Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA DE SEGOVIA

Grado en Ingeniería Informática de Servicios y Aplicaciones

Corrección de errores en imágenes mediante producto de códigos de Reed-Solomon

---

Alumno: Pablo Ibarlucea González

Tutor: Jose Ignacio Farrán Martín

Corrección de errores en imágenes mediante producto de códigos de Reed-Solomon

Pablo Ibarlucea González

Índice general

Índice de figuras

Índice de tablas

Resumen

Se exploran los códigos correctores de errores, en concreto el producto de códigos de Reed-Solomon aplicado a la codificación de imagenes. El enfoque principal es un algoritmo iterativo de decodificación para el cuál se ha desarrollado una aplicación web, de propósito divulgativo o educativo, que permite visualizar el proceso de corrección. Además se han investigado otros algoritmos como el algoritmo de Sudan y el algoritmo de Berlekamp-Masey.

Palabras claves: Códigos correctores, producto, Reed-Solomon, aplicación web, algoritmos, iterativo, imágenes.

Capítulo 1

Descripción del proyecto

En este capítulo se aborda el contexto del proyecto y los objetivos fundamentales que impulsan este trabajo.

1.1. Introducción

Los códigos correctores son esenciales en los procesos de transmisión de información que sustentan y definen nuestro mundo moderno. En numerosas ocasiones, resulta inevitable que se produzcan errores en la transmisión de datos, especialmente en entornos donde el medio de transmisión es intrínsecamente impredecible y difícilmente controlable, como ocurre en las transmisiones analógicas que atraviesan el aire, edificaciones u otros obstáculos. En estas circunstancias, la degradación de la señal es prácticamente inevitable, lo que puede afectar a la integridad y fiabilidad de los datos transmitidos.

Una fascinante aplicación de esta tecnología se encuentra en la exploración espacial, donde la comunicación entre naves espaciales y la Tierra enfrenta desafíos únicos y extremos. La Agencia Espacial Estadounidense (NASA) ha estado a la vanguardia en la utilización de códigos correctores, como los códigos de Reed-Solomon, para garantizar la integridad de los datos transmitidos desde sus misiones pioneras. El histórico programa Voyager en 1977 marcó un hito significativo en la aplicación de estos códigos, demostrando su eficacia en la transmisión de información vital a través de vastas distancias cósmicas. Desde entonces, estos códigos han sido una herramienta fundamental en diversas misiones espaciales, donde las condiciones extremas y las vastas distancias aumentan exponencialmente la probabilidad de que los datos sufran errores en el proceso de transmisión.[12]

La influencia de los códigos Reed-Solomon no se limita únicamente al espacio. En la esfera de los medios de almacenamiento óptico, como los CDs y DVDs, donde la integridad de los datos se encuentra en riesgo debido a arañazos y daños físicos, estos códigos han demostrado ser una defensa invaluable. La disposición inteligente de los códigos Reed-Solomon en forma de producto proporciona una capa adicional de protección contra errores concentrados en áreas contiguas de información, como las que resultan de un rayón en un disco. [7]

El presente trabajo se enfoca en explorar en profundidad los códigos de Reed-Solomon y su potencial aplicación en la corrección de errores en imágenes. A través de una interfaz gráfica intuitiva y didáctica, se presenta un proceso iterativo que destaca cómo estos códigos trabajan en conjunto para detectar y corregir errores en una imagen, proporcionando una visión detallada de su funcionamiento en un contexto práctico. Al adentrarse en esta investigación, se pretende comprender mejor la teoría subyacente de los códigos de Reed-Solomon y su relevancia en la resolución de desafíos reales en el mundo de la transmisión y el almacenamiento de datos.

Mediante este enfoque, se aspira a fomentar la comprensión de los códigos de corrección de errores, al mismo tiempo que se proporciona una herramienta visualmente enriquecedora para aquellos interesados en explorar este campo. Además, se refuerza la idea de que la corrección de errores es una disciplina esencial en el ámbito de la información, una ciencia en constante evolución que resguarda la integridad y confiabilidad de nuestros datos en movimiento perpetuo.

Finalmente, me gustaría destacar que este trabajo está coordinado con el trabajo final del grado de matemáticas [3] y que los resultados teóricos se encuentran allí detallados completamente.

1.2. Objetivos del trabajo

Los objetivos de este trabajo se han delineado en dos categorías fundamentales: objetivos a nivel teórico y objetivos a nivel práctico.

- **Objetivos a nivel teórico.** En esta vertiente, se persigue la exposición de conceptos esenciales que conducen al entendimiento del producto de códigos de Reed-Solomon. Esta comprensión se desarrollará de manera complementaria al enfoque teórico más profundo que se encuentra en el trabajo final del grado de matemáticas. [3]

  - **Fundamentos Matemáticos:** Se abordarán los fundamentos matemáticos necesarios, incluyendo la noción de cuerpos finitos y la teoría de códigos correctores. El enfoque se centrará en proporcionar una base sólida para comprender los códigos de Reed-Solomon.
  - **Algoritmos de Decodificación:** Se presentarán y explorarán diversos algoritmos de decodificación. Además de su mera presentación, se buscará profundizar en su comprensión a través de su implementación y se ilustrará su aplicación a través de ejemplos concretos.

- **Objetivos a nivel práctico.** En esta vertiente, se orientan los esfuerzos hacia la creación de una aplicación web con propósitos divulgativos, que permita visualizar de manera interactiva el proceso de corrección de errores mediante productos de códigos de Reed-Solomon.

- **Herramientas y Librerías:** Se expondrá el funcionamiento de las herramientas y librerías seleccionadas para llevar a cabo la implementación de los algoritmos. Esta comprensión proporcionará un marco contextual para la posterior aplicación práctica.
- **Implementación del Algoritmo Iterativo:** Se llevará a cabo la implementación del algoritmo de decodificación iterativa en productos de Reed-Solomon, con el objetivo de hacerlo aplicable a imágenes. Esto permitirá una corrección efectiva de errores en el contexto visual.
- **Interfaz Web Interactiva:** Crear una aplicación web que ofrezca una demostración interactiva del algoritmo iterativo. Los usuarios podrán aplicar el algoritmo a imágenes de su elección, y la interfaz gráfica permitirá introducir errores en la imagen antes de iniciar el proceso de decodificación.
- **Estudio Experimental:** Se realizará un estudio experimental para evaluar la capacidad correctora del algoritmo bajo diversas configuraciones y parámetros. Este análisis proporcionará información valiosa sobre el rendimiento y la eficacia del enfoque propuesto.

1.3. Antecedentes

Como se ha mencionado, el producto de códigos de Reed-Solomon ha encontrado su aplicación en medios de almacenamiento óptico como CDs y DVDs, proporcionando una sólida protección contra los errores inducidos por arañazos y daños físicos en las superficies de los discos. Sin embargo, en este trabajo se aborda un enfoque innovador que implica una iteración en el proceso de decodificación. Esta iteración se ha introducido con el propósito de mejorar la capacidad de recuperación de errores, permitiendo una corrección más efectiva y la restauración de la integridad de los datos en situaciones donde la cantidad de errores es considerable. En contraste con la tradicional aplicación de códigos de Reed-Solomon en discos, este proyecto presenta una nueva perspectiva al implementar la iteración en el proceso de corrección, abriendo el camino hacia una mayor robustez y confiabilidad en la recuperación de datos.

Como antecedente de la aplicación web, Tom Høholdt desarrolló una que demostraba la corrección de errores en DVDs utilizando códigos de Reed-Solomon. Esta aplicación, escrita en Java, ofrecía a los usuarios la oportunidad de experimentar el proceso de corrección de errores mediante una interfaz gráfica interactiva. Sin embargo, aunque esta herramienta brindó una comprensión práctica de los códigos de Reed-Solomon, lamentablemente ya no está disponible. [5]

En este contexto, el presente proyecto busca avanzar en esta dirección mediante el desarrollo de una nueva aplicación web, utilizando Python como lenguaje de programación, que permitirá a los usuarios explorar de manera interactiva y visual el proceso de corrección de errores basado en códigos de Reed-Solomon, específicamente en el contexto de imágenes. Con una interfaz moderna y accesible, esta herramienta aspira a llenar el vacío dejado por la aplicación previa de Tom Høholdt y proporcionar una experiencia

mejorada para aquellos interesados en comprender la importancia y el funcionamiento de los códigos de corrección de errores en el ámbito de la informática y la transmisión de datos.

1.4. Formato de la memoria

La memoria está principalmente estructurada en capítulos, 7 para ser concretos, seguidos de un apéndice que contiene los manuales de instalación y de usuario. Las referencias bibliográficas se pueden encontrar al final de la memoria.

Los tres primeros capítulos constituyen un grupo que aborda el contexto y organización del trabajo comenzando por este capítulo introductorio y siguiendo con los capítulos de metodología y planificación.

Posteriormente se incluye un capítulo que explica brevemente los fundamentos matemáticos que son el sustrato de todo el trabajo. Se darán las definiciones imprescindibles y algunos resultados clave. Además se justificará la utilidad práctica de los conceptos matemáticos desde este primer momento resaltando la manera en que se ajustan al problema particular de la correción de errores en imágenes digitales.

El siguiente capítulo entra en más detalle a explicar algoritmos de decodificación aplicables a los códigos de Reed-Solomon con los que tratamos, el enfoque que tomaremos es el de intercalar la explicación teórica con el código utilizado en la implementación. Este capítulo está pensado para ser complementario a los cuadernos Jupyter que se proporciona, ofreciendo el contexto matemático y detalles sobre el código.

Después, se sigue un enfoque similar para explicar el algoritmo iterativo. Sin embargo, le dedicamos un capítulo propio ya que es la pieza central del trabajo por estar implementado en la aplicación web. Con intención de facilitar la exposición, la lógica de la aplicación web también se incluye en un cuaderno jupyter que es complementado por este capítulo.

Para concluir, en el último capítulo analizaremos con retrospectiva el trabajo.

Capítulo 2

Metodología

En este capítulo se especifica el proceso seguido en el desarrollo del producto software principal de este trabajo, la aplicación web. Sin embargo dado el marco fuertemente teórico en el que está encajada, también se incluye en este proceso el estudio que ha sido necesario.

Por otra parte se presentarán las herramientas utilizadas a lo largo del proyecto y finalmente se explicará la decisión de arquitectura en lo que respecta a la aplicación web.

2.1. Proceso de desarrollo

El proceso de desarrollo utilizado ha sido el de desarrollo en cascada, por ser un proceso directo y estructurado debido a que los requisitos eran claros desde el comienzo y no había perspectiva de cambios.

El proceso de desarrollo de software en cascada es un enfoque tradicional y lineal para el desarrollo de software. Se caracteriza por su naturaleza secuencial y estructurada, donde cada fase del proyecto se completa antes de pasar a la siguiente. El proceso recibe su nombre de la idea de que el progreso fluye en una dirección, como una cascada, con cada fase descendiendo hacia la siguiente. Las etapas clave en el proceso de cascada original son las siguientes.

- **Recolección y Análisis de Requisitos:** En esta fase inicial, el equipo del proyecto trabaja en estrecha colaboración con los interesados para recolectar y documentar los requisitos detallados para el software. Estos requisitos describen lo que el software debe hacer, sus características, funcionalidades y expectativas de los usuarios.

  En nuestro caso esto se corresponde con las primeras reuniones con el tutor, donde se determina lo que se va a desarrollar, limitando claramente el alcance del trabajo.

- **Diseño del Sistema:** Durante esta fase, se planifica la arquitectura general y el diseño del sistema de software. Se definen las especificaciones técnicas, los componentes de software y sus relaciones. Este paso a menudo implica la creación de diagramas detallados, diagramas de flujo y modelos de sistema.

- **Implementación:** Se comienza a escribir código basado en las especificaciones de diseño. Todo el software se desarrolla en esta fase y se crean los componentes y módulos individuales. La codificación se realiza siguiendo los documentos de diseño y los lenguajes de programación elegidos para el proyecto.

- **Pruebas:** Una vez que el código está completo, se realizan pruebas exhaustivas para asegurarse de que el software cumpla con los requisitos especificados y funcione correctamente. Esta fase incluye varios métodos de prueba, como pruebas unitarias (pruebas de componentes individuales), pruebas de integración (pruebas de las interacciones entre componentes) y pruebas de sistema (pruebas de todo el software en su conjunto).

  En nuestro caso se realizan pruebas manuales, unitarias, de integración y de sistema debido a la pequeña escala del proyecto.

- **Despliegue y Mantenimiento:** Después de las pruebas exitosas, el software se integra e implementa en el entorno objetivo. Esto implica instalar el software en el hardware o plataforma prevista y configurarlo para el uso de los usuarios finales seguido de mantenimiento y soporte continuo.

  En principio esta fase no iba a ocurrir, concluyendo el proyecto con la entrega, sin embargo finalmente se ha realizado un despliegue en el servidor del grupo de investigación Singacom, disponible en el enlace `http:\www.singacom.uva.es`. Alguno de los detalles del proceso de despliegue se dan en el manual de instalación.

A estas fases habituales, añadimos una sexta, entre el diseño y la implementación, dedicada al estudio teórico preciso para comenzar a escribir el código y los algoritmos.

El modelo resultante se detalla gráficamente en la figura 2.1

2.2. Herramientas utilizadas

En esta sección, se describen detalladamente las diversas herramientas que se emplearon a lo largo del proyecto, abarcando desde los lenguajes de programación hasta las utilidades utilizadas para la confección de esta memoria.

- **Python:** La piedra angular de este proyecto fue el lenguaje de programación Python, en su versión 3.10. La elección de Python como el lenguaje principal se fundamentó en su versatilidad y agilidad, lo que permitió el desarrollo de scripts funcionales en un tiempo reducido. Además, su amplia popularidad garantizó la disponibilidad de una vasta gama de librerías especializadas, como el módulo empleado para aritmética en cuerpos finitos y códigos correctores. Mi familiaridad y profundo conocimiento de Python también desempeñaron un papel esencial en esta elección. Las siguientes son algunas de las herramientas específicas empleadas conjuntamente con Python:

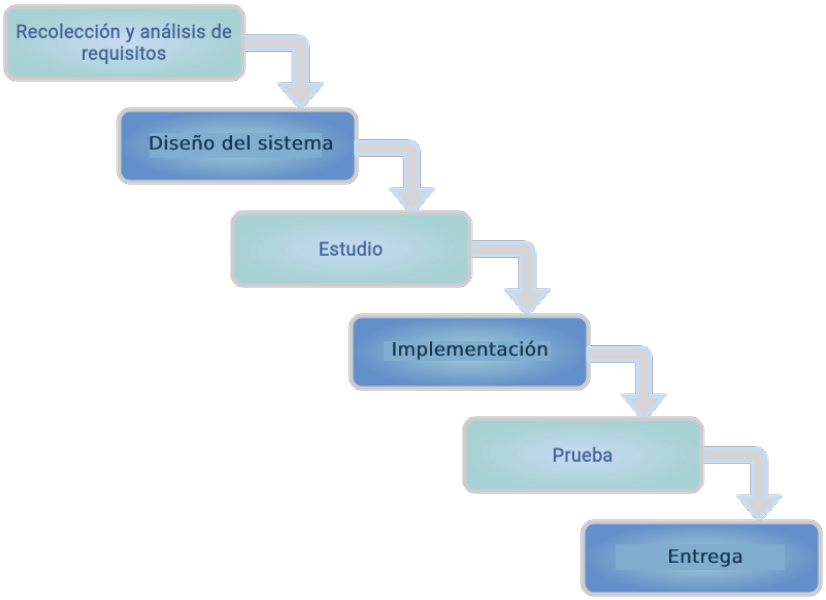


Figura 2.1: Proceso de desarrollo en cascada

- `numpy` : una librería fundamental en el ecosistema de programación en Python, reconocida por su capacidad para manipular matrices con gran eficiencia. Su popularidad se ha mantenido constante a lo largo de los años, y es ampliamente adoptada en la comunidad de desarrollo gracias a su versatilidad y alto rendimiento. En nuestro caso fue especialmente valioso para trabajar con las matrices asocidas a las imágenes.
- `galois` : una librería especializada que brinda soporte para operaciones en cuerpos finitos, que son fundamentales en la teoría de códigos correctores de errores. También implementa códigos correctores con algún algoritmo de decodificación. Otra característica destacada es su compatibilidad con la librería NumPy, permitiendo la integración fluida de operaciones en cuerpos finitos en entornos donde se requiere manipulación de matrices. [4]
- `pillow` : una librería para el tratamiento de imágenes en Python. Simplificó la manipulación de imágenes en el proyecto. Permitió la lectura y escritura de imágenes en diversos formatos y su conversión en estructuras de datos manejables, como matrices NumPy.
- `Flask` : es un microframework web, lo que significa que proporciona las herramientas esenciales y necesarias para construir aplicaciones web, pero sin imponer una estructura rígida o excesivas funcionalidades. Esta flexibilidad es el motivo principal por el cuál se escogió.

