Proyecto de Grado

Ingeniería en Computación

Facultad de Ingeniería, UdelaR

libJLS, una biblioteca para compresión de imágenes sin pérdida

Montevideo, Uruguay, Agosto de 2022

Alumnos:
Juan García
Andrés Kent

Supervisores:
Gadiel Seroussi
Álvaro Martín
Ignacio Ramírez

Resumen

Los algoritmos de compresión de imágenes sin pérdida han ido evolucionando con el correr de los años, obteniendo en muchos casos mejores tasas de compresión a costo de mayor complejidad computacional. El algoritmo LOCO-I, en el que se basa el formato JPEG-LS, busca encontrar un equilibrio entre una buena tasa de compresión y una baja complejidad computacional.

En este proyecto se desarrolló una biblioteca de calidad comercial en lenguaje C que implementa las funcionalidades definidas en la norma de JPEG-LS, centrándose en su versión sin pérdida de datos, sumando también alguna funcionalidad definida en la extensión del formato. En ella se ofrece una interfaz amplia y clara, buscando abarcar desde usuarios casuales a través de una versión simplificada, hasta usuarios que desean aprovechar todo lo que brinda el formato, a través de funciones específicas para cada funcionalidad.

Junto a la biblioteca también se ofrece una extensa documentación en forma de guía de usuario, con códigos de ejemplo para facilitarle su utilización, además de un generador de documentación de la API.

El resultado es una biblioteca robusta, con buena documentación y con una modularidad que facilita su ampliación a futuro. Si bien la biblioteca no alcanza los tiempos de ejecución más rápidos que se han registrado para JPEG-LS, los aproxima, habiendo elegido un balance entre la modularidad del código y su máxima optimización.

Palabras clave— JPEG-LS, LOCO-I, Compresión de imágenes sin pérdida, Biblioteca de compresión de imágenes

Índice general

Capítulo 1

Introducción

El avance de la tecnología tanto en fotografía digital como en calidad y resolución de pantallas tiene como consecuencia que las imágenes que manejan los distintos dispositivos tengan cada vez mayores dimensiones. Mayores dimensiones implican mayor cantidad de datos, por lo que el problema del espacio de almacenamiento que ocupan las imágenes tiene cada vez más importancia en la industria. Para encarar este problema, existen varios *algoritmos de compresión* que buscan reducir el espacio que ocupa una imagen. Los algoritmos de compresión *con pérdida* pueden potencialmente sacrificar calidad de imagen para reducir lo máximo posible su tamaño. Por otro lado, los algoritmos de compresión *sin pérdida* reducen el tamaño de la imagen sin comprometer la calidad, únicamente a partir de la reducción de información redundante en su codificación.

A lo largo de los años se han desarrollado múltiples algoritmos de compresión de imágenes, que han ido mejorando las tasas de compresión y soportando nuevas funcionalidades. Pero la tasa de compresión, por más que sea uno de los principales atributos que hacen a un buen compresor, no es el único factor a tener en cuenta. Otro aspecto fundamental a la hora de evaluar el desempeño de un algoritmo de compresión es su *complejidad computacional*, es decir, los recursos requeridos por el algoritmo para completar su ejecución. La complejidad computacional de un algoritmo se ve reflejada en el tiempo de compresión/descompresión: a mayor complejidad, más tiempo una imagen tarda en ser procesada. Esto, como es de esperar, puede resultar un problema principalmente en aplicaciones que manejen imágenes de grandes dimensiones, lo cual es cada vez más común.

Con este problema en mente se diseñó LOCO-I (LOw COmplexity LOssless COmpression for Images) [1], un algoritmo de compresión de imágenes sin pérdida que busca mantener una buena tasa de compresión, comparable con las otras alternativas disponibles, pero a una menor complejidad computacional. Alrededor de este algoritmo se diseñó y, a través de la norma ISO/IEC 14495-1 [2], se estandarizó JPEG-LS, un formato para imágenes sin pérdida que soporta la mayor parte de funcionalidades de los algoritmos modernos, como ser muestras de hasta 16 bits,

tablas de mapeo, entre otras, sumando las ventajas que ofrece la baja complejidad computacional de LOCO-I. Posteriormente se estandarizó la extensión del formato a través de la norma ISO/IEC 14495-2 [3], donde se añaden algunas funcionalidades adicionales como transformadas de color y codificación aritmética.

