRZ-11	□ Pendiente
REQ-02: Codificar NRZ-L	NRZ-01 a NRZ-08	□ Pendiente
REQ-03: Codificar Manchester	MAN-01 a MAN-08	□ Pendiente
REQ-04: Graficar señales	Todos	□ Pendiente
REQ-05: Exportar imagen	INT-02	□ Pendiente
REQ-06: Mostrar información	INT-04	□ Pendiente

8.5 Estrategia de Testing

Niveles de prueba:

1. **Pruebas unitarias** (unittest)
   - Cada función de codificación aislada
   - Validadores de entrada
   - Funciones auxiliares
2. **Pruebas de integración**
   - Flujo completo: entrada → codificación → gráfica
   - Interacción GUI ↔ backend
3. **Pruebas de aceptación**
   - Demostración a usuarios finales (compañeros)
   - Validación con profesor

Métricas de calidad:

- Cobertura de código: >80%
- Todos los casos críticos (alta prioridad) pasados
- 0 bugs críticos en producción

---

9. CÓDIGO IMPLEMENTADO

9.1 Módulo: codificadores.py

```
"""
Módulo de codificación de señales digitales
Autores: Lara, Larrea, Paredes
Fecha: Noviembre 2025
"""

import numpy as np

def codificar_nrz_l(bits):
    """
    Codifica una secuencia binaria en formato NRZ-L.

    Args:
        bits (str): Secuencia binaria (ej: "10110")

    Returns:
        tuple: (tiempo, señal) como arrays de NumPy

    Ejemplo:
        >>> t, s = codificar_nrz_l("101")
        >>> print(s)
        [1, 1, -1, -1, 1, 1]
    """
    n = len(bits)
    tiempo = np.zeros(2 * n)
    señal = np.zeros(2 * n)

    for i, bit in enumerate(bits):
        # Cada bit ocupa 2 puntos (inicio y fin del intervalo)
        tiempo[2*i] = i
        tiempo[2*i + 1] = i + 1

        # Nivel de voltaje según el bit
        nivel = 1 if bit == '1' else -1
        señal[2*i] = nivel
        señal[2*i + 1] = nivel

    return tiempo, señal


def codificar_manchester(bits):
    """
    Codifica una secuencia binaria en formato Manchester.

    En Manchester:
    - Bit '1': transición de -1 a +1 en la mitad del intervalo
    - Bit '0': transición de +1 a -1 en la mitad del intervalo

    Args:
        bits (str): Secuencia binaria

    Returns:
        tuple: (tiempo, señal) como arrays de NumPy

    Ejemplo:
        >>> t, s = codificar_manchester("10")
        >>> # Bit '1': -1,-1,+1,+1  |  Bit '0': +1,+1,-1,-1
    """
    n = len(bits)
```

```
    tiempo = np.zeros(4 * n)  # 4 puntos por bit (transición en medio)
    señal = np.zeros(4 * n)

    for i, bit in enumerate(bits):
        # Puntos de tiempo: inicio, mitad_antes, mitad_después, fin
        base = 4 * i
        tiempo[base:base+4] = [i, i+0.5, i+0.5, i+1]

        if bit == '1':
            # Bit '1': bajo en primera mitad, alto en segunda mitad
            señal[base:base+4] = [-1, -1, 1, 1]
        else:
            # Bit '0': alto en primera mitad, bajo en segunda mitad
            señal[base:base+4] = [1, 1, -1, -1]

    return tiempo, señal


def obtener_info_esquema(esquema):
    """
    Retorna información educativa sobre un esquema de codificación.

    Args:
        esquema (str): "NRZ-L" o "Manchester"

    Returns:
        dict: Información del esquema
    """
    info = {
        "NRZ-L": {
            "nombre": "Non-Return to Zero Level",
            "descripcion": "Codificación simple donde '1'=+V y '0'=-V
sin retorno a cero.",
            "ventajas": [
                "Eficiente en ancho de banda (BW = R)",
                "Fácil de implementar",
                "Bajo consumo energético"
            ],
            "desventajas": [
                "Pobre sincronización con secuencias largas de mismo
bit",
                "
```