- **HTML5, CSS3:** forman los cimientos de la web moderna, desempeñando roles fundamentales en la creación y presentación de contenido en línea. HTML5, la quinta versión del lenguaje de marcado estándar, proporciona la estructura y los elementos

necesarios para organizar y describir el contenido de una página web, permitiendo la inclusión de texto, imágenes, enlaces y otros elementos multimedia. Por otro lado, CSS3, la última iteración de las Hojas de Estilo en Cascada, se dedica a la presentación y el diseño visual de los elementos HTML, permitiendo la personalización y el estilo de páginas web a través de formatos como colores, fuentes, márgenes y más. Estos dos lenguajes trabajan en conjunto para brindar una experiencia web atractiva y cohesiva. En este proyecto, HTML5 y CSS3 han sido utilizados para crear y dar forma a la interfaz de usuario, garantizando una experiencia de usuario intuitiva y agradable en la aplicación web desarrollada

- **JavaScript:** un lenguaje de programación de alto nivel que se ejecuta en el navegador web, desempeñó un papel esencial en la programación del lado del cliente. Se utilizó para agregar interactividad a la aplicación web, mejorando la experiencia del usuario. También, a través de la librería `fabric.js` se permite la modificación de una imagen que el usuario sube.

- **Git:** es un sistema de control de versiones distribuido ampliamente utilizado en el desarrollo de software. Permitió un seguimiento detallado de los cambios en el código fuente, permitiendo revertir a una versión anterior en caso de errores fatales. Se utilizó a través de la plataforma GitHub.

- **Docker** Docker es una plataforma que permite la creación, distribución y ejecución de aplicaciones en entornos aislados llamados contenedores. Se utilizó con el objetivo de garantizar la consistencia y portabilidad del entorno de desarrollo, evitando problemas de dependencias y configuraciones entre diferentes máquinas.

- **Jupyter:** es un entorno de desarrollo interactivo que permite la creación y compartición de documentos que contienen código, visualizaciones y texto explicativo. Se utilizó para implementar algoritmos que complementaron mi estudio y el trabajo final de grado en matemáticas. También permitió ejemplificar lo aprendido.

- **LaTeX:** es un sistema de composición tipográfica ampliamente utilizado para crear documentos técnicos y científicos de alta calidad. Se eligió para la creación de esta memoria debido a su capacidad para producir documentos bien estructurados y formateados. Se utilizó a través de la plataforma Overleaf.

- **draw.io** es una herramienta en línea para crear diagramas de manera colaborativa. Se utilizó para crear diagramas que ilustraran la arquitectura del proyecto y otros conceptos técnicos.

- **ray.so** es una herramienta que transforma fragmentos de código en imágenes atractivas y legibles, lo que facilita la presentación de conceptos técnicos de manera visualmente agradable.

2.3. Arquitectura

En esta sección se describirán las decisiones de arquitectura adoptadas para la implementación de la aplicación web. Comenzando por la arquitectura lógica, se ha optado por un enfoque de 2 capas con el propósito de simplificar la implementación y optimizar el rendimiento.

La elección de esta arquitectura se basa en varias consideraciones. En lugar de optar por un modelo más complejo, se prioriza la simplicidad para evitar introducir una complejidad innecesaria en el sistema. Dado que la aplicación tiene un enfoque educativo y no se espera un alto tráfico ni una gran cantidad de usuarios concurrentes, una arquitectura de 2 capas proporciona una escalabilidad adecuada para satisfacer estos requisitos.

Además, al reducir el número de capas en comparación con las aplicaciones empresariales convencionales, se minimiza la sobrecarga de comunicación y procesamiento entre las capas. Esto puede llevar a un mejor rendimiento, especialmente en casos donde la aplicación presenta un nivel de complejidad relativamente bajo.

A continuación, se detallarán de manera exhaustiva las funciones y responsabilidades de cada una de las capas:

- **Capa de Presentación (Frontend):**

  En esta capa reside la interfaz gráfica de usuario, que permite a los usuarios interactuar con la aplicación y visualizar el proceso de corrección de errores. Las principales funciones incluyen:

  - **Página de Inicio:** Proporciona una interfaz acogedora y amigable que da la bienvenida a los usuarios y brinda instrucciones sobre cómo utilizar la aplicación.
  - **Carga de Imagen:** Permite a los usuarios cargar imágenes en la aplicación, las cuales son enviadas al servidor para su procesamiento.
  - **Corrupción de Imagen:** Ofrece a los usuarios la posibilidad de corromper las imágenes mediante dibujos a mano alzada o introducción de ruido aleatorio, con la opción de solicitar al servidor que realice la corrupción.
  - **Visualización de Pasos:** Muestra visualmente los pasos intermedios del proceso de corrección de errores, ilustrando cómo se realiza la corrección en la imagen y presentando el resultado final.

- **Capa de Lógica y Procesamiento (Backend - Servidor en Flask):**

  Esta capa contiene la lógica central de la aplicación y se encarga del procesamiento de datos y la coordinación del proceso de corrección de errores. Sus principales responsabilidades son:

  - **Controlador de Rutas:** Gestiona las solicitudes HTTP provenientes de la interfaz de usuario y dirige las peticiones hacia las funciones de procesamiento correspondientes.

- **Gestión de Imágenes:** Al recibir una imagen cargada desde la interfaz gráfica, la almacena temporalmente para su procesamiento. A medida que se ejecuta el proceso de corrección de errores mediante el algoritmo de decodificación de Reed-Solomon, se generan pasos intermedios que demuestran cómo se detectan y corrigen los errores en la imagen.
- **Algoritmo de Ruido Aleatorio:** Implementa la corrupción de imágenes con ruido aleatorio, procesando las imágenes y devolviéndolas con la corrupción aplicada.
- **Algoritmo de Codificación y Decodificación:** Implementa el algoritmo de codificación de imágenes y el proceso iterativo de decodificación para corregir errores en las imágenes cargadas. Genera pasos intermedios que ofrecen una visión detallada de cómo se lleva a cabo la corrección.

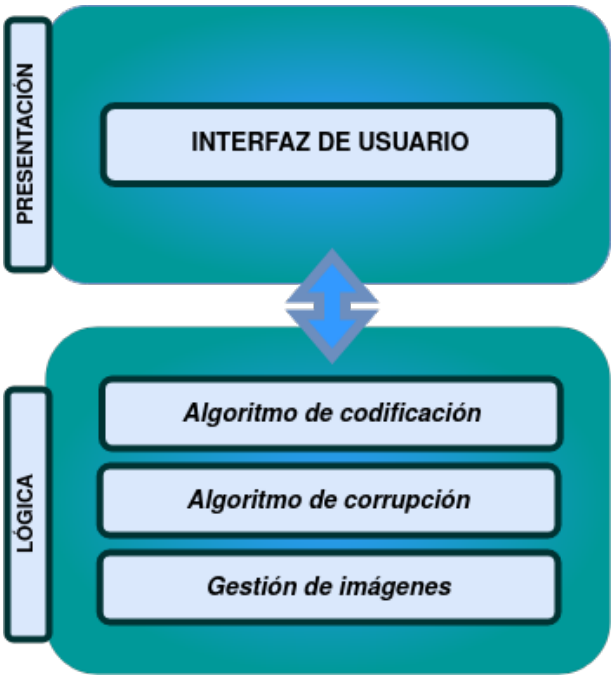


Figura 2.2: Resumen arquitectura lógica

Por otra parte la arquitectura física seleccionada se basa en un enfoque cliente-servidor directo y sencillo, con una correspondencia directa de estas capas físicas con las capas lógicas de presentación y lógica respectivamente. Los usuarios interactúan con la interfaz de usuario desde sus propios dispositivos, que actúan como clientes. En este escenario, la capa de presentación, que maneja la interacción visual y la experiencia del usuario, se ejecuta en los dispositivos de los usuarios. Cuando los usuarios realizan acciones en la interfaz, como cargar una imagen para su corrección, se generan solicitudes desde los dispositivos de los usuarios al servidor, que es la parte centralizada del sistema. Esta solicitud activa

el procesamiento de la imágenes, la corrección de errores, en el servidor. El servidor se encarga de realizar los cálculos necesarios y devuelve los resultados correspondientes a la interfaz del usuario en sus dispositivos.

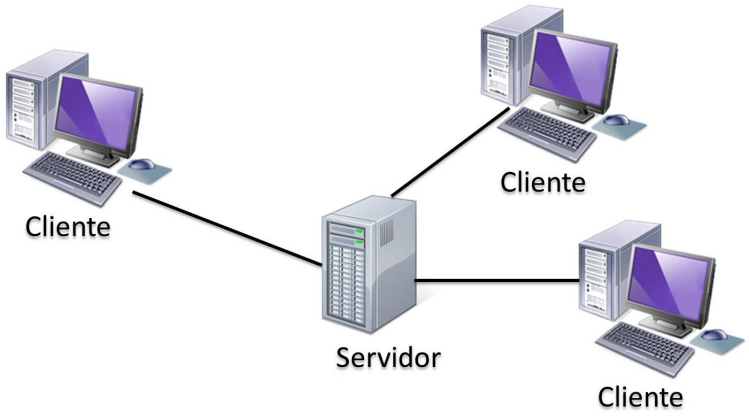


Figura 2.3: Arquitectura física

Capítulo 3

Planificación

En este capítulo, se abordarán las cuestiones fundamentales relacionadas con la planificación de actividades en el contexto de este proyecto. Se delineará cuidadosamente el enfoque temporal y la distribución de recursos. Además, se analizará la disposición estratégica de tiempos y tareas para asegurar un progreso coherente y una ejecución efectiva. A través de una visión detallada, este capítulo presenta lo que ha sido una hoja de ruta sólida que orientará el desarrollo y la finalización exitosa del proyecto.

3.1. Planificación temporal

Este trabajo de fin de grado se encuadra en una asignatura de 12 créditos (ECTS), estos créditos representan una estimación del esfuerzo necesario para completar una asginatura satisfactoriamente, en concreto 12 créditos corresponden a 300h de trabajo. La distribución de estas horas de trabajo va a ser el punto de partida de la planificaciñon temporal.

Se dispone de los meses de verano: junio, julio y agosto durante los cuales realizar el trabajo. Una estimación razonable del tiempo que podrá ser dedicado cada día es la propia de una media jornada laboral, 5h. Descontando los fines de semana y una semana en agosto para las vacaciones, esto nos deja con las 300h perfectamente distribuidas.

Ahora teniendo en cuenta el proceso de desarrollo declarado en el capítulo anterior y los objetivos declarados en el capítulo primero, vamos a dividir en tareas de duración más fácilmente estimables la realización del proyecto. Por supuesto, estas tareas deben reflejar también el proceso de redacción de este texto.

Merece la pena resaltar que no se va a tener en cuenta el proceso de estudio teórico en esta planificación ya que considero que esa parte se encuadra en el trabajo equivalente del grado de matemáticas. En cambio en esta fase del proceso de desarrollo en cascada que hemos detallado en la figura 2.1, se contará con la creación del cuadernillos de jupyter que exploran y ejemplifican los conceptos teóricos sin los cuales la implementación no sería posible.

- **Recolección y Análisis de Requisitos (1d):**
  - Reunión y análisis : 1 día
- **Diseño del Sistema (4d):**
  - Diseño de la arquitectura: 2 día
  - Selección de las herraminetas: 1 día
  - Diseño de la interfaz: 1 día
- **Estudio (19d):**
  - Cuadeno Jupyter Berlekamp-Masey : 3 días
  - Cuaderno Jupyter Sudan: 4 días
  - Cuaderno Jupyter Combinar : 3 días
  - Cuaderno Jupyter Iterativo: 5 días
  - Estudio del algoritmo Iterativo: 4 días
- **Implementación (16d):**
  - Configurar herramientas y entorno de desarrollo: 1 día
  - Configurar Flask: 1 día
  - Funcionalidad de cargar una imagen: 2 día
  - Funcionalidad de corromper: 2 días
  - Integrar algoritmo iterativo: 4 días
  - Funcionalidad mostrar pasos intermedios: 3 días
  - Dar estilo a la interfaz gráfica: 3 días
- **Pruebas (8d) :**
  - Pruebas de sistema : 3 días
  - Revisiones necesarias: 5 días

A las etapas del proceso de desarrollo se le suma la redacción de esta memoria

- **Redacción de la memoria (10d):**
  - Capítulo introductorio (1 día)
  - Capítulo descripción del proyecto (1 día)
  - Capítulo de planificación (1 día)
  - Capítulo de fundamentos matemáticos (1 día)

- Capítulos de algoritmos (3 días)
- Manuales (2 días)
- Conclusión (1 día)

El total de días sumando todas las tareas es de 58, con un par de días de margen por si fueran necesarios en algún momento.

Esta división de las tareas, por días, y gracias al proceso de desarrollo en cascada, ha permitido un flujo del proyecto muy estructurado y secuencial.

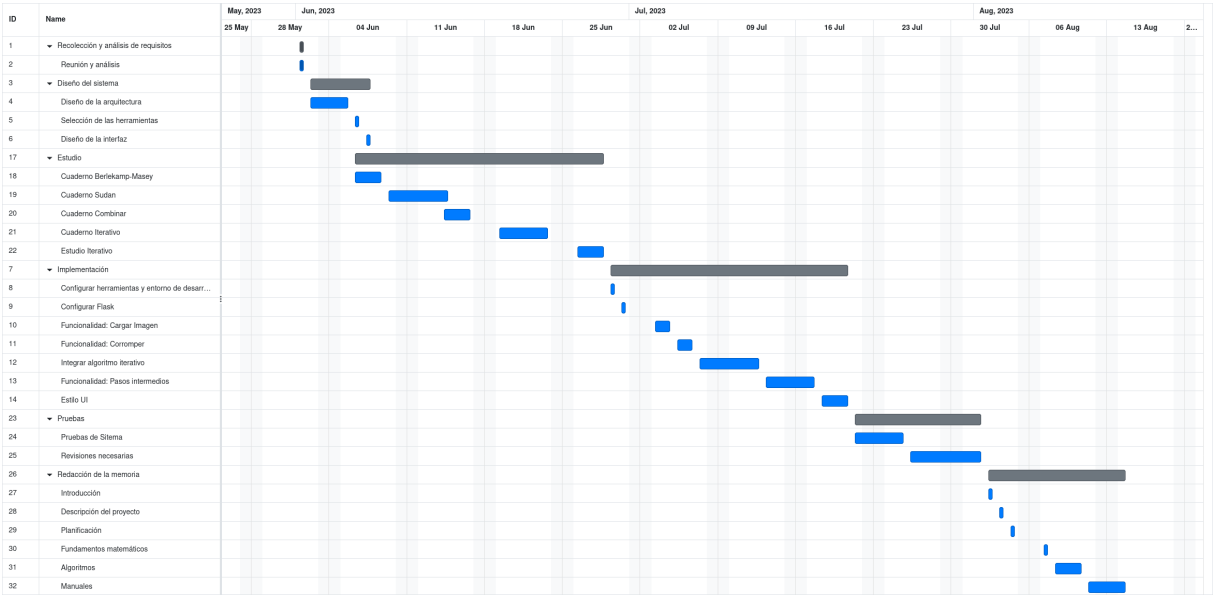


Figura 3.1: Diagrama de Gantt

Las desviaciones con respecto de este plan han sido mínimas, realizando cada tarea mayotitariamente dentro del rango de tiempo asignado.

3.2. Presupuesto Económico

3.2.1. Hardware y Software

En esta sección, evaluaremos los recursos de hardware y software utilizados en el proyecto. Se considerarán los costos asociados con la adquisición y uso de estos recursos.

- **Hardware:** El principal recurso de hardware utilizado en el proyecto es mi ordenador portátil, cuyo fue de 1000 euros. Su vida útil se estima es de aproximadamente 4 años. Será utilizado para el proyecto durante buena parte de 3 meses, o sea un 6,25 % de su vida útil, por lo tanto podemos estimar el coste del hardware para el proyecto en 62,5.

- **Software:** Se utilizaron herramientas y programas de código abierto, por lo que no se incurrirá en costos directos de software.

3.2.2. Recursos Humanos

La valoración de los recursos humanos es crucial para el presupuesto. Para ello se ha decidido en un rol o puesto laboral que se ajusta al trabajo que se ha realizado y se ha utilizado el saliario medio de este puesto en España para dar la valoración.