Los algoritmos más populares de compresión de imágenes sin pérdida en la actualidad son PNG [4] y GIF [1], sumado al reciente surgimiento de WebP [2].

PNG ofrece una gran variedad de funcionalidades, tiempos de ejecución razonables y una buena tasa de compresión. Por otro lado, GIF soporta únicamente imágenes de hasta 256 colores, basándose en el concepto de *paletas de colores*, que consiste en definir una lista de colores y limitar los posibles valores de los píxeles a los colores de la lista. Esta técnica le permite lograr muy buenos niveles de compresión, aunque en caso de comprimir una imagen de más de 256 colores, el resultado de la compresión en formato GIF contiene pérdida, ya que algunos colores se deben aproximar a su valor más cercano en la paleta. Es especialmente popular su uso en forma de secuencia de imágenes, funcionalidad en la que destaca. WebP fue desarrollado por Google hace poco más de una década con el objetivo de convertirse en el nuevo estándar, ofreciendo compresión con y sin pérdida y una gran variedad de funcionalidades.

Típicamente los formatos de compresión de imágenes ofrecen implementaciones distribuidas en formato de biblioteca, para que los usuarios desarrolladores puedan incorporarlas a sus aplicaciones y así soportar imágenes en ese formato. Sin embargo, luego de publicada la norma de JPEG-LS, nunca se publicó una implementación oficial del formato. Existen implementaciones como la desarrollada por la universidad de British Columbia y otras que se pueden encontrar en la web, pero en todos los casos están incompletas o no cumplen con todos los aspectos de la norma. Sin duda que la más valiosa es la desarrollada por los propios responsables del diseño del algoritmo LOCO-I, implementada durante el proceso de diseño del algoritmo para evaluar su funcionamiento.

El objetivo de este proyecto es entonces diseñar, desarrollar y publicar una biblioteca en lenguaje C que implemente todo lo que ofrece el formato JPEG-LS. Las principales características que debe cumplir la solución desarrollada son:

- Interfaz práctica de uso sencillo y eficiente, aplicable en diversos escenarios.
- Implementación correcta, eficiente y robusta, en lenguaje C.
- Compatibilidad con plataformas Linux, Windows y Mac.

[1] GRAPHICS INTERCHANGE FORMAT(sm). CompuServe Incorporated. Ver. 87. Especificación disponible en: `https://www.w3.org/Graphics/GIF/spec-gif87.txt`

[2] "An image format for the Web". Documentación disponible en: `https://developers.google.com/speed/webp`

- Diseño modular que permita modificaciones y extensiones.
- Implementación de las funcionalidades más importantes que ofrecen JPEG-LS y su extensión.
- Documentación de usuario de la biblioteca y del diseño de la solución.

Para validar su correcto funcionamiento y robustez, la biblioteca debe ser sometida a pruebas exhaustivas que abarquen una amplia variedad de situaciones. La implementación desarrollada por los responsables del diseño de LOCO-I fue la utilizada como referencia para evaluar la adecuación a la norma de la versión desarrollada en este proyecto y como punto de comparación para evaluar su rendimiento.

Implementar una biblioteca desde cero implica un diseño previo de la estructura del código, que debe ser suficientemente adaptable a la incorporación de nuevas funcionalidades, manteniendo un orden y una claridad que facilite construir a partir de la base inicial implementada. De hecho, uno de los principales desafíos es lograr un balance entre el rendimiento, modularidad y claridad del código. Por ejemplo, la utilización de *macros* de C puede favorecer el rendimiento en algunos casos, a cambio de afectar la fácil comprensión del código, por lo que deben ser usados con criterio.

Por otro lado, la interfaz ofrecida debe seguir la línea de lo que proponen otras bibliotecas similares, para así facilitar su adopción para usuarios desarrolladores que estén familiarizados con el uso de estas, y simplificar la incorporación del soporte de JPEG-LS a aplicaciones ya existentes. En este sentido buscamos mejorar sobre lo que ofrecen dichas bibliotecas, notar los aspectos en los que flaquean y diseñar la interfaz de forma de fortalecer dichos aspectos.