La posición laboral que mejor se ajusta a este trabajo es la de programador El salario de programador promedio en España es de 14,62€ por hora [9], la mayor parte del trabajo podría ser parte de las responsabilidades de un programador, incluyendo la redacción de documentación, sin embargo otra parte del trabajo (diseño, estudio, ...) quedaría fuera de lo esperado para este rol con lo que finalmente estimamos un sueldo de 15€ por hora. Dada la estimación de tiempo derivada de la valoración en créditos ECTS del trabajo, 300h, concluimos que el presupuesto final de recursos humanos para el proyecto es de 4500€.

3.2.3. Presupuesto Total

En esta tabla, presentaremos un resumen de los aspectos económicos del proyecto, incluyendo una comparativa entre las estimaciones iniciales y los costos reales:

	Hardware	**Software**	**RRHH**	**Total**
Estimación Inicial	62,5 €	0 €	4500 €	*4562,5 €*
Costo Real	62,5 €	0 €	4500 €	*4562,5 €*

Tabla 3.1: Resumen del Presupuesto Total

Se observa que los costos estimados se mantienen consistentes con los costos reales, lo que indica una gestión efectiva de los recursos y un presupuesto bien planificado.

En conclusión, el análisis económico del proyecto revela que los recursos de hardware y software no han generado costos adicionales, y los recursos humanos han sido eficientemente administrados dentro de las estimaciones presupuestarias (ya que se ha ajustado a la estimación de tiempo dada por los 12 créditos ECTS de 300h).

Capítulo 4

Fundamentos matemáticos

Como ya se ha mencionado en la introducción, este trabajo ha sido desarrollado en conjunto con el trabajo final del grado de matemáticas. Por completitud se incluye en esta memoria un breve resumen de algunos conceptos teóricos utilizados, dando razones también para su relevancia en la implementación. Para una descripción más detallada de los mismos nos referimos al trabajo de matemáticas.

4.1. Cuerpos finitos

Los cuerpos finitos son estructuras algebraicas que consisten en un conjunto finito de elementos junto con dos operaciones, la suma y la multiplicación, que satisfacen ciertos axiomas. Un cuerpo es un tipo particular de anillo conmutativo con identidad, en el que todos los elementos no nulos tienen un inverso multiplicativo. Una de las claves de la importancia de estas estructuras es que se pueden construir sobre ellas espacios vectoriales.

Los ejemplos más comunes son el cuerpo de los números racionales o el cuerpo de los números reales pero estos son infinitos. ¿Cuál es un cuerpo finito?

La aritmética modular con la que estamos familiarizados módulo un número primo resulta en un cuerpo finito, ya que todos los elementos tienen inverso.

Sea p un número primo, entonces $\mathbb{F}_p = \mathbb{Z}/(p)$ es un cuerpo finito de p elementos.

Además se puede construir cuerpos finitos de orden la potencia de un primo a partir del anillo de polinomios del cuerpo de orden primo correspondiente tomando el cociente por un polinomio irreducible.

Sea p un número primo, f un polinomio irreducible (no se puede factorizar) con coeficientes en el cuerpo $\mathbb{F}_p$ de grado n, entonces $\mathbb{F}_{p^n} = \mathbb{F}_p[x]/(f)$ es un cuerpo finito de p^n elementos.

De hecho estos son los únicos cuerpos finitos que existen.

Si K es un cuerpo finito, entonces su orden (número de elementos) es potencia de un primo. Además si dos cuerpos finitos tienen el mismo orden, son isomorfos.

El cuerpo finito de orden p^n se denota como $\mathbb{F}_{p^n}$ o $GF(p^n)$, donde GF significa *Galois Field* (Cuerpo de Galois) en honor a Évariste Galois, quien hizo importantes contribuciones al estudio de estas estructuras algebraicas.

Una propiedad que usaremos recurrentemetne es que siempre existe un elemento α, que llamamos primitivo, tal que

$$\left\{\alpha^i : i \in \{1, 2, \ldots, q-1\}\right\} = \mathbb{F}_q \setminus \{0\}$$

En cuanto a su aplicación, a parte de en códigos correctores, se utilizan por ejemplo en criptografía. El algoritmo de cifrado simétrico AES (Advanced Encryption Standard) y el algoritmo de firma digital DSA (Digital Signature Algorithm), utilizan operaciones algebraicas en cuerpos finitos.

Dos de las razones principales por las cuáles los cuerpos finitos son tan útiles en computación son

- **Representación simbólica** En informática es común utilizar cuerpos finitos binarios, su orden es una potencia de 2. Así podemos, por ejemplo, representar un byte como un elemento de $GF(2^8)$. En general, la posibilidad de representar los datos con símbolos de un alfabeto finito se adapta mejor a la realidad de los sistemas digitales que otras estructuras no discretas pero al mismo tiempo se mantiene la estructura de cuerpo permitiendo utilizar las potentes herramientas del álgebra lineal.

- **Operaciones eficientes** Los cuerpos finitos permiten realizar operaciones aritméticas de manera eficiente. La adición y multiplicación en un cuerpo finito se pueden implementar mediante algoritmos rápidos y optimizados, lo que resulta crucial en aplicaciones criptográficas donde se requiere un procesamiento eficiente de grandes cantidades de datos.

4.2. Códigos Correctores

Como referencia básica para el estudio de los códigos correctores se ha utilizado principalmente la segunda edición del libro .ª course in error-correcting codes"de Justesen y Høholdt [6], haciendo hincapié en los capítulos 4, Reed-Solomon Codes and Their Decoding"; 5, Çyclic Codes"; y 9, "Decoding Reed-Solomon and BCH Codes". Además, recurriremos a publicaciones originales y otras fuentes de referencia cuando busquemos profundizar en aspectos específicos.

El escenario que nos vamos a plantear recurrentemente es el de una **palabra** de n (longitud del código) símbolos que se recibe a través de un canal no fiable, que puede introducir errores. El objectivo del código corrector es permitir la corrección de estos errores. Los símbolos serán elementos de un cuerpo finito $\mathbb{F}_q$.

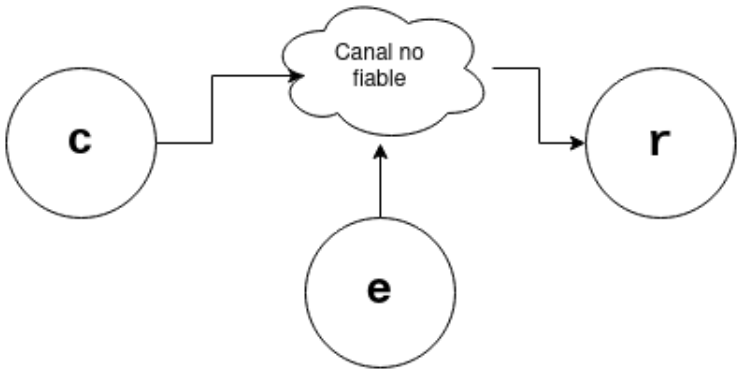


Figura 4.1: Contexto de los códigos correctores. Se envía una palabra c a la que se le introduce un error e para resultar en la palabra recibida r.

Los códigos que nos preocupan aquí son los códigos lineales, que no son más que subespacios vectoriales de $\mathbb{F}_q^n$. La dimensión del código es la dimensión como espacio vectorial y se suele denotar por k.

Los códigos correctores son capaces de corrigir errores ya que hacen corresponder la palabra recibida (que posiblemente no esté en el código) con la palabra del código más cercana. Por lo tanto un parámetro del código que es conveniente hacer grande es la distancia mínima d, la mínima cantidad de símbolos que separan dos palabras cualesquiera del código.

Una distancia mínima grande significa que las palabras del código están dispersas y es posible corregir más errores. El número de errores que un código puede corregir en cualquier caso decimos que es su capacidad correctora t.

Si un código tiene distancia mínima d entonces puede detectar $d-1$ errores y corregir $\lfloor \frac{d-1}{2} \rfloor$.

Otro concepto importante en los algoritmos de decodificación que veremos es el de síndrome. Se trata de un valor que depende únicamente del error introducido y no de la palabra enviada. Se calcula de la siguiente manera.

Si el código es un subespacio de dimensión k, entonces se puede describir mediante $n-k$ ecuaciones implícitas es decir, de manera matricial, existe una matriz H de dimensiones $(n-k, n)$ para un código $\mathcal{C}$ tal que

$$Hc^T = 0 \iff c \in \mathcal{C}$$

Entonces definimos el sindrome de una palabra recibida como

$$S = Hr^T$$

4.3. Códigos Reed-Solomon

Vamos a definir solamente la clase de códigos Reed-Solomon más utilizada. Dados un cuerpo finito $\mathbb{F}_q$ y un elemento primitivo α entonces el código Reed-Solomon de longitud $n = q - 1$ y dimensión k (denotado simplemente $RS[n, k]$) es

$$\{(f(1), f(\alpha), \ldots, f(\alpha^{n-1})) : f \in \mathbb{P}_k\}$$

donde $\mathbb{P}_k$ es el espacio de polinomio con coeficientes en el cuerpo finito de grado menor que k.

Otra manera de ver las palabras del código es como polinomios, a la palabra

$$(c_0, c_1, \ldots, c_{n-1})$$

Le hacemos corresponder el polinomio

$$c(x) = c_0 + c_1 x + \cdots + c_{n-1} x^n - 1$$

Se puede comprobar que de esta manera las palabras del código van a ser las que tengan como raíces $\alpha, \alpha^2, \ldots, \alpha^{n-k}$. Este otro enfoque permite calcular el síndrome de una palabra como

$$S = (r(\alpha), r(\alpha^2), \ldots, r(\alpha^{n-k}))$$

4.4. Códigos producto

La idea de los códigos producto es muy sencilla. En vez de ver la información de manera secuencial se dispone matricialmente y se codifican las filas con un código y las columnas con otro.

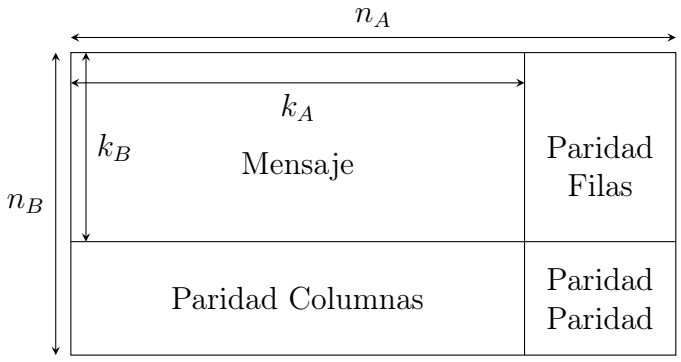


Figura 4.2: Estructura de un código producto.

En la figura 4.2 se ve la estrutura de una palabra de un código producto si se utiliza un codificador sistemático (que deja inalterado el mensaje y solo añade redundancia). Se puede obtener tal codificación a partir de codificaciones sistemáticas de los códigos que

componen el producto, primero codificaríamos por filas y posteriormente las columnas (incluyendo las columnas de paridad resultantes de codificar las filas). Tambíen es una posibilidad codificar primero las columnas y luego las filas ya que es equivalente.

Una de las principales ventajas de estos códigos es que están particularmente bien equipados para tratar con errores en ráfaga, es decir errores que se producen en una zona contigua de los datos, siempre que sean en dirección vertical u horizontal. Esto se debe a que independientemente de cuántos errores se hayan producido, si por ejemplo están aislados a una columna, cada fila tendrá a lo sumo un error lo que es fácilmente corregible.

Por este motivo se utiliza el producto de códigos Reed-Solomon en la codificación de DVDs que son propensos a rayones e imperfecciones, la información se almacena en el disco poniendo una fila tras otra de manera que un rayón tenderá a producir errores en una fila aislada.

Otro motivo por el cuál nos interesa este tipo de códigos en su aplicación a la codificación de imágenes es que ya tienen una estructura matricial que encaja a la perfección con la representación de una imagen.

En la práctica vamos a utilizar los códigos producto de la siguiente manera: tabajaremos, por supuesto, con el cuerpo $\mathbb{F}_{256}$ para representar bytes. Una imagen normal, a color, tiene 3 bytes por cada píxel correspondiendo cada byte a un canal de color (RGB), ya que digitalmente se representa un color dividiendolo en sus componentes roja, verde y azul asignando a cada una un valor entre 0 y 255.

Procederemos separando cada uno de estos canales, de manera que tenderemos 3 matrices del tamaño original de la imagen compuestas por bytes. Estas 3 matrices se codificarán y descodificarán por separado. Además utilizaremos el formato de imagen PNG, ya que es un formato de compresión sin pérdida (*lossless*). Esto es necesario ya que la información codificada también se va a almacenar en formato de imagen y necesitamos que los valores de los pixeles sean exactamente iguales para poder realizar la decodificación, no vale con una aproximación como la que nos daría el formato JPEG.

Capítulo 5

Algoritmos de decodificación

En este capítulo presentamos distintos algoritmos que se pueden utiilizar para la decodificación de códigos Reed-Solomon, con su implementación en `python` utilizando la librería `galois`.

5.1. Berlekamp-Massey

La referencia principal para el estudio del algoritmo de Berlekamp-Masey ha sido [2].

Se puede seguir esta sección con el cuaderno `jupyter` adjunto `bms.ipynb`.

El algoritmo de Berlekamp-Massey tiene una aplicación más general para encontrar la recurrencia lineal de longitud mínima que pueda generar una secuencia dada. El algoritmo da la recurrencia lineal a través de su polinomio asociado. Esto es, dada una sucesión

$$(V_1, V_2, \dots)$$

El algoritmo encuentra un polinomio de la forma

$$\Lambda(x) = 1 + \Lambda_1 x + \Lambda_2 x^2 + \dots \Lambda_L x^L$$

del menor grado posible tal que se cumple

$$\sum_{j=0}^{L} \Lambda_j V_{k-j} = 0 \quad \text{para } k > L$$

En el contexto de corrección si la sucesión que recibe el algoritmo es el síndrome de una palabra en la que se ha producido una cantidad corregible de errores, entonces el polinomio producido por el algoritmo tendrá como raíz α^{-i} si se ha producido un error en la posición i. Por ello decimos que este es el polinomio localizador de errores.

```python
def berlekamp_masey(S : list, GF: galois.FieldArray):
    t = len(S)//2
    # Condiciones iniciales
    lambd = galois.Poly.One(GF)
    x = galois.Poly([1, 0], field=GF) # Polinomio x de manera más legible en la
variable x
    beta = lambd
    l = 0
    for r in range(1,2*t):
        # Calcular discrepancia
        discr = GF(0)
        for j in range(l + 1):
            discr = discr + (lambd.coeffs[-j - 1] * S[r - 1 - j])
        if discr != GF(0):
            nlambd = lambd - discr * x * beta
            if 2 * l < r:
                beta = lambd* (discr ** -1)
                l = r - l
            else:
                beta = x*beta
            lambd = nlambd
        else:
            beta = x*beta
    return lambd
```

El proceso es inductivo; para cada $r = 1, \ldots, 2t$ se encontrará la recurrencia de menor longitud que genere la secuencia truncada $S_1, \ldots, S_r$. En el siguiente paso se comprueba si la recurrencia hallada en el paso anterior sigue sirviendo mediante la discrepancia Δ (la diferencia del termino real y el dado por la recurrencia). En caso de que no coincidan se actualiza de la siguiente manera

$$\Lambda^r(x) = \Lambda^{r-1} - \frac{\Delta_r}{\Delta_m} x^{(r-m)} \Lambda^{m-1}(x)$$

Donde m es el último número de iteración para el cuál se incremnetó la longitud de la recurrencia. Se puede comprobar que esta nueva produce los primeros r elementos y que lo hace con longitud mínima.

En el algoritmo, en cada paso se va actualizando un polinomio auxiliar que se puede utilizar para adaptar la recurrencia de esa manera. En concreto la variable `beta` en el algoritmo lleva el valor

$$\frac{1}{\Delta_m} x^{r-m-1} \Lambda^{m-1}(x)$$

Por ejemplo aplicar el algoritmo a la sucesión $(1, 1, 2, 3, 5, 8)$, en un cuerpo primo lo suficientemente grande resulta en el polinomio

$$1 - x - x^2$$

Lo que significa, tal y como se ve a simple vista, que cada término es la suma de los dos anteriores.