Como resultado se obtuvo una biblioteca de calidad comercial, que cumple con la amplia mayoría de los objetivos planteados. Con el foco en brindar una buena experiencia al usuario, se implementó una interfaz simplificada para el uso típico de la biblioteca, acompañada de una interfaz más completa que ofrece las herramientas necesarias para tener el control total del procesamiento de las imágenes. Se desarrolló una extensa documentación que comprende una guía de usuario, que profundiza en todas las funcionalidades que ofrece la biblioteca y detalla su forma de uso acompañada de códigos de ejemplo para su mejor entendimiento, sumado a una documentación completa del código generada con la herramienta Doxygen[3]. Uno de los principales aspectos que admiten mejoras es el rendimiento. El tiempo

[3]Herramienta que permite la generación de documentación en forma de sitio web (en formato HTML), PDF u otros formatos a partir de la documentación del código fuente. Para más información: `https://doxygen.nl`

de codificación es de un 65 % superior al de la implementación de referencia y un 95 % superior para la decodificación. Aún así, los tiempos de codificación logrados por la biblioteca llegan a ser 2 veces más rápidos que los de PNG, y 6 veces más rápidos que los de WebP, por lo que en este sentido la implementación le saca ventaja a sus alternativas. Consideramos que el sacrificio en rendimiento es el precio que se pagó por una mayor modularidad y claridad del código, comparado con la implementación de referencia, que había sido optimizada casi exclusivamente para la reducción de tiempos de ejecución.

En el presente informe, en primera instancia, presentamos conceptos relacionados a la compresión de imágenes sin pérdida a modo de marco teórico en el capítulo 2. También mencionamos algunos de los formatos de compresión de imágenes más populares en la actualidad. En el capítulo 3 profundizamos en el diseño del algoritmo LOCO-I y las características del formato JPEG-LS. A continuación, en el capítulo 4 detallamos varios aspectos vinculados con la implementación de la biblioteca, haciendo énfasis en la arquitectura, el flujo interno de operaciones y el diseño de la *interfaz de programación de aplicaciones* (*application programming interface*, o API). Luego presentamos los resultados de la evaluación de la biblioteca en el capítulo 5, aclarando los aspectos considerados a la hora de la evaluación y en qué consistieron las pruebas realizadas. Para cerrar el informe, en los capítulos 6 y 7 mencionamos los trabajos futuros que se pueden realizar a partir del resultado obtenido y conclusiones del proyecto. En los anexos 8 incluimos el manual de usuario para la biblioteca desarrollada, además de los resultados completos de las pruebas realizadas.

Capítulo 2

Marco teórico

El objetivo de este capítulo es presentar algunos conceptos básicos necesarios para comprender el contenido del documento. Explicamos el concepto de compresión y descompresión de imágenes, técnicas comúnmente utilizadas y mencionamos algunos de los formatos más populares de compresión de imágenes en la actualidad.

A lo largo de todo el documento, las palabras compresor y codificador (o descompresor y decodificador) se utilizan como sinónimos.

2.1. Compresión de imágenes

Una imagen es representada como una matriz de píxeles, donde cada píxel tiene *componentes* (o *canales*), asociados a un espacio de color. El espacio de color más utilizado es RGB, teniendo cada píxel tres componentes en este caso (rojo, verde y azul). La figura 2.1 ilustra esta representación. El color del píxel es formado por la combinación de todos sus componentes. En algunos casos, el espacio de color incluye un canal adicional especial llamado *transparencia* o *alpha*. Este canal es utilizado para que, al superponer la imagen sobre un fondo, este sea visible en cierto grado a través de la imagen. El valor del canal alpha indica entonces cuánto se transparenta ese píxel sobre el fondo.

Cada componente de un píxel es representado con un número que indica su intensidad. El número de bits utilizados para representar un componente, denominado *profundidad*, define su rango de valores posibles. Típicamente cada componente es representado con ocho bits, por lo que puede variar en el rango [0, 255], aunque en algunos casos se llegan a utilizar hasta 2 bytes para cada componente, agrandando ese rango a [0, 65535].

Codificar una imagen implica codificar los componentes de todos sus píxeles. Para realizar esta tarea, los algoritmos compresores escanean los píxeles por líneas y codifican los valores de los componentes buscando reducir al máximo la *redundancia* en la codificación. Esta redundancia proviene, por ejemplo, de la similitud

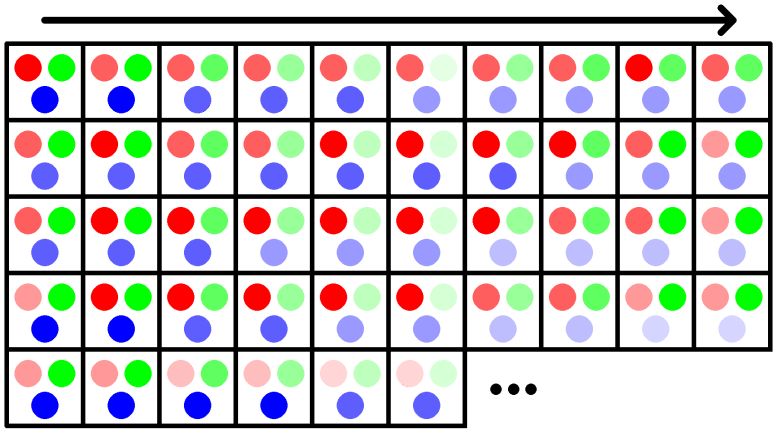

Figura 2.1: Representación de una imagen en RGB. Notar la variación de intensidades en los componentes, que permite formar píxeles de distintos colores. La flecha indica el sentido en el que se escanea la imagen, comenzando por el píxel superior izquierdo.

que típicamente se observa entre píxeles vecinos, lo cual hace que algunos de ellos sean predecibles en función de los demás (profundizaremos en ello en la sección 3.1) Algunos algoritmos incluso realizan múltiples escaneos para recolectar información que les permita reducir aún más la redundancia, a costa de un aumento de la complejidad computacional del algoritmo.

La capacidad de compresión de un algoritmo se ve reflejada en su *tasa de compresión*. Ésta se puede definir como:

$$(tasa\ de\ compresión) = \frac{(\#\ de\ bits\ en\ imagen\ comprimida)}{(\#\ de\ pixeles\ en\ imagen\ original)}$$

Al comprimir una imagen se busca que este valor, que en el presente informe será representado en *bits/pixel*, sea lo más bajo posible.

Existen dos tipos de compresión de imágenes: *sin pérdida* (*lossless*) o *con pérdida* (*lossy*).

2.1.1. Compresión sin pérdida

La *compresión sin pérdida* tiene como principal objetivo que la imagen reconstruida por el descompresor coincida exactamente con la original. Para esto, se encarga de reducir el espacio de la imagen únicamente descartando información redundante. Habitualmente las tasas de compresión para este caso son bastante peores que las que se pueden obtener mediante compresión con pérdida (descrita

en la siguiente subsección). De todas formas, la fidelidad total que ofrece es una ventaja que en muchos casos resulta imprescindible.

Generalmente esta técnica es utilizada para imágenes detalladas de productos, fotografías de buena calidad, imágenes cuya adquisición tiene un costo elevado, imágenes médicas, u otras situaciones donde la pérdida de calidad puede resultar un problema para el usuario. Para este tipo de compresión, algunos de los formatos más populares son PNG y GIF. Este es el tipo de compresión que realiza JPEG-LS.

2.1.2. Compresión con pérdida

La *compresión con pérdida* consiste en la aplicación de una transformación a una imagen (en general con cierta pérdida de detalle), seguida de una compresión sin pérdida de dicha imagen transformada. Al reconstruir una imagen que fue comprimida con pérdida, la imagen descomprimida no necesariamente coincide en forma exacta con la original, pero tiene la gran ventaja de que permite obtener tasas de compresión mucho mejores. Por otro lado, su principal inconveniente es que la calidad de la imagen se puede ver afectada hasta niveles inaceptables para ciertas aplicaciones.