Ahora para la aplicación a la decodificación veamos un ejemplo en que se introducen dos errores en una palabra de un código Reed-Solomon$[7, 3]$. Codificamos primero 3 símbolos de información cualesquiera, comprobamos que el síndrome de la palabra codificada (calculado evaluando el polinomio asociado en $\alpha, \alpha^2, \alpha^3, \alpha^4$) es nulo

```python
#Usando un código RS[7,4] sobre GF(8)
rs = galois.ReedSolomon(7,3)
alpha = rs.field.primitive_element
c = rs.encode([1,1,2])

#El sindrome de la palabra codificada debería ser 0
print('Síndrome de la palabra codificada')
print([(galois.Poly(c))(alpha**i) for i in range(1,rs.n-rs.k+1)])
```

```
OUTPUT
Síndrome de la palabra codificada
[GF(0, order=2^3), GF(0, order=2^3), GF(0, order=2^3), GF(0, order=2^3)]
```

Ahora se introduce un error y se pasa al algoritmo el sindrome de la palabra con el error. Para el polinomio resultante se prueban todos los elementos α^{-i}, marcando para aquellos que sean raíces la posición i como error.

```python
#Síndromes
pol = galois.Poly(r)
S = [pol(alpha**i) for i in range(1,rs.n-rs.k+1)]

l = berlekamp_masey(S,rs.field)

# Los errores estan en la posición 0 y 1 (derecha a izquierda por defecto).
# El polinomio localizador de errores debe de tener como raíces alpha**0 y
alpha**-1
for i in range(rs.n):
    if l(alpha**-i) == rs.field(0):
        print(f'Error en la posición {i}')
```

```
OUTPUT
Error en la posición 0
Error en la posición 1
```

Finalmente se puede obtener la magnitud del error de manera sencilla a partir de la posición del error por ejemplo resolviendo el siguiente sistema

$$\begin{pmatrix} x_{i_1} & x_{i_2} & \dots & x_{i_t} \\ x_{i_1}^2 & x_{i_2}^2 & \dots & x_{i_t}^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_{i_1}^t & x_{i_2}^t & \dots & x_{i_t}^t \end{pmatrix} \begin{pmatrix} e_{i_1} \\ e_{i_2} \\ \vdots \\ e_{i_t} \end{pmatrix} = \begin{pmatrix} S_1 \\ S_2 \\ \vdots \\ S_t \end{pmatrix}$$

Donde $i_1, \dots, i_t$ son las posiciones de los errores.

5.2. Algoritmo de Sudan

El algoritmo que exploraremos en esta sección se clasifica como un algoritmo de decodificación por lista, en contraposición a la típica recuperación de una única palabra corregida. Este algoritmo se presenta en la referencia básica [6], además se ha consultado la referencia original[10].

Se puede seguir esta sección con el cuaderno `sudan.ipynb`.

Los algoritmos de este tipo no se limitan a ofrecer una única solución corregida, sino que proporcionan múltiples posibilidades. Este enfoque tiene el beneficio de mejorar la capacidad de corrección de errores en comparación con métodos convencionales. Es relevante destacar que en escenarios donde este algoritmo entra en juego, la presencia de múltiples palabras candidatas dentro de los rangos considerados es poco común por lo que en la mayoría de situaciones se comportará como un algoritmo de decodificación única con mejor capacidad de corrección.

Una de las principales desventajas de este algoritmo es que impone condiciones estrictas sobre los parámetros del código sobre el que se va a aplicar, dependiendo de los parámetros del algoritmo. En general se necesita que la tasa del código, la cantidad de símbolos de información en un mensaje, k/n, sea muy baja, entorno a $1/3$ para poder ofrecer una mejora en la capacidad correctora.

Se puede seguir esta sección con el cuaderno `jupyter` adjunto `sudan.ipynb`.

El algoritmo de Sudan tiene dos pasos. En el primero se busca, mediante la resolución de un sistema lineal, un polinomio en dos variables $Q(x, y))$ que interpola la palabra recibida de una manera concreta; y en el segundo se buscan polinomios $f(x)$ que cumplen $Q(x, f(x)) = 0$. Las palabras más cercanas a la recibida serán las generadas por estos polinomios.

Se consideran dos parámetros para el algoritmo: la capacidad correctora τ, cuántos errores se pueden llegar a tolerar; y la longitud máxima de lista l, cuántas palabras se pueden devolver como máximo.

Sea r la palabra recibida y $x_i = \alpha^{i-1}$ los puntos de evaluación para un código $RS[n, k]$, entonces buscamos un polinomio no nulo de la forma

$$Q(x, y) = Q_0(x) + Q_1(x)y + Q_2(x)y^2 + \cdots + Q_l(x)y^l$$

Cumpliendo

- $Q(x_i, r_i) = 0, i = 1, 2, \ldots, n.$
- $\deg Q_j(x) \leq n - \tau - 1 - j(k-1) = l_j, j = 0, 1, \ldots, l.$

Si se cumplen estas propiedades entonces se dará lo siguiente

Si $f(x)$ **cumple** $\deg f(x) \leq k$ **y** $d(f(x_1), \ldots, f(x_n), r) < \tau$**, entonces**

$$(y - f(x))|Q(x, y)$$

o equivalentemente

$$Q(x, f(x)) = 0$$

Es decir, podremos buscar los polinomios que generan palabras que están a dsitancia de la recibida menor que τ encontrando factores de este tipo.

Esto se debe a la elección de los grados de los polinomios Q_j, que hacen que $Q(x, f(x))$ sea un polinomio en x de grado a lo sumo $n-\tau-1$ pero por otra parte como $Q(x_i, r_i) = 0$ y $f(x_i) = r_i$ en al menos $n-\tau$ posiciones, entonces $Q(x, f(x))$ tiene al menos $n-\tau$ ceros. Esto solo es posible si $Q(x, f(x))$ es de hecho el polinomio nulo.

Para encontrar un polinomio Q de esta forma

$$Q_j(x) = \sum_{r=0}^{l_j} Q_{j,r} x^r$$

Entonces el sistema lineal para calcular este polinomio es el siguiente

$$\sum_{j=0}^{l} \begin{pmatrix} r_1^j & \dots & 0 & 0 \\ 0 & r_2^j & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & r_n^j \end{pmatrix} \begin{pmatrix} 1 & x_1 & \dots & x_1^{l_j} \\ 1 & x_2 & \dots & x_2^{l_j} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \dots & x_n^{l_j} \end{pmatrix} \begin{pmatrix} Q_{j,0} \\ Q_{j,1} \\ \vdots \\ Q_{j,l_j} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{5.1}$$

La implementación de este paso se ha realizado utilizando las herramientas de álgebra lineal de `numpy`.

Los coeficientes se añaden a la matriz por rangos de columnas correspondientes a cada uno de los polinomios Q_j, y para cada uno de estos por filas.

El mayor inconveniente ha sido que esta librería no permite obtener soluciones no nulas de sistemas con matrices no cuadradas o de rango no completo, por ello el algoritmo primero define la matriz según la ecuación anterior, luego la convierte en triangular mendiante la función `row_reduce` y finalmente completa la matriz con filas que la transforman en una matriz de rango completo.

De esta manera se puede utilizar ya la función de numpy `linalg.solve`. Para evitar que la solución sea nula se fuerza a que el coeficiente de mayor grado del último de los polinomios Q_j sea 1.

```python
def interpolar(r, tau, l, rs):
    GF = rs.field
    alpha = GF.primitive_element
    n = rs.n
    k = rs.k
    # numero de incognitas del sistema
    unknowns = (l + 1) * (n - tau) - (l * (l + 1) * (k - 1)) // 2
    # inicializar matriz
    A = GF.Zeros((unknowns, unknowns))
    # grados de los polinomios Q_j
    ls = []
    # determinan el rango de columnas que se definen
    # independientemente para cada Q_j
    last_col_index = 0
    new_col_index = 0
    # define las ecuaciones
    for j in range(l + 1):
        rows = []
        l_j = n - tau - 1 - j * (k - 1)  # Grado del polinomio Q_j
        ls.append(l_j)
        last_col_index = new_col_index
        new_col_index = new_col_index + l_j + 1
        for i in range(n):
            rows.append([r[i] ** j * (alpha ** (i)) ** k for k in range(l_j + 1)])
        A[:n, last_col_index:new_col_index] = GF(
            [item for row in rows for item in row]
        ).reshape(n, l_j + 1)
    # Se añaden ecuaciones para obtener una matriz de rango completo
    # para poder calcular una solucion no nula del sistema
    A = A.row_reduce()
    rank = 0
    ones_needed_in = [i for i in range(unknowns)]
    for i in range(unknowns):
        for j in range(unknowns):
            if A[i][j] == GF(1):
                ones_needed_in.pop(ones_needed_in.index(j))
                rank = rank + 1
                break
    for i in range(unknowns-rank):
            A[rank+i][ones_needed_in[i]]=GF(1)
    Q = np.linalg.solve(A,GF([0]*(unknowns-1) + [1]))
    # Lista larga de coeficientes a lista de polinomios
    Qs=[]
    c=0
    for l_j in ls:
        Qs.append(galois.Poly(Q[c:c+l_j+1], field=GF,order="asc"))
        c = c+l_j+1
    return Qs
```

Para el siguiente paso, la factorización, observamos que si $Q(x, f(x)) = 0$ entonces en particular $Q(0, f(0)) = 0$, es decir el termino independiente u_0 de f es raíz de $Q(0, y)$.

Para continuar buscando coeficientes primero ponemos $M(x, y) = Q(x, xy + u_0)$ con lo que este polinomio será divisible por x. Dividimos todo lo posible por x para obtener $M'(x, y)$ tal que $M'(0, y) \neq 0$ para todo y. El siguiente coeficiente será una raíz de $M'(0, y)$, y así sucesivamente.

Luego necesitamos 3 funciones auxiliares para el proceso: sustituir y por $xy+u$, dividir todo lo posible por x y encontrar raíces. Como la librería de galois no ofrece la posibilidad de trabajar directamente con polinomios en varias variables trabajamos con una lista de polinomios en una sola variable, empezando por los polinomios Q_j que da el algoritmo de interpolación y manteniendo esa estructura.

Comenzamos por la función de substitución `subs`. Todas las funciones toman como parámetro el objeto que representa el código Reed-Solomon para tener acceso al contexto necesario.

```
def subs(Qs,u,rs):
    GF = rs.field
    k =rs.k
    l = len(Qs)
    newQs=[galois.Poly.Zero(field=GF)]*l
    newQs[0]= Qs[0]

    for i in range(1,l):
        aux = [galois.Poly.Zero(field=GF)]*(i+1)
        for k in range(i+1):
            aux[k] = aux[k] + GF(math.comb(i,k) % GF.characteristic)*u**(i-
k)*galois.Poly.Identity(field=GF)**k
        for k in range(i+1):
            newQs[k] = newQs[k] + Qs[i]*aux[k]
    return newQs
```

En esta función `aux` representa el polinomio $(xy + u)^i$, pero de nuevo con una lista de polinomios en x que son *coeficientes* de y^k.

A continuación la función para dividir todo lo posible por x, `divX`.

```python
def divX(Qs,rs):
    newQs = Qs
    GF = rs.field
    while all( Q.coeffs[-1]==0 for Q in newQs):
        newQs = [Q//galois.Poly.Identity(field=GF) for Q in newQs]
    return newQs
```

Comprueba que ninguno de los polinomios tenga termino independiente y si es así divide por x.

Finalmente la ultima función auxiliar `roots`, que busca las raíces de $Q(0, y)$, simplemente probando todos los elementos del cuerpo finito.

```python
def roots(Qs,rs):
    GF = rs.field
    ret = []
    for i in range(GF.order):
        if sum([Q.coeffs[-1]*GF(i)**j for j,Q in enumerate(Qs)],start=GF(0)) == 0:
            ret.append(GF(i))
    return ret
```

Juntando estas funciones auxiliares es sencillo escribir el algoritmo, se ha hecho de forma recursiva ya que en cada paso las posibles raices pueden dar varias elecciones para un coeficiente creando una estructura de arbol.

```python
def r(Qs,sol,m,rs):
    if m==rs.k:
        #Caso base
        sols = [galois.Poly(sol,field=rs.field,order='asc')]
    else:
        sols = []
        for u in roots(Qs,rs):
            #Una llamada para cada posible raíz
            sols = sols + r(divX(subs(Qs,u,rs),rs),sol+[u],m+1,rs)
    return sols
```

Para demostrar esta estructura de árbol podemos probar el algoritmo con el polinomio (Ver figura 5.1)

$$y^3 + y^2(1 + x + x^2) + y(x + x^2 + x^3) + x^3 = (y - 1)(y - x)(y - x^2)$$

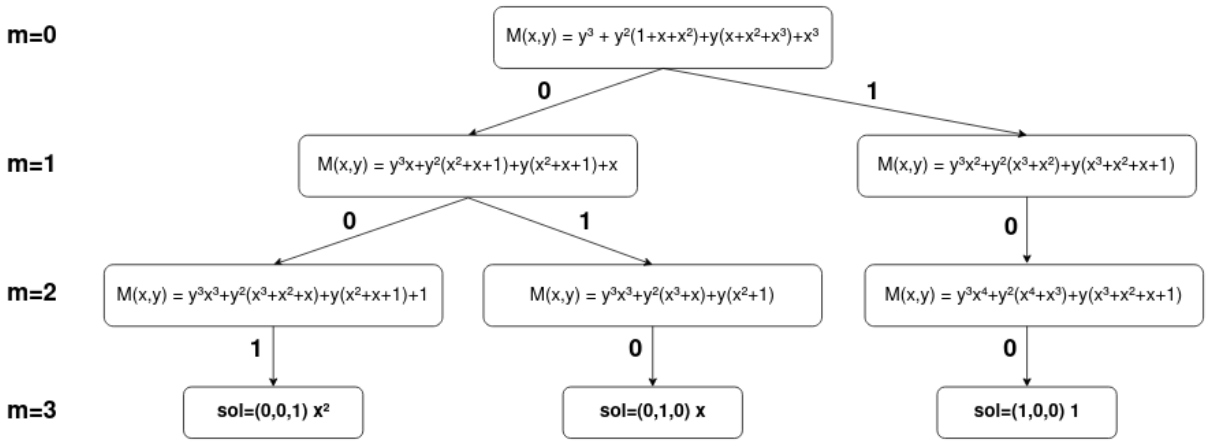


Figura 5.1: Traza del algoritmo para el polinomio $(y-1)(y-x)(y-x^2)$, donde se puede ver la estructura de árbol

Para dar por finalizada esta sección juntemos todo en un ejemplo, en primer lugar usamos el algoritmo de interpolar sobre una palabra recibida.

```python
GF = galois.GF(16, repr="power")
rs = galois.ReedSolomon(15, 3, field=GF)
alpha = GF.primitive_element

received = GF(
    [
        alpha**4,
        0,
        alpha**4,
        alpha**13,
        alpha**12,
        alpha**6,
        alpha**2,
        0,
        alpha**12,
        alpha**2,
        alpha**5,
        alpha**5,
        1,
        alpha**9,
        alpha**14,
    ]
)
Qs = interpolar(received, 7, 2, rs)
print(Qs)
```

```
OUTPUT
[Poly((a^2)x^7 + (a^14)x^6 + (a^9)x^5 + (a^5)x^4 + (a^3)x^2 + (a^12)x
+ a^2, GF(2^4)),
Poly((a)x^5 + (a^2)x^4 + (a^4)x^3 + (a^10)x^2 + (a^11)x + a^13, GF(2^4)),
Poly(x^3 + a^9, GF(2^4))]
```

Luego podemos encontrar los factores de este polinomio, recuperar las palabras del código dadas por estos y comprobar que estén en ese radio τ de la palabra recibida, ya que la garantía que tenemos es que de existir tales palabras, entonces serán dadas por uno de los factores pero no viceversa: puede haber factores que no den una de las palabras que buscamos y en tal caso habría que descartarla.

```
fs = r(divX(Qs, rs), [], 0, rs)
print(fs)
cs = [GF([f(alpha**i) for i in range(rs.n)]) for f in fs]
print(
    f"La distancia a r de cs[0] es {sum([1 if cs[0][i]!=received[i] else 0 for i in
range(rs.n)])}"
)
print(
    f"La distancia a r de cs[1] es {sum([1 if cs[1][i]!=received[i] else 0 for i in
range(rs.n)])}"
)
```

```
OUTPUT
[Poly((a^6)x^2 + (a^5)x + a^14, GF(2^4)),
Poly((a^11)x^2 + (a)x + a^9, GF(2^4))]
La distancia a r de cs[0] es 7
La distancia a r de cs[1] es 7
```

Ambas palabras están a la distancia requerida entonces ambas se incluyen en el resultado del algoritmo de Sudan.

5.3. Codigos producto con decodificador con lista

Este algoritmo se basa en uno introducido en [1].

El siguiente algoritmo es aplicable a cualquier producto de códigos, dado un decodificador con lista para uno de los códigos que lo compongan. Además es necesaria una manera de comprobar la pertenencia al otro código, lo cuál es normalmente muy sencillo (en códigos lineales tenemos siempre una matriz de paridad).

Se puede seguir esta sección con el cuaderno `comb.ipynb`.