Las imágenes comprimidas con pérdida suelen ser utilizadas en situaciones donde la prioridad es minimizar el tamaño de la imagen, siendo cierta reducción de calidad algo aceptable ya que seguramente no sea notorio para la mayoría de usuarios. Es muy común su utilización en sitios web, donde se prioriza que el sitio cargue rápidamente, siendo el tamaño de las imágenes que lo componen un factor crítico en la velocidad de carga del sitio. Por lo general estos algoritmos ofrecen distintos niveles de compresión, dependiendo de cuánta fidelidad exige el usuario para la imagen reconstruida. El formato más popular de compresión con pérdida es JPEG [5].

2.2. Técnicas de compresión sin pérdida

2.2.1. Códigos

Un *código* para una variable aleatoria X es un mapeo desde $\mathcal{X}$, el rango de X, a $\mathcal{D}^*$, el conjunto de cadenas de largo finito de símbolos de un alfabeto binario $\mathcal{D} = \{0, 1\}$ [4]. *C(x)* denota la *palabra de código* correspondiente a x [6]. Por ejemplo, *C(rojo)* = 00, *C(verde)* = 01, *C(azul)* = 10 es un posible código para $\mathcal{X} = \{rojo, verde, azul\}$. A no ser que se especifique lo contrario, supondremos a continuación que $\mathcal{X}$ es siempre un conjunto finito.

Para codificar un mensaje formado por una cadena de símbolos de $\mathcal{X}$, las palabras de código son concatenadas en una única cadena de bits, que luego es

[4] El caso binario es el más común, pero $\mathcal{D}$ puede ser cualquier conjunto finito.

almacenada o transmitida. Una *extensión* C^* de un código C es un mapeo de cadenas de largo finito de $\mathcal{X}$ a cadenas de largo finito de $\mathcal{D}$, definida por

$$C(x_1x_2\cdots x_n) = C(x_1)C(x_2)\cdots C(x_n)$$

donde $C(x_1)C(x_2)\cdots C(x_n)$ indica la concatenación de las palabras de código correspondientes.

Para poder interpretar el mensaje se debe realizar el proceso inverso, leyendo las palabras de código desde la cadena de bits y asociándolas a símbolos de $\mathcal{X}$.

Un código se denomina *unívocamente decodificable* si su extensión es inyectiva, es decir, si toda palabra de código de C^* tiene una única cadena de símbolos de $\mathcal{X}$ que la produce. Por ejemplo, para el caso de *C(rojo)* = 1, *C(verde)* = 0, *C(azul)* = 10 al leer la cadena "10" no se sabe si se están leyendo las palabras "1" y "0" (rojo-verde) o la palabra "10" (azul). Por lo tanto, este código no es unívocamente decodificable. El uso de códigos unívocamente decodificables permite que no exista ambigüedad a la hora de decodificar el mensaje.

2.2.2. Código prefijo

Un *código prefijo* es un código con la propiedad de que ninguna palabra de código es prefijo de ninguna otra palabra de código. Por ejemplo, un código con las palabras {1, 01, 00} es un código prefijo, pero {0, 1, 10} no lo es ya que "1" es prefijo de "10".

Los códigos prefijo son un caso particular de códigos unívocamente decodificables, muy simples de decodificar. El receptor puede decodificar fácilmente el mensaje detectando las palabras de código $C(x_1), C(x_2), \cdots, C(x_n)$ una a una, secuencialmente, decodificando cada símbolo $x_1, x_2, \cdots, x_n$ individualmente.

2.2.3. Codificación de Huffman

La *codificación de Huffman* es una técnica desarrollada por D. A. Huffman en el año 1952 [7] con el objetivo de generar códigos prefijos de longitud variable que permitan minimizar el largo de código esperado. Para esto, se parte de una distribución de probabilidad conocida para los símbolos a codificar. A partir de esta distribución, se busca que los símbolos con mayor probabilidad de ocurrencia se representen utilizando una menor cantidad de bits, mientras que los símbolos menos comunes se representen utilizando más bits.

Mediante la codificación de Huffman se pueden obtener los códigos prefijos óptimos para símbolos con una distribución de probabilidad conocida. Debido a su simplicidad y al no ser una técnica patentada, los códigos de Huffman siguen siendo ampliamente utilizados, siendo en muchos casos usados como componentes

de otros métodos de compresión como "Deflate", el algoritmo utilizado para el formato PNG.

2.2.4. Codificación aritmética

La *codificación aritmética* [8], al igual que la codificación de Huffman, parte de una distribución de probabilidad de ocurrencia sobre los símbolos, asignando a aquellos con mayor probabilidad de ocurrencia palabras de menor longitud, y a los de menor ocurrencia, palabras de mayor longitud. A diferencia de la codificación de Huffman que separa el mensaje en símbolos y asocia a cada símbolo una palabra de código, la codificación aritmética codifica un mensaje entero mediante una cadena de bits que corresponde a la parte fraccionaria de un número q entre 0 y 1.

El algoritmo de codificación ajusta iterativamente un intervalo en el que se encuentra q, que inicialmente es [0, 1). Al codificar cada símbolo del mensaje, este intervalo se ajusta en función de la probabilidad de ocurrencia de dicho símbolo. Una vez que se procesaron todos los símbolos, el intervalo que queda definido identifica el mensaje codificado. No es necesario representar el intervalo resultante en su totalidad, sino que simplemente alcanza con almacenar un número q dentro de ese intervalo con precisión suficiente para distinguir entre sí los intervalos correspondientes a diferentes mensajes.

Supongamos que se quiere codificar un mensaje con los símbolos {C, L, O}, donde se sabe que la probabilidad de ocurrencia de cada símbolo es {0.1, 0.3, 0.6} respectivamente. Se quiere codificar el mensaje "LOCO". El intervalo comienza siendo [0, 1). En base a las probabilidades de ocurrencia de cada símbolo, se definen los intervalos [0, 0.1) asociado al símbolo "C", [0.1, 0.4) al símbolo "L" y [0.4, 1) al símbolo "O". El primer símbolo a codificar es "L", por lo que el intervalo se reduce a [0.1, 0.4). Para el siguiente paso, dentro de [0.1, 0.4) se definen otros 3 intervalos, de anchos proporcionales a las probabilidades de ocurrencia de cada símbolo: [0.1, 0.13), [0.13, 0.22), [0.22, 0.4). El siguiente símbolo a codificar es "O", por lo que el intervalo elegido es [0.22, 0.4). Ahora, los nuevos intervalos son [0.22, 0.238), [0.238, 0.292), [0.292, 0.4), y como el símbolo a codificar es "C", el intervalo elegido es [0.22, 0.238). Por último, se definen los intervalos [0.22, 0.2218), [0.2218, 0.2272), [0.2272, 0.238), y al ser "O" el símbolo a codificar, el intervalo pasa a ser [0.2272, 0.238). En binario, el intervalo final sería entonces [0.001110100010100111, 0.0011110011011011001), por lo que se puede elegir el valor 0.001111 dentro de ese intervalo, es decir, 001111 si ignoramos el dígito a la izquierda que siempre es cero. De esta forma solo fueron necesarios 6 bits para codificar el mensaje "LOCO".

Vale la pena notar que la decodificación es extremadamente simple. Para este caso alcanzaría con tomar el valor 0.001111, cuya representación decimal es 0.234375, y realizar el proceso inverso. El valor 0.234375 corresponde al intervalo

[0.1, 0.4), por lo que el primer símbolo decodificado es “L”. También se encuentra en el intervalo [0.22, 0.4) por lo que el segundo símbolo es “O”. El siguiente intervalo es [0.22, 0.238), por lo que el tercer símbolo es “C”. Mientras que en el último caso, el intervalo donde se encuentra 0.234375 es [0.2272, 0.238), por lo que el último símbolo a decodificar es “O”.

Cuando es diseñada cuidadosamente, la codificación aritmética puede obtener un largo de código óptimo en un sentido matemático bien definido [6].

2.2.5. Codificación de Golomb

La *codificación de Golomb* es otro tipo de codificación utilizada para compresión sin pérdida, desarrollada por S. W. Golomb en los años 60 [9]. Un código de Golomb es óptimo para codificar enteros no negativos aleatorios que siguen una distribución de probabilidad geométrica. Esto hace que la codificación de Golomb sea particularmente efectiva cuando la frecuencia de ocurrencia de valores pequeños es significativamente mayor que la de valores grandes.