La idea del algoritmo es sencilla. Supongamos que tenemos un decodificador con lista para el código de las filas que devuelve l posibilidades para cada fila. Si combinamos estas de manera ordenada, tal que las opciones que menos difieran de la matriz recibida sean las primeras en la lista, entonces podemos recorrer esta lista de combinaciones comprobando las columnas. La palabra más cercana a la recibida será la primera que pase este test para todas sus columnas.

Combinar los resultados de los decodificadores con lista para las filas no se puede hacer por fuerza bruta, presentamos primero un algoritmo para combinar listas ordenadas.

El algoritmo que vamos a dar combinará 2 listas ordenadas de números enteros y devuelve una lista ordenada de tuplas (i, j, n) donde i, j son los indices de los elementos respectivamente en la primera lista y en la segunda lista cuya suma resulta en n.

```python
import bisect

def fusion(u: list, v: list, l: int):
    g = []
    f = []
    for i in range(l):
        g.append([i, -1, u[i]])
    i = 0
    while i < l:
        z = g.pop(0)
        if z[1] == -1:
            z[1] = 0
            z[2] = u[z[0]] + v[z[1]]
            bisect.insort(g, z.copy(), key=lambda x: x[2])
        else:
            f.append(z.copy())
            i = i + 1
            z[1] = z[1] + 1
            if z[1] < l:
                z[2] = u[z[0]] + v[z[1]]
                bisect.insort(g, z, key=lambda x: x[2])
    return f

assert fusion([0, 2, 3, 5], [0, 1, 1, 3], 4) == [
    [0, 0, 0],
    [0, 1, 1],
    [0, 2, 1],
    [1, 0, 2],
]
```

En la implementación hemos utilizado la función `insort` de la librería `bisect` para insertar en una lista manteniendo el orden.

La lista de candidatos `g` se inicializa con la primera lista, e indices inválidos para la segunda lista. En cada paso del bucle se obtiene el primer elemento de la lista de candidates y se avanza el indice de la segunda lista, esta tupla pasará a ser parte del resultado final y se reintroduce también en `g` de manera ordenada.

El resultado final es la lista `f`, que contiene los elementos fusionados de manera ordenada. Este enfoque permite fusionar listas ordenadas de manera eficiente, manteniendo su orden.

En el código se incluye también un test de la función mediante un ejemplo.

El siguiente paso va a ser adaptar este algoritmo para que combine las filas en matrices. Tomará como argumento una lista ordenada de tuplas de la forma (m, d) donde m es una matriz y d es la distancia a la correspondiente submatriz de la palabra recibida.

Por ejemplo si m es una posibilidad dada por la combinación de los resultados de

decodificar con lista las dos primeras filas, será una matriz con dos filas y d será su distancia a las dos primeras filas de matriz recibida.

Tomará otro argumento que va a ser otra lista ordenada de tuplas (r, d) donde r será una única fila y d su correspondiente distancia.

El resultado será de nuevo otra lista ordenada de tuplas (m, d) donde se habrá añadido una de las filas de la segunda lista a una de las matrices de la primera.

```
def combines(m,l):
        target_shape = m[0][0].shape
        if len(target_shape) == 1:
            target_shape = (2,rs.n)
        else:
            target_shape = (target_shape[0]+1,rs.n)
        results = fusion([it[1] for it in m], [it[1] for it in l], limit)
        ret = []
        for result in results:
            rr = GF.Zeros(target_shape)
            rr[:-1,:] = m[result[0]][0]
            rr[-1,:] = l[result[1]][0]
            ret.append(
                (rr,result[2])
            )
        return ret
```

En la implementación simplemente usamos la función presentada antes con cuidado de mantener el formato.

Para obtener la lista final con posibles matrices basta con llamar repetidamente a esta función añadiendo en cada paso una de las filas.

Ahora podemos dar un decodificador para el producto de dos códigos Reed-Solomon. Como ya hemos mencionado esta misma técnica sirve perfectamente para otro tipo de códigos si se tiene un decodificador en lista.

```python
def get_combining_decoder(rs: galois.ReedSolomon, rsB: galois.ReedSolomon, limit : int):
    ld = get_list_decoder(rs,limit)
    GF = rs.field

    def combines(m,l):
      #...

    def decodesAm(M):
        n_rows = M.shape[0]
        partial = ld(M[0,:])
        i=1
        while i < n_rows:
            partial = combines(partial,ld(M[i,:]))
            i = i +1
        return partial

    def combining_decoder(M):
        for r in decodesAm(M):
            if all(rsB.detect(np.transpose(r[0]))):
                return r
        raise Exception('Undecodedable')

    return combining_decoder
```

En esta implementación usamos el concepto de `closure` en el que una función se guarda con el contexto que utiliza del entorno en el que fue definido, así podemos reutilizar el decodificador con lista `ld` sin tener que instanciarlo de nuevo en cada llamada a `combining_decoder`.

La función `decodesAm` es la encargada de obtener la lista final de la manera que ya se ha mencionado, llamando repetidamente a la función `combines` para añadir las filas una a una.

Además para la comprobación por columnas se ha utilizado el método `detect` que implementan los objetos `ReedSolomon` de la librería `galois`. Esta función devuelve un valor booleano, siendo el resultado falso cuando se detecta algún error en la palabra.

Se podría utilizar cualquier decodificador con lista pero para poder dar ejemplos más sencillos hemos implementado también un decodificador con lista perfecto, que es sólo utilizable para códigos muy pequeños ya que recorre el código entero para devolver una lista de *todas* las palabras ordenadas por distancia a una palabra dada.

De nuevo para esto se han utiilizado `closures`, ya que de otra manera la lista que contiene toda las palabras del código tendría que ser generada de nuevo con cada llamada del código.

```python
def get_perfect_list_decoder(rs : galois.ReedSolomon):
    GF = rs.field
    words = []
    for i in range(rs.n):
        for j in range(rs.n):
            for k in range(rs.n):
                for l in range(rs.n):
                    words.append(rs.encode(GF([i,j,k,l])))

    def perfect_list_decoder(r : galois.FieldArray):
        return(sorted([(w,d(w,r)) for w in words],key=lambda t: t[1]))

    return perfect_list_decoder

def get_list_decoder(rs: galois.ReedSolomon, limit:int):
    pld = get_perfect_list_decoder(rs)

    def list_decoder(r: galois.FieldArray):
        return pld(r)[:limit]

    return list_decoder
```

Donde se ha utilzado la siguiente función para calcular la distancia

```python
def  d(r: galois.FieldArray,c: galois.FieldArray):
    assert(len(r)==len(c))
    return sum([0 if r[i] == c[i] else 1 for i in range(len(r))])
```

Ahora, utilizando un límite suficientemente grande podemos encontrar una de las palabras más cercanas a cualquier palabra recibida.

En el siguiente ejemplo se va encontrar la palabra más cercana en un código producto de $RS[7,4]$ y $RS[7,6]$ a una palabra aleatoria.

```
rs = galois.ReedSolomon(7,4,field=galois.GF(8,repr='power'))
rsB = galois.ReedSolomon(7,6,field=galois.GF(8,repr='power'))
M = rs.field.Random((rsB.n,rs.n))
R = get_combining_decoder(rs,rsB,30)(M)
```

Obtenemos que la palabra más cercana a la matriz

$$\begin{pmatrix} \alpha^6 & \alpha^3 & \alpha^2 & \alpha^4 & \alpha^6 & \alpha^6 & \alpha^2 \\ \alpha & \alpha^3 & 1 & \alpha & \alpha^5 & 1 & \alpha^6 \\ \alpha^6 & \alpha^6 & \alpha^2 & \alpha^5 & \alpha^6 & \alpha^3 & 1 \\ \alpha^3 & \alpha^2 & \alpha^4 & \alpha^6 & 1 & \alpha^6 & \alpha^6 \\ \alpha^2 & \alpha & \alpha & 0 & \alpha^4 & \alpha & 0 \\ \alpha & \alpha & \alpha & \alpha^5 & 1 & \alpha^3 & \alpha^4 \\ \alpha^4 & \alpha^3 & \alpha^3 & 1 & \alpha & \alpha^3 & 1 \end{pmatrix}$$

es

$$\begin{pmatrix} \alpha^6 & \alpha^3 & \alpha^2 & \alpha^4 & \alpha^6 & \alpha^6 & \alpha \\ \alpha^2 & \alpha^3 & 0 & \alpha & \alpha^5 & 1 & \alpha^6 \\ \alpha^6 & \alpha^6 & \alpha^2 & \alpha^2 & \alpha^6 & \alpha^3 & \alpha^5 \\ \alpha^3 & \alpha^2 & \alpha^4 & \alpha^6 & \alpha^6 & \alpha & \alpha^6 \\ \alpha^2 & \alpha^3 & \alpha & \alpha & \alpha^4 & \alpha & 0 \\ 1 & \alpha & \alpha & \alpha^3 & 1 & \alpha^3 & 0 \\ \alpha^6 & \alpha^3 & \alpha^4 & 1 & \alpha & \alpha^3 & 1 \end{pmatrix}$$

que dista 14 de la primera. (Donde la representación es en función de un elemento primitivo α).

Capítulo 6

Aplicación Web : Algoritmo Iterativo

En este capítulo detallaremos en profundidad el proceso para aplicar el algoritmo iterativo a imágenes. Se puede seguir este capítulo con el cuaderno incluido `iter.ipynb`.

El algoritmo iterativo como tal, no tiene ninguna complejidad teórica. Se basa en decodificar las filas y columnas varias veces, hasta que no se puedan corregir más errores. Esto se basa en la observación que la correción de un error en una columna puede eliminar los errores necesarios para que una fila que anteriormente no era corregible, lo sea; y viceversa.

Dado que la imagen puede tener unas dimensiones distintas a las del tamaño del código que vamos a considerar, gran parte de la complejidad del proceso viene dada por la manera de deividir la imagen en bloques del tamaño adecuado, componerla y recomponerla.

Veamos primero qué papel juega esto en la codificación.

6.1. Codificación

Se ha elegido, con el motivo de proporcionar una visualización clara del proceso de corrección, mantener una codificación sistemática. Esto es, los datos se encuentran inalterados después de la codificación, simplemente se añade redundacia.

Para conseguir esto a partir de un codificador sistemático de un código Reed-Solomon (como el proporcionado por la librería `galois`), primero se creó una función para obtener la redundacia de un vector de datos de longitud arbitraria, dividiendolo en bloques del tamaño requerido por el código y concatenando las redundancias de cada bloque.

```python
def encode_row(row : galois.FieldArray,rsCode:galois.ReedSolomon) -> galois.FieldArray:
    nChunks = len(row) // rsCode.k
    GF = rsCode.field
    if len(row) % rsCode.k == 0:
        nChunks = nChunks - 1
    parity = []
    for i in range(nChunks):
        parity.append(
            rsCode.encode(GF(row[i * rsCode.k : (i + 1) * rsCode.k]), output="parity")
        )
    # Bloque posiblemente incompleto
    parity.append(rsCode.encode(GF(row[rsCode.k * nChunks :]), output="parity"))

    parity = [np.array(arr) for arr in parity]
    return np.concatenate(parity)
```

Una vez se tiene esto, es sencillo aplicarlo a todas las filas de una matriz (imagen). Transponiendo el resultado de codificar las filas, podemos aplicar de nuevo lo mismo para codificar las columnas y después deshacer la transposición.

```python
def encode_image_rows(data,rsCode) :
    image_size = data.shape
    encoded_rs = np.array([encode_row(row,rsCode) for row in data])
    # Out:
    encoded_data = np.zeros(
        shape=(
            image_size[0],
            encoded_rs.shape[1] + image_size[1],
        ),
        dtype="uint8",
    )
    encoded_data[:, : image_size[1]] = data
    encoded_data[:, image_size[1] :] = encoded_rs
    return encoded_data

def encode_image(data,rsCode):
   return encode_image_rows(
     encode_image_rows(data,rsCode).transpose((1,0)),
     rsCode
   ).transpose((1,0))
```

Veamos esto con un ejemplo. Por sencillez se trabajará con una versión en blanco y negro de las imágenes. De esta manera tendrá un solo canal : cada píxel se correspondrá con un único byte. En la aplicación web se trabaja con imagenes a color pero pasar de uno a tres canales (RGB) es trivial, simplemente se superponen los resultados de aplicar el algoritmo sobre cada uno de los canales.

Un resultado de esta manera de codificar la información RGB es que, debido a que la decodificación de un canal también es independiente de la de los otros dos, se puede dar una corrección parcial de un píxel. Por ejemplo, se obtienen los valores correctos de dos de los canales pero el tercero es incorregible. En este caso se acercará más al aspecto del píxel original que un fallo de corrección total.

```
from PIL import Image

im = Image.open("paisaje.jpg").convert('L')
print('Imagen sin codificar:')
display(im)
data = np.array(
    im
)
rsCode = galois.ReedSolomon(255,230) #k puede variar
encoded_data = encode_image(data,rsCode)
print('Imagen codificada:')
im = Image.fromarray(encoded_data)
display(im)
```

Se ha usado la librería `PIL` para cargar la imagen y transformarla en un array de `numpy`. Además ene este ejemplo concreto se está usando una longitud de bloque de 230 lo que resulta en 25 bytes de redundancia por bloque. Este parametro de la dimensión del código k, puede variar. El resultado de el ejemplo se muestra en la figura 6.1 se puede observar que se ha añadido la redundancia al final de la imagen, que aparece representada como ruido.

En una aplicación práctica no veríamos en ningún momento los datos codificados ni la redundancia de esta manera, los datos codificados se decodifican primero antes de mostrarse, para obtener la imagen original. Se ha decidido representar la redundancia para tener una mejor visualización de los datos reales que se envian y tener de un vistazo una idea del porcentaje de la información que representa la imagen original frente a la redundancia.

(a) Imagen original

(b) Imagen codificada

Figura 6.1: Resultado de la codificación

6.2. Descomposición en bloques

Para poder descodificar facilmente descompondremos de nuevo la imagen en bloques, recuperando para cada bloque de datos la redundancia correspondiente. Esto se hace localizando 4 secciones rectangulares en la imagen codificada: El bloque de datos (de la imagen original), la redundancia de las filas, la redundancia de las columnas, y la parte correspondiente a la redundancia de la redundancia. Siguiendo las 4 secciones del esquema presentado en la figura 4.2.

La idea conceptual es sencilla pero la lógica se vuelve algo complicada al tener que acomodar los bloques incompletos en los extremos inferior y derecho de la imagen.

Las variables `remRow`, `remCol` se utilizan para calcular el tamaño de estos bloques incompletos (si existen, si se alinean los bloques perfectamente con la imagen estas variables son 0).

Los bloques completos tienen dimensión (n, n) mientras que los incompletos de la parte inferior tienen dimensión (`remCol` $+(n-k), n)$, los de la parte derecha (n,`remRow`$+(n-k)$) y el bloque de la esquina inferior derecha (`remCol` $+(n-k)$,`remRow`$+(n-k)$). Esto se debe a que la redundancia añadida es la misma aún si se codifican menos de k bytes, automáticamente se utiliza un código acortado.