Esta codificación consiste en definir un parámetro de Golomb M, y para un valor N que se quiera codificar se codifica primero el cociente $q = \lfloor N/M \rfloor$ y luego el resto $r = N \mod M$. Para codificar el cociente q se utiliza una codificación unaria inversa, es decir, si se quiere codificar el valor 5 se escribe una cadena de 5 bits en 0, seguido de un bit en 1 (000001). El resto r se codifica usando una codificación binaria truncada simple, que consiste en definir un valor $b = \lfloor \log_2 M \rfloor$ y si $r < 2^{b+1} - M$ se codifica r en su representación binaria usando b bits, mientras que en caso contrario se codifica el valor $r + 2^{b+1} - M$ usando $b + 1$ bits. Para el caso de $M = 2^k$ la codificación resulta particularmente simple y admite una implementación muy eficiente.

2.3. Formatos de compresión de imágenes

Existen múltiples formatos de compresión de imágenes. Cada formato ofrece distintas funcionalidades, tasas de compresión, velocidad de compresión y otros aspectos que los diferencian del resto. A continuación se presentan los más populares.

2.3.1. JPEG

JPEG (Joint Photographic Experts Group) [5] es uno de los formatos más utilizados para fotografías. Utiliza un algoritmo de compresión con pérdida, ofreciendo distintos grados de compresión. Un grado de compresión mayor generará un archivo más pequeño pero en general de menor calidad, mientras que un grado bajo de compresión obtiene una imagen muy similar a la original a cambio de un

archivo de mayor tamaño. Los archivos en formato JPEG suelen tener las extensiones .jpeg o .jpg . JPEG-LS es un formato de la familia de normas JPEG que ofrece compresión sin pérdida.

2.3.2. PNG

PNG (Portable Network Graphics) [4] es uno de los formatos de compresión de imágenes sin pérdida más populares. Fue concebido como un formato genérico para imágenes, obteniendo buenos resultados tanto para imágenes naturales como artificiales. El algoritmo en el que se basa no está sujeto a patentes, a diferencia del de JPEG cuando se definió. Entre otras diferencias con JPEG, una de las más notorias para los usuarios es que PNG soporta transparencia, representada como un componente adicional que admite distintos niveles de intensidad, además de algunos atributos extra como la corrección gamma. Sin embargo no es compatible con el espacio de color CMYK (Cyan Magenta Yellow blacK) utilizado por las impresoras, por lo que no es de mucha utilidad para diseños de impresión.

El método de compresión que utiliza PNG se conoce como deflación ("Deflate algorithm"). Este método usa una combinación del algoritmo LZ77 [10] con codificación de Huffman.

2.3.3. GIF

GIF (Graphics Interchange Format) es un formato de compresión sin pérdida para imágenes de hasta 256 colores, definidos en una paleta de colores. Las imágenes en este formato son comprimidas usando el algoritmo LZW [11]. A diferencia de PNG que permite píxeles con distintos niveles de transparencia, GIF solo soporta transparencia total, es decir, únicamente admite píxeles totalmente transparentes o totalmente visibles.

Más allá de que es un formato de compresión sin pérdida, esto solo aplica a imágenes de 256 colores o menos. Para imágenes de más de 256 colores, el algoritmo adapta la imagen reduciendo el número de colores, teniendo por consecuencia una reducción de calidad. En estos casos el algoritmo define una paleta de 256 colores inteligentemente, de forma de obtener una imagen lo más fiel posible a la original. Esta reducción del número de colores es la clave para que GIF logre una buena compresión.

GIF es utilizado tanto para imágenes como para animaciones. Para este último caso, el formato soporta la definición de un conjunto de frames (imágenes) que pueden compartir la paleta de colores o no.

2.3.4. WebP

WebP es un formato que soporta tanto compresión con pérdida como sin ella, además de soportar tanto animaciones como transparencia. Lo está desarrollando

Google con el objetivo de reemplazar los formatos JPEG, PNG y GIF, habiendo tenido el lanzamiento de su versión inicial en el año 2010.

Capítulo 3

Algoritmo LOCO-I y formato JPEG-LS

En este capítulo detallamos el funcionamiento del algoritmo LOCO-I, utilizado como base para el formato JPEG-