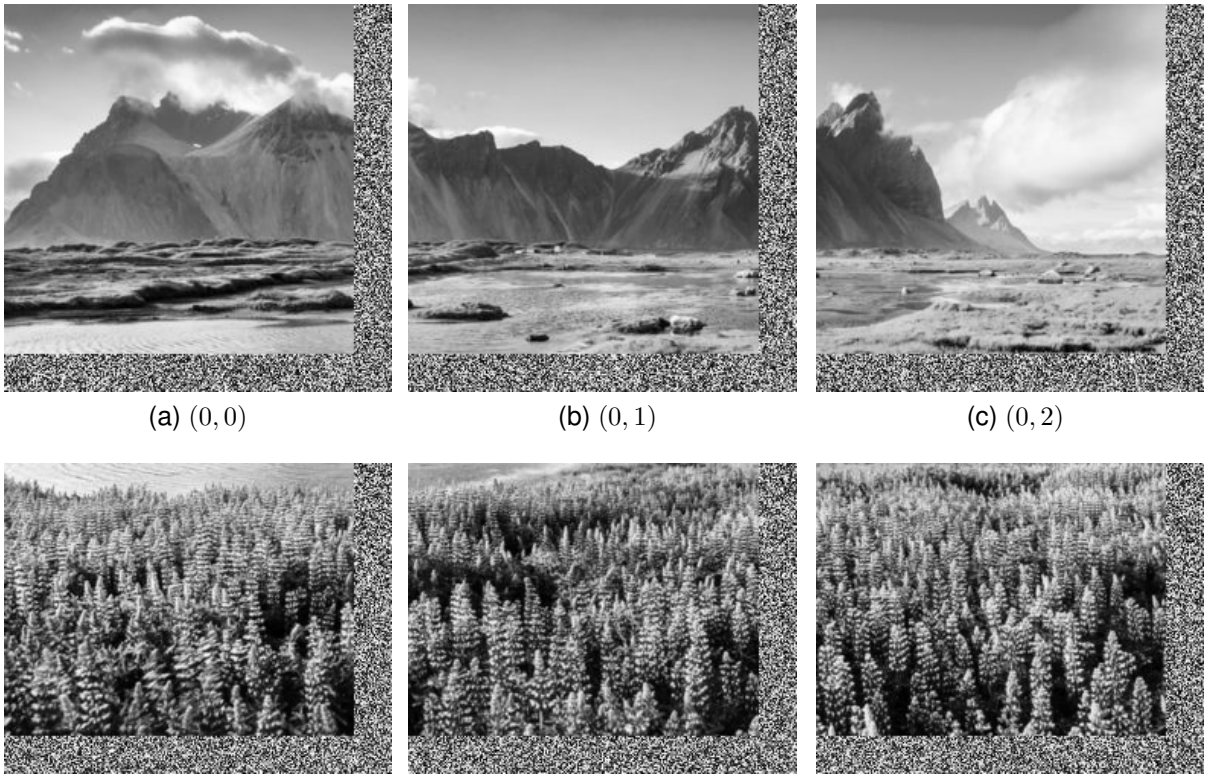
Los códigos acortados se utilizan para codificar mensajes de otras longitudes completando hasta k con ceros y codificando ese mensaje, igualmente el codificador al recibir una palabra corta, introduce entre el mensaje y la redundacia ceros hasta llegar a n y decodifica esa palabra.

```python
def divide_image(data, original_size, rsCode):
    k = rsCode.k
    n = rsCode.n

    nColChunks = original_size[0] // rsCode.k
    remCol = original_size[0] % rsCode.k

    nRowChunks = original_size[1] // rsCode.k
    remRow = original_size[1] % rsCode.k

    blocks = []
    for j in range(nColChunks):
        block_row = []
        for i in range(nRowChunks):
            block = np.zeros(shape=(n, n), dtype="uint8")
            block[:k, :k] = data[k * j : k * (j + 1), k * i : k * (i + 1)]
            block[k:, k:] = data[
                original_size[0] + j * (n - k) : original_size[0] + (j + 1) * (n - k),
                original_size[1] + i * (n - k) : original_size[1] + (i + 1) * (n - k),
            ]
            block[:k, k:] = data[
                k * j : k * (j + 1),
                original_size[1] + i * (n - k) : original_size[1] + (i + 1) * (n - k),
            ]
            block[k:, :k] = data[
                original_size[0] + j * (n - k) : original_size[0] + (j + 1) * (n - k),
                k * i : k * (i + 1),
            ]
            block_row.append(block)
        if remRow != 0:
            block = np.zeros(shape=(n, remRow + (n - k)), dtype="uint8")
            block[:k, :remRow] = data[
                k * j : k * (j + 1), rsCode.k * nRowChunks : original_size[1]
            ]
            block[k:, remRow:] = data[
                original_size[0] + j * (n - k) : original_size[0] + (j + 1) * (n - k),
                original_size[1] + nRowChunks * (rsCode.n - rsCode.k) :,
            ]
            block[:k, remRow:] = data[
                k * j : k * (j + 1),
                original_size[1] + nRowChunks * (rsCode.n - rsCode.k) :,
            ]
            block[k:, :remRow] = data[
                original_size[0] + j * (n - k) : original_size[0] + (j + 1) * (n - k),
                rsCode.k * nRowChunks : original_size[1],
            ]
            block_row.append(block)
        blocks.append(block_row)
```

```
if remCol != 0:
        lastRow = []
        for i in range(nRowChunks):
            block = np.zeros(shape=(remCol + (n - k), n), dtype="uint8")
            block[:remCol, :k] = data[
                rsCode.k * nColChunks : original_size[0], k * i : k * (i + 1)
            ]
            block[remCol:, k:] = data[
                original_size[0] + nColChunks * (n - k) :,
                original_size[1] + i * (n - k) : original_size[1] + (i + 1) * (n - k),
            ]
            block[
                remCol:,
                :k,
            ] = data[original_size[0] + nColChunks * (n - k) :, k * i : k * (i + 1)]
            block[:remCol, k:,] = data[
                rsCode.k * nColChunks : original_size[0],
                original_size[1] + i * (n - k) : original_size[1] + (i + 1) * (n - k),
            ]
            lastRow.append(block)
        if remRow != 0:
            block = np.zeros(shape=(remCol + (n - k), remRow + (n - k)), dtype="uint8")
            block[:remCol, :remRow] = data[
                rsCode.k * nColChunks : original_size[0],
                rsCode.k * nRowChunks : original_size[1],
            ]
            block[remCol:, remRow:] = data[
                original_size[0] + nColChunks * (n - k) :,
                original_size[1] + nRowChunks * (n - k) :,
            ]
            block[:remCol, remRow:] = data[
                rsCode.k * nColChunks : original_size[0],
                original_size[1] + nRowChunks * (n - k) :,
            ]
            block[remCol:, :remRow] = data[
                original_size[0] + nColChunks * (n - k) :,
                rsCode.k * nRowChunks : original_size[1],
            ]
            lastRow.append(block)
        blocks.append(lastRow)

    return blocks
```

La función produce una matriz de bloques `blocks` de tal manera que `blocks[i][j]` es el bloque de la i-ésima fila y j-ésima columna de bloques.

Aplicando esta función a la imagen codificada del ejemplo de la sección anterior se obtienen 6 bloques según se muestra en la figura 6.2



(a) $(0,0)$ (b) $(0,1)$ (c) $(0,2)$

(d) $(1,0)$ (e) $(1,1)$ (f) $(1,2)$

Figura 6.2: División por bloques

Ahora se puede corregir cada uno de esos bloques independientemente, pero necesitamos una manera de revertir el proceso y recuperar la imagen original a partir de los bloques. Pero esto es mucho más sencillo si tenemos en cuenta que recuperar la imagen es necesario para presentarla ya decodificada, entonces no es necesario devolver la redundancia a su posición.

Se trata solamente por lo tanto de extraer ordenadamente de cada bloque el trozo de la imagen y colocarlo en su lugar correspondiente dentro de una matriz que se incializa a las dimensiones de la imagen original, aunque de nuevo los bloques incompletos deben de ser tratados de manera distinta.

Se itera sobre las filas de bloques, tratando por separado el último bloque de cada una de estas. También se trata de manera la última fila de bloques entera, y dentro de estas, el último bloque también representa una excepción.

```python
def join_image(blocks,original_size,rsCode):
    n = rsCode.n
    k = rsCode.k

    joined_image = np.zeros((original_size[0],original_size[1]), dtype='uint8')
    remRow = blocks[0][-1].shape[1] - (n-k)
    for j in range(len(blocks)-1):
        block_row=blocks[j]
        for i in range(len(block_row)-1):
            block = block_row[i]
            joined_image[k * j : k * (j + 1), k*i : k*(i+1)] = block[:k,:k]
        block = block_row[-1]
        joined_image[k * j : k * (j + 1),-remRow:] = block[:k,:remRow]
    #last row
    block_row=blocks[-1]
    remCol = block_row[0].shape[0] - (n-k)
    for i in range(len(block_row)-1):
        block = block_row[i]
        joined_image[-remCol:,k*i : k*(i+1)] = block[:remCol,:k]
    block = block_row[-1]
    joined_image[-remCol:,-remRow:] = block[:remCol,:remRow]

    return joined_image
```

6.3. Decodificación

Como ya hemos mencionado, ahora solo tenemos que preocuparnos de desarrollar una función que decodifique cada bloque, ya que se van a decodificar por separado.

```python
def decode_block_by_rows(block,rsCode):
    stats={
        "uncorrected blocks" : 0,
        "errors" : 0
    }
    decoded = np.zeros_like(block)
    for j in range(block.shape[0]):
        decoded[j, :], e = rsCode.decode(block[j, :],errors=True,output="codeword")
        if e == -1 :
            stats["uncorrected blocks"] = stats["uncorrected blocks"] + 1
        else:
            stats["errors"] = stats["errors"] + e
    return decoded,stats

def decode_block_by_cols(block,rsCode):
    decoded , stats = decode_block_by_rows(block.transpose((1,0)),rsCode)
    return decoded.transpose((1,0)) , stats
```

La lógica llegados a este punto es sencilla ya que en cada fila y columna de un bloque

tenemos una palabra del código que podemos pasar al decodificador. Podemos volver a utilizar la técnica de transponer la imagen para decodificar por columnas.

Además de la decodificación, se recogen estadísticas sobre cuantos errores se han corregido, y cuantas filas/columnas han sido incorregibles.

A continuación damos una función, que utilizando estas, decodifica o bien por filas o bien por columnas, según un parámetro dado `mode` todos los bloques que se pasan con el formato dado por la función `divide_image` presentada. Además agrega las estadísticas parciales de cada bloque para dar un cómputo global de los errores.

```
def decode_blocks(blocks,rsCode,mode='row'):
    stats={
        "uncorrected blocks" : 0,
        "errors" : 0
    }
    decode_fun = decode_block_by_rows if mode == 'row' else decode_block_by_cols
    for block_row in blocks:
        for j in range(len(block_row)):
            block_row[j], ps = decode_fun(block_row[j],rsCode)
            for k in stats:
                stats[k] =stats[k] + ps[k]
    return blocks,stats
```

Finalmente ya podemos dar el algoritmo iterativo en su completitud. Como ya hemos dicho el concepto es muy simple, se alternará la decodificación por filas con la decodficación por columnas hasta que no se encuentren más errores.

Este es el algoritmo que con modificaciones mínimas (para guardar las estadísticas e imágenes intermedias) se ha utilizado en la aplicación web.

Sin embargo para dar un ejemplo tendremos que crear una función que introduzca un error en la imagen. Van a ser errores aleatorios, que se podrán introducir con una densidad determinada ρ.

Es decir, si se utiliza con $\rho = 0{,}05$ esto significaría que el $5\,\%$ de los pixeles van a contener un error. Esta es la posibilidad que se le darña al usuario en la interfaz gráfica en conjunto con *rayar* la imagen a mano alzada. Además esto será útil más adelante cunado comprobemos la capacidad correctora del algoritmo experimentalmente.

```
def iterative_decode_image(data,original_size,rsCode,priting= "final"):
    errors = -1
    it = 0
    blocks = divide_image(data,original_size,rsCode)
    while(errors):
        it = it +1
        if (it%2) :
            mode = 'row'
        else:
            mode = 'col'
        blocks, stats = decode_blocks(blocks,rsCode,mode)
        errors = stats["errors"]
    im = Image.fromarray(join_image(blocks,original_size,rsCode))
    return it,im
```

Y la función para corromper la imagen:

```
from scipy import sparse

def corrupt_data(data,rho=0.05):
    corrupted_data = np.zeros_like(data)
    corrupted_data = data + sparse.random(
        data.shape[0], data.shape[1], density=rho, dtype="uint8")
    return corrupted_data
```

6.4. Ejemplo

Vamos a introducir un 6 % de error en al codificación que hemos dado de ejemplo en el capítulo y vamos a aplicar el algoritmo iterativo.

En la figura 6.3 se puede ver el resultado de corromper la imagen.

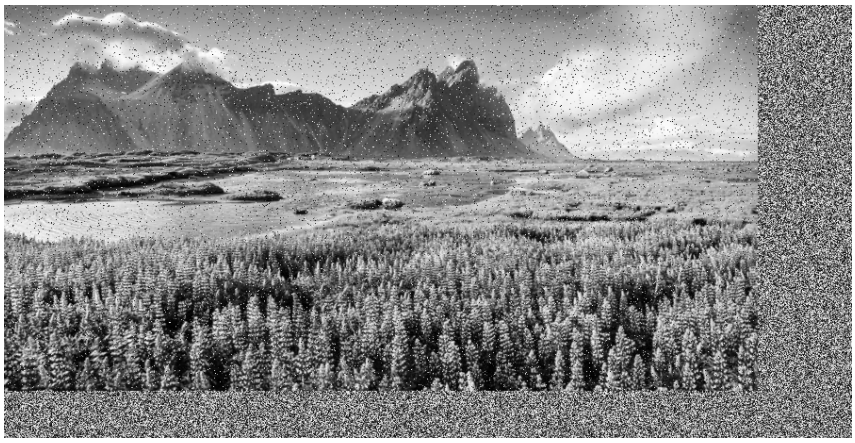


Figura 6.3: Imagen corrupta

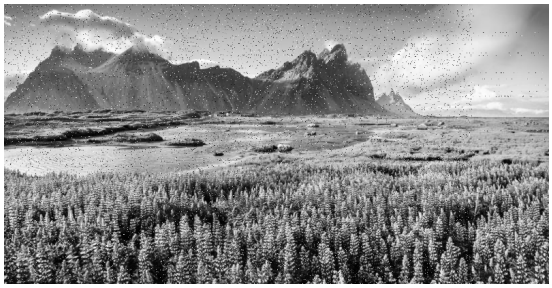
El resultado por iteraciones de la decodifciación usando el algoritmo se presenta en 6.4. En este ejemplo hay varias cosas resaltables.

En primer lugar, se puede observar como en la segunda iteración el extremo derecho queda ya libre de errores, mientras que el resto de la imagen va a requerir de más iteraciones. Esto se debe a que, como ya hemos mencionado, los bloques extremos son incompletos lo que resulta en una proporción de redundancia con respecto de información útil mucho mayor, y es esta proporción la que determina la capacidad correctora del código.

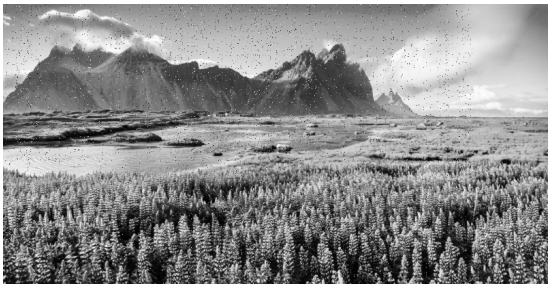
Por otra parte, podemos ver como en la quinta iteración la imagen está ya completamente corregida y sin embargo es necesario una última iteración para comprobarlo, ya que el algoritmo para cuando ya no se puede corregir más errores.

El algoritmo ha sido capaz de corregir un 6 % de errores en 6 % iteraciones usando uan dimensión de los códigos constituyentes $k = 230$. Pero, ¿Podría haber corregido más errores? ¿Se puede predecir cuántas iteraciones son necesarias?

En la siguiente sección trataremos de responder a estas preguntas haciendo uso de los datos recogidos durante ejecuciones repetidas del algoritmo, es decir, experimentalmente.



(a) Iteración 1: Se han corregido 6805 errores,y no se han podido corregir 1072 filas.



(b) Iteración 2: Se han corregido 9283 errores,y no se han podido corregir 558 columnas.

(c) Iteración 3: Se han corregido 6075 errores,y no se han podido corregir 172 filas.

(d) Iteración 4: Se han corregido 2069 errores,y no se han podido corregir 32 columnas.

(e) Iteración 5: Se han corregido 456 errores,y no se han podido corregir 0 filas.

(f) Iteración 6: Se han corregido 0 errores,y no se han podido corregir 0 columnas.

Figura 6.4: Progreso del algoritmo iterativo.

6.5. Tests y rendimiento

Nos dedciamos primero al problema de obtener una estimación de la densidad de errores máxima corregible.

Para ello he definido el problema de la siguiente manera.

Dada una dimensión k de los códigos constituyentes se busca estimar una densidad de errores aleatorios ρ tal que la probabilidad de no corregir completamente datos con esa densidad de errores sea $0<p<t$, donde t representa un umbral de presición pequeño.

Para ello primero tenemos el problema de estimar tal probabilidad dada una densidad, ya que se trata de un proceso que involucra aleatoriedad.

Además cada ejecución del algoritmo es bastante costosa en términos de recursos computacionales, luego se busca intetar obtener una buena aproximación usando el menor número posible de ejecuciones.

Para minimizar las ejecuciones podemos valernos del hecho de que no nos interesa conocer con precisión el valor de probabilidad de error si es grande. Por ello, si se fallan un número `FAILURES` de veces el algoritmo para y devuelve su estimación. También se introduce un limite duro al máximo de ejecuciones.

Finalmente, a modo de otra pequeña optimización, se tiene también un número mínimo de iteraciones `TRIVIAL_ITS`. El número elevado de iteraciones es un buen indicador de que estamos en la zona crítica que queremos explorar, ya que un número bajo significa que el algoritmo lo resuelve fácilmente o que rápidamente deja de poder corregir errores. Al rededor del valor límite de densidad que buscamos, se necesitan más iteraciones para corregir correctamente.

```python
# Estimate probability of succesfull decoding
def estimateProbabilityOfSuccess(density,k,encoded_data):
    TRIVIAL_ITS = 5
    FAILURES = 4
    MAX = 40
    num_failures = 0
    while True:
        runCount = 0
        while True:
            runCount = runCount + 1
            corrupted_data = corrupt_data(encoded_data, density)
            im = Image.fromarray(corrupted_data)
            it, im = iterative_decode_image(
                corrupted_data, original_size, rsCode, printing="none"
            )
            success = (np.array(im) == 0).all()
            if it <= TRIVIAL_ITS:
                return int(not success)
            if not success:
                num_failures = num_failures + 1
                if num_failures == FAILURES:
                    return FAILURES / runCount
            if runCount >= MAX:
                return num_failures / runCount
```

Los valores que se muestran en el código para los distintos parámetros se encontraron a base de prueba y error, procurando encontrar buenas estimaciones en los casos que nos interesan impactando en la menor medidad posible en el rendimiento.

Podemos utilizar esta función para navegar el conjunto de las posibles densidades y encontrar una estimación del valor que nos interesa.

Lo haremos de la siguiente manera: Se comienza con una densidad base cualquiera y se itera calculando la probabilidad de fallo.

En cada iteración si la probabilidad es 0, esto significa que el algoritmo ha corregido correctamente en todas las ocasiones (La confianza que tenemos en estos valores viene determinada por supesto, por el número máximo de ejecuciones que era parámetro de la anterior función para estimar la probabilidad), en este caso se realiza un incremento porcentual en la densidad dado por `delta`. En caso de que la probabilidad sea mayor que nuestro límite buscado t (`threshold`) entonces había demasiados errores y la densidad se disminuye.

La optimización principal de este algoritmo es que el paso `delta` se reduce cuando se da un cambio de dirección. Por ejemplo, si se viene incrementando en los ultimos pasos la densidad y ahora determinamos que hay que decrementarla, eso significa que el valor buscado está en algún lugar intermedio y hay que investigar ese area de manera más detenida: con un paso más pequeño.

Se ha permitido al algoritmo devolver densidades que obtuvieron probabilidad 0 cuando el último valor probado en estas circunstancias está muy proximo (`epsilon`).

Por último se ha añadido un valor mínimo (`DELTA_MIN`) al paso *delta*, para no variar la densidad en órdenes de magnitud que nos son irrelevantes y un multiplicador para el cambio cuando se realiza un incremento. Esta última decisión está justificada por el hecho que, de la manera en que estimamos la probabilidad, es mucho más rápido aproximarse desde arriba a la solución.

Veamos ahora los resultados de estos tests. Se ha aplicado la función de test a dimensiones desde $k = 155$ hasta $k = 240$ con un incremento de 5. Además de los parámetros visibles en el código, para estos tests se ha tomado t (`threshold`)$= 0{,}05$, es decir se buscan densidades que permitan una tasa de fallo no mayor al 5 %.

Para relativizar estos resultados los compararemos con la capacidad correctora del código producto dada por su distancia mínima

$$\frac{D-1}{2}$$

y a su vez la distancia minima del código producto viene dada por el producto de las distancias mínimas de los códigos que lo componen

$$d = n - k + 1 \quad D = d^2$$

```python
# Test para encontrar la capacidad correctora máxima
def run_test(k, baseDensity, threshold=0.05, epsilon=0.001):
    rsCode = galois.ReedSolomon(255, k)
    data = np.zeros((k, k), dtype="uint8")
    encoded_data = encode_image(data, rsCode)
    original_size = data.shape

    density = baseDensity
    delta = 0.5
    direction = 1
    DELTA_CHANGE = 1.7
    UPDELTA_MULTIPLIER = 5
    DELTA_MIN = 0.0005
    last_ok = -100
    while True:
        p = estimateProbabilityOfSuccess(density)
        if p > 0 and p <= threshold:
            return (p, density)
        if p > threshold:
            if direction == 1:
                delta = (
                    (delta / DELTA_CHANGE) * (p - threshold)
                    if (delta / DELTA_CHANGE) * (p - threshold) > DELTA_MIN
                    else DELTA_MIN
                )
            density = density * (1 - delta)
            direction = -1
        elif p == 0:
            if abs(density - last_ok) < epsilon:
                return (p, density)
            last_ok = density
            if direction == -1:
                delta = (
                    delta / DELTA_CHANGE
                    if delta / DELTA_CHANGE > DELTA_MIN
                    else DELTA_MIN
                )
            density = density * (1 + UPDELTA_MULTIPLIER * delta)
            direction = 1
```

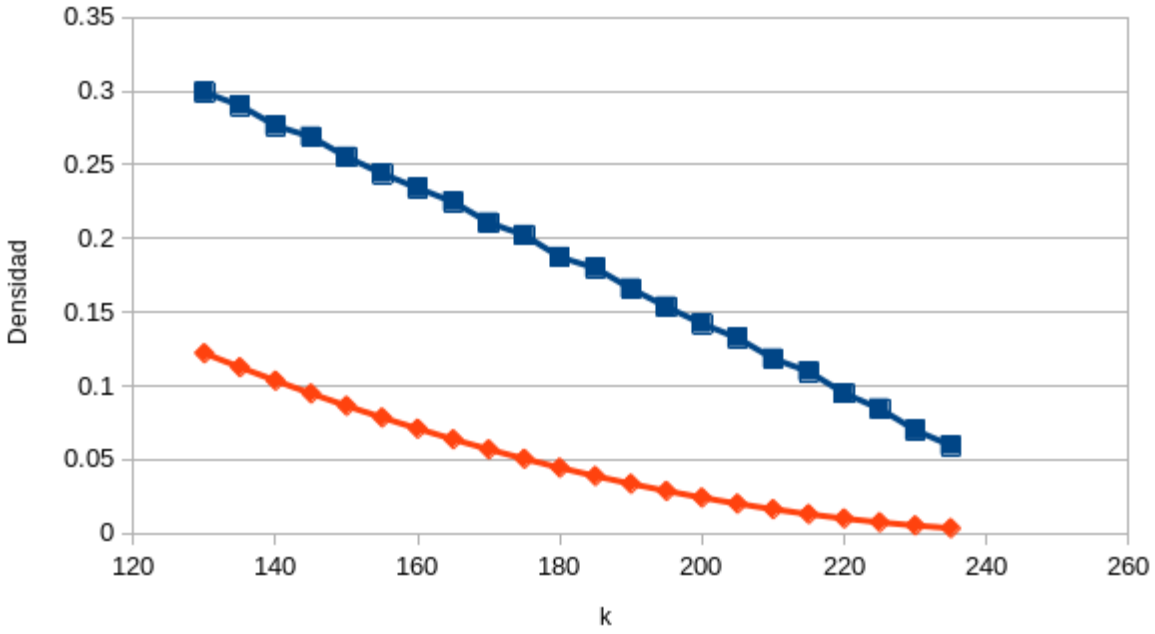


Figura 6.5: Relación entre dimensión y capacidad correctora del algoritmo iterativo (Azul) y dada por la distancia mínima (Naranja).

En la figura 6.5 se muestra la relación entre dimesión del código y capacidad correctora. La capacidad correctora dada por la distancia minima también ha sido escalada para expresar una densidad de errores y no un número de errores corregibles.

Se aprecia que el algoritmo iterativo ofrece una mejora clara. La capacidad correctora dada por la distancia minima garantiza que se pueda corregir cualquier patrón de error con esa densidad, por otra parte, el algoritmo iterativo corrige los errores solamente en la mayoría de los casos a cambio de una capacidad correctora mucho mayor.

Por otra parte, mientras que la capacidad correctora dada por la distancia mínima sigue una relación cuadrática con la dimensión del código, se puede apreciar a priori una relación lineal en la capacidad correctora del algoritmo iterativo.

A continuación pasamos a estudiar el número de iteraciones necesarias, de media, para corregir una imagen dada una densidad y una dimensión.

En primer lugar vamos a calcular la media de iteraciones necesarias tomando las densidades máximas halladas previamente, aplicando el algoritmo 50 veces para cada pareja de parámetros.

Aprovechamos además para validar estas estimaciones de la densidad máxima de errores observando también la tasa de error en estas ejecuciones. Los resultados se presentan en la tabla 6.2.

Se puede observar que la tasa de acierto corresponde en general con lo buscado. La hipótesis era que las iteraciones serían similares si se introduce una proporción de errores similiar *relativa* a la densidad máxima, e independiente del parámetro de dimensión. Sin

embargo parece que esto no es del todo cierto, aunque para parámetros k bajos (esto es se tiene mucha redudancia) el número de iteraciones es bastante uniforme, para parámetros altos se aprecian valores de media de iteraciones superiores.

En conclusión, hemos cuantificado la mejora que supone el algoritmo iterativo en comparación con la corrección de errores garantizada por la distancia mínima pero también hemos podido ver cómo a medida que nos acercamos a la capacidad máxima de corrección de errores se incurre en un costo computacional (número de iteraciones) mucho mayor y que crece rápidamente cerca de este límite.

Tabla 6.1: Datos del gráfico de la figura 6.5.

k	**Iterativo**	**Distancia mínima**
130	0.299	0.122
135	0.290	0.113
140	0.277	0.103
145	0.269	0.095
150	0.255	0.086
155	0.244	0.078
160	0.234	0.071
165	0.225	0.064
170	0.211	0.057
175	0.202	0.050
180	0.187	0.044
185	0.180	0.039
190	0.166	0.033
195	0.154	0.029
200	0.142	0.024
205	0.132	0.020
210	0.118	0.016
215	0.110	0.013
220	0.095	0.010
225	0.084	0.007
230	0.070	0.005
235	0.059	0.003

k	$.75\rho_{max}$	$.90\rho_{max}$	$.95\rho_{max}$	$.97\rho_{max}$	ρ_{max}	**Tasa de acierto con** ρ_{max}
130	3.0	4.0	5.0	5.6	9.28	0.96
135	3.0	4.0	5.0	5.8	9.1	1.0
140	3.0	4.0	5.0	5.76	8.82	0.96
145	3.0	4.0	5.0	6.0	9.4	0.9
150	3.0	4.0	5.0	5.92	9.2	1.0
155	3.0	4.0	5.0	5.58	7.44	1.0
160	3.0	4.0	5.2	6.12	10.54	0.94
165	3.0	4.0	5.24	6.24	10.08	0.9
170	3.0	4.0	5.34	6.12	9.5	0.96
175	3.0	4.04	5.36	6.22	9.18	0.96
180	3.0	4.04	5.2	6.0	8.44	1.0
185	3.0	4.58	5.86	6.8	10.92	0.96
190	3.0	4.88	5.98	6.92	10.56	0.96
195	3.0	4.3	5.26	6.02	8.18	1.0
200	3.1	4.98	6.0	6.84	10.28	1.0
205	3.26	5.0	5.92	6.86	9.38	1.0
210	3.54	5.0	6.04	7.0	9.98	1.0
215	3.98	5.06	6.92	8.04	12.44	0.9
220	4.0	5.36	7.06	8.2	13.36	0.98
225	4.0	5.34	6.78	7.64	10.52	1.0
230	4.0	6.02	7.8	9.18	14.62	0.96
235	4.02	6.04	7.74	8.86	12.26	1.0

Tabla 6.2: Media de iteraciones con un porcentaje dado de la densidad máxima.

Capítulo 7

Conclusiones

A continuación, en este capítulo, se discutirá si se han conseguido los objetivos que se detallaron en el primer capítulo, así como ventajas e incovenientes de la solución propuesta, en qué destaca este trabajo en relación con otros comparables y líneas de trabajo futuro.

7.1. Consecución de los objetivos

En cuanto a los objetivos teóricos creo que puedo decir que han sido alcanzados con creces.

Antes de comenzar este trabajo mi entendimiento de los códigos correctores en general se limitaba a una idea intituiva mientras que ahora puedo exponer con confianza los temas que se han mencionado aquí y otros cuantos relacionados con los que me he cruzado en el camino, que me han hecho descubrir que este área de estudio es mucho más extensa de lo que había anticipado.

Por supuesto mucho de esto se debe al trabajo final del grado de matemáticas, complementario a este, para el cual he tenido que lidiar con libros de texto y publicaciones académicas durante horas; sin embargo no puedo destacar lo suficiente la gran medida en que implementar los algoritmos de decodificación y verlos en acción ha ayudado a comprenderlos a otro nivel.

También me he esforzado por transmitir de manera clara en este texto, parte de lo que he podido aprender a lo largo del proyecto.

A nivel práctico, dado que el componente teórico del proyecto ya era grande, desde un primer momento el objetivo fue mantener un enfoque en la sencillez, estando los límites de la aplicación web bien establecidos.

Opino que la aplicación web ofrece una manera simple de observar el algoritmo iterativo en acción, de una manera más interactiva que a lo que se puede llegar con cuadernos Jupyer además de permitir la divulgación a un público más amplio que quizás no tenga acceso sencillo a las herramientas de desarrollo que se necesitan para ejecutar Jupyter.

Para el proceso de desarrollo se ha utilizado Flask, un framework minimalista que encaja bien con el enfoque limitado que se ha impuesto. Me gustaría destacar que esta

herramienta no es la que se utilizada en el grado para el desarrollo web, con que lo añado a la larga lista de cosas que he aprendido durante el proyecto.

La aplicación web a mi juicio cumple perfectamente los requisitos que se habían impuesto, permite la carga de imágenes del usuario, permite corromper la imagen a mano alzada y con ruido aleatorio y muestra de manera clara el proceso de decodificación.

7.2. Ventajas e incovenientes de la solución

Se ha hecho una decisión consciente a la hora de elegir las herramientas y la arquitectura en favor de la sencillez y velocidad de desarrollo en detrimento de la escalabilidad de la aplicación.

Flask es minimalista pero puede configurarse de manerae que permita mejorar la escalabilidad pero también no hemos usado ningún framework (más allá de las plantillas Jinja, muy simples) para el desarrollo de interfaz web lo cuál podría suponer un problema en caso de tener que escalar la aplicación.

Consideré que plantear la aplicación de esta manera era beneficioso en general ya que el limite de la aplicación estaba muy bien definido y no era muy grande.

7.3. Líneas de trabajo futuras

El algoritmo iterativo es el más visual, pudiendo ver los pasos intermedios del proceso, pero en líneas de trabajo futuras se podría intentar crear una visualización de otros algoritmos o implementarlos para permitir la comparación entre algoritmos.

Otra posibilidad es implentar una técnica llamada *interleaving*, intercalación, que mezcla los pixeles antes de codificarlos para mejorar la capacidad de corregir errores contiguos (de ráfaga).

Además, la aplicación y este trabajo se ha limitado a códigos Reed-Solomon pero se podría adaptar fácilmente la aplicación para utilizar cualquiera de los códigos implementados en la librería `galois` y el algoritmo iterativo funcionaría sin más modificaciones, se permitiría así la comparación entre distintas posibilidades.

Finalmente también se podría investigar la posibiilidad de otra manera de tratar las imágenes. Actualmente se codifica cada uno de los canales RGB de manera independiente, pero se podrían combinar de alguna manera o considerar la posibilidad de utilizar otra unidad de información que no fuera el byte.

Manuales

Manual de Instalación

Jupyter

En el trabajo se incluyen 4 cuadernos jupyter. A continuación se detalla como se pueden ejecutar estos.

Primero tenemos que asegurarnos de que Python3 esté instalado en nuestro sistema, podemos seguir los pasos en

`https://www.python.org/downloads/`

para el sistema operativo necesario.

Una vez tenemos Python instalado, el proceso para instalar y ejecutar jupyer es el mismo buien desde el símbolo del sístema o Powershell desde Windows o desde la terminal en Linux ya que usaremos el gestor de paquetes de Python, `pip`.

```
> pip install jupyter
```

Antes de ejecutar el jupyer debemos asegurarnos de instalar los paquetes que se han utilizado en los cuadernos. Junto con los cuadernos se inlcuye un archivo de dependencias `requirements.txt` que podemos utilizar para instalar todas de golpe.

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

o lo que es equivalente

```
> pip install numpy galois Pillow scipy
```

Ahora podemos ejecutar el siguiente comando, que abre una interfaz web en nuestro navegador por defecto para jupyter que mostrará nuestro sistema de archivos y podemos navegar hasta la carpeta en que tengamos los cuadernos y abrirlos.

```
> jupyter notebook
```

Figura 7.1: Interfaz de jupyter.

Una vez abramos un cuaderno podemos ir ejecutando las celdas, entendiendo de manera interactiva el código. (Ver figura 7.1).

Aplicación Web

En esta sección veremos como desplegar la aplicación web, el proceso de desarrollo y pruebas se ha hecho desplegandola en el entorno local (`localhost`) pero los pasos serían equivalentes en un servidor que hiciera el puerto al que está asociada la aplicación disponible a través de internet.

Para el despliegue nos situamos en la carpeta `webapp`. Se dan dos opciones para el despliegue, la primera es en un contenedor de docker. También necesitaremos la herramienta docker-compose. Para la instalación en windows nos referimos a la guía de Microsoft

`https://learn.microsoft.com/es-es/virtualization/windowscontainers/quick-start/set-up-environment?tabs=dockerce`.

Para la instalación en linux, esta herramienta se suele encontrar en los repositorios por defecto de los gestores de paquetes, en mi caso apt:

```
> sudo apt install docker docker-compose
```

Una vez tengamos estas herramientas disponibles, podemos simplemente ejecutar el siguiente comando

```
> docker-compose -up
```

Que se encargará de configurar el contenedor de docker, descargando la imagen del contenedor para python, instalando las dependencias necesarias y finalmente ejecutando el comando para correr el servidor de Flask.

En el archivo de configuración `compose.yaml` se define en que puerto se ejecutará la aplicación, por defecto está configurado en el puerto 5000, si se desea cambiar por ejemplo al puerto 80, deberíamso cambiar la línea de configuración `ports` a `- "5000:80"`. Esto relaciona el puerto 5000 dentro del contenedor con el puerto 80 de nuestro servidor.

Si todo funciona correctamente la consola mostrará los pasos que se siguen para levantar el contenedor y finalmente un mensaje que indica que se está escuchando en el puerto indicado.

```
web_1  |  * Debug mode: off
web_1  | WARNING: This is a development server. Do not use it in a
    production deployment. Use a production WSGI server instead.
web_1  |  * Running on all addresses (0.0.0.0)
web_1  |  * Running on http://127.0.0.1:5000
web_1  |  * Running on http://172.18.0.2:5000
web_1  | Press CTRL+C to quit
```

La aplicación estará en ese momento corriendo y usable.

Por otra parte si se quiere evitar el uso de contenedores tendremos que movernos a la carpeta `app`, dentro de `webapp`. Aquí primero tenemos otro archivo con las dependencias para la aplicación web, de manera similar a las dependencias para los cuadernos de Jupyter ejecutamos el siguiente comando para instalarlas todas.

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

Ahora podemos ejecutar el comando para iniciar Flask

```
> python3 -m flask run
```

Despues de unos momentos deberíamos ver el mismo mensaje de que todo está ejecutándose correctamente.

Despliegue real

En esta sección se explica brevemente la solución utilizada para el despliegue de la aplicación, disponible en `http://www.singacom.uva.es/DVDdecoding`. Aprovecho para dar las gracias al grupo de investigación SINGACOM por ofrecerme presentar mi aplicación en su web.

En este caso, las circunstancias del despliegue son particulares ya que la aplicación se tiene que servir en una ruta coexistiendo con la página web previamente presente, además la tecnología usada por esta es diferente (`Apache` + `PHP`).

Tras probar varias opciones, finalmente se ha desplegado con la ayuda de la directiva `ProxyPass` del modulo de `Apache` `mod_proxy`, de manera que `Apache` actua como un proxy inverso al manejar las peticiones en la ruta determinada redigiendolas a otro puerto del mismo servidor.

En este puerto se ejecuta una instancia de `waitress`, un servidor ligero de aplicaciones `WSGI`, ya que el incluido con Flask es más apropiado para el desarrollo.

Por otra parte haciendo uso de unos scripts sencillos para los que se programa su ejecución periódica (`cron`), se comprueba que la aplicación esté en constante ejecución, manteniéndola disponible después de reinicios del servidor u otras circunstancias imprevistas.

Manual de usuario

La página de inicio de la aplicación, el punto de entrada para el usuario, es muy sencilla. Simplemente se muestra un mensaje de bienvenida y una pequeña explicación. (Ver figura 7.2).

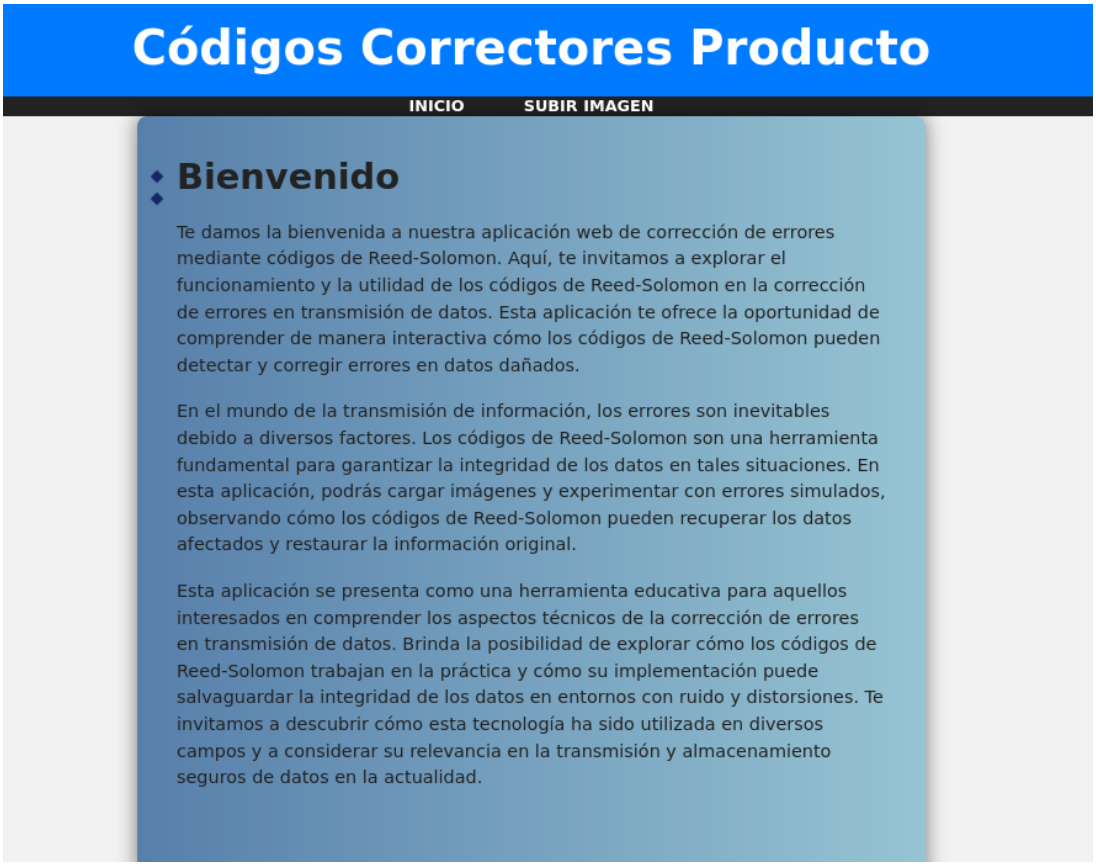


Figura 7.2: Pantalla de inicio de la aplicación

En la barra de navegación superior tenemos dos botones, INICIO y SUBIR IMAGEN. Clicar el botón de subir imagen nos lleva al proceso que comienza la funcionalidad de la aplicación web.

Se muestra la pantalla de la figura 7.4, con un deslizador para controlar el parámetro k del código.

Clicando en subir imagen se abrirá un cuadro de diálogo donde el usuaio puede navegar su sistema de archivos para localizar la imagen que quiere subir. Una vez seleccionado el botón pasará a mostrar el nombre del archivo de imagen.

En caso de no querer subir una imagen propia, se proporciona una imagen por defecto

Figura 7.3: Imagen por defecto

Para seguir con el proceso, una vez se haya seleccionado el parámetro k deseado, pulsamos el botón SUBIR Y PROCESAR. Los controles serán reemplazados por un indicador circular de progreso, señalizando que el procesado (codificadión) de la imagen seleccionada está teniendo lugar. (7.5)

Figura 7.4: Pantalla de subida de imagen de la aplicación

Figura 7.5: Pantalla de subida en espera

Una vez finaliza este proceso somos redireccionados a una pantalla en la que se nos muestra la imagen codificada (figura 7.6), con la redundadncia explícitamente visible en forma de ruido en los bordes de la imagen.

Figura 7.6: Pantalla que muestra la imagen codificada y las distintas opciones de corrupción

En esta pantalla tenemos distintas opciones. Con el ratón podemos garabatear (rayar) la imagen, pudiendo controlar el ancho de línea con el deslizador correspondiente. Otro opción es presionar el botón de ruido aleatorio, que introducirá errores de manera uniforme en la imagen con una densidad controlable mediante otro deslizador (por defecto 2 %). En ejemplo de la combinación de ambas opciones se puede observar en la figura 7.7. Si pulsamos el botón borrar podemos volver a tener la imagen corregida sin los errores introducidos.

Figura 7.7: Ejemplo de imagen corrupta en la aplicación

Para proceder al proceso de decodificación pulsaremos el botón GUARDAR, de nuevo los controles serán sustituidos por un marcador de progreso circular ya que la decodificación es un proceso computacionalmente exigente y puede tomar unos segundos. Cuando este proceso se haya completado, seremos redirigidos a la pantalla que muestra los resultados.

En esta pantalla (7.8) se muestran los resultados de la corrección de la imagen por etapas, por las que podemos navegar utilizando las flechas que aparecen encima de la imagen. También se muestran estadísticas de cada iteración de la corrección, en concreto los errores que se han podido corregir y en cuantos bloques (trozos de filas o de columna que son las unidades de procesamiento del código) se han detectado fallos que no se han podido corregir.

Figura 7.8: Pantalla que muestra los resultados de la decodificación

En la figura 7.9, se exponen todos los pasos que muestra la aplicación web para el ejemplo dado.

Figura 7.9: Pasos de la corrección de errores correspondientes al ejemplo del manual de usuario.

Contenido incluido

Se incluyen junto a esta memoria dos carpetas `cuadernos` y `webapp`, que contienen respectivamente los cuadernos de jupyter y el material necesario para ellos y el código fuente de la aplicación web.

En la carpeta `cuadernos` se incluye

- `requirements.txt` Archivo de dependencias de Python necesarias para poder ejecutar los cuadernos.
- `bms.ipynb` Cuaderno donde se implementa y explica el algoritmo de Berlekamp-Masey tratado en la sección 1 del capítulo 5.
- `sudan.ipynb` Cuaderno donde se implementa y explica el algoritmo de Sudan tratado en la sección 2 del capítulo 5.
- `comb.ipynb` Cuaderno donde se implementa y explica el algoritmo que utiliza el resultado de decodificar en lista para decodificar un producto tratado en la sección 3 del capítulo 5.
- `iter.ipynb` Cuaderno donde implementa, aplica a imagenes, y realiza pruebas del algoritmo iterativo de decodificación de códigos producto tratado en el capítulo 6. Utiliza los dos siguientes archivos.
- `paisaje.jpg` Imagen que se utiliza de ejemplo
- `final.csv`, `its.csv` Output de las últimas celdas de `iter.ipynb` (Pruebas de rendimiento)

En la carpeta `webapp` se incluye el archivo de configuración de `docker-compose` y la carpeta `app` que contiene todo el código fuente de la aplicación web así como un archivo de configuración de Docker `Dockerfile` y las dependencias de Python necesarias para la aplicación `requirements.txt`.

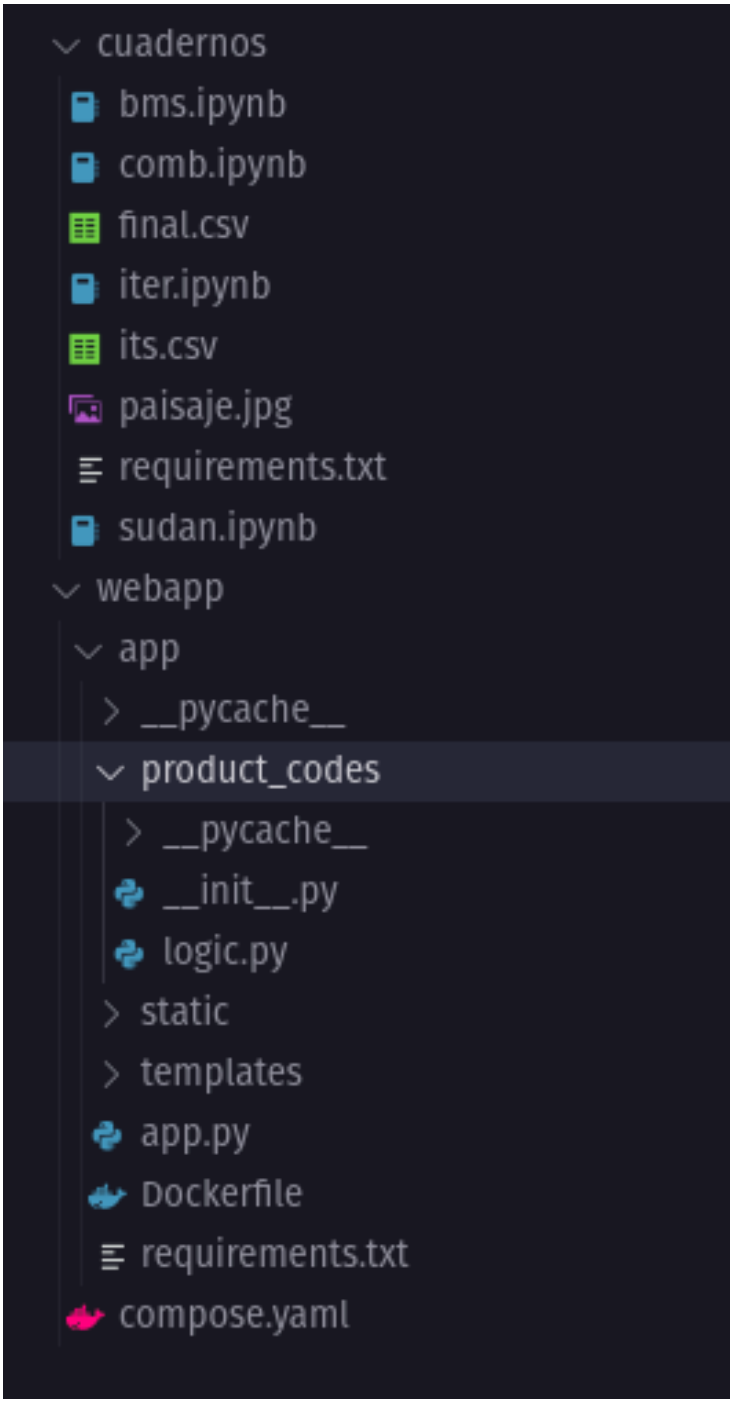


Figura 7.10: Archivos incluidos con la memoria

Bibliografía

[1] Omar Al-Askary. "Iterative decoding of product codes". Tesis doct. Signaler, sensorer och system, 2003.

[2] Richard E Blahut. *Algebraic codes for data transmission*. Cambridge university press, 2003.

[3] Pablo Ibarlucea González. "Trabajo de Fin de Grado de Matemáticas: Producto de códigos Reed-Solomon". Universidad de Valladolid, 2023.

[4] Matt Hostetter. *Galois: A performant NumPy extension for Galois fields*. Nov. de 2020. URL: `https://github.com/mhostetter/galois`.

[5] T. Høholdt. *RS en-/decoding in DVD players*. URL: `http://www2.mat.dtu.dk/people/T.Hoeholdt/DVD/index.html` (visitado 22-07-2023).

[6] Jørn Justesen y Tom Høholdt. *A Course In Error-Correcting Codes*. European Mathematical Society, 2017. ISBN: 9783037191798.

[7] Johnny Phuong Nguyen. "Applications of Reed-Solomon codes on optical media storage". Tesis doct. Citeseer, 2011.

[8] Irving S Reed y Gustave Solomon. "Polynomial codes over certain finite fields". En: *Journal of the society for industrial and applied mathematics* 8.2 (1960), págs. 300-304.

[9] *Salario medio para Programador en España, 2023*. URL: `{https://es.talent.com/salary?job=programador}`.

[10] Madhu Sudan. "Decoding of Reed Solomon codes beyond the error-correction bound". En: *Journal of complexity* 13.1 (1997), págs. 180-193.

[11] Madhu Sudan. "List decoding: Algorithms and applications". En: *ACM SIGACT News* 31.1 (2000), págs. 16-27.

[12] Wikipedia contributors. *Reed–Solomon error correction*. 2023. URL: `https://en.wikipedia.org/wiki/Reed%E2%80%93Solomon_error_correction` (visitado 04-05-2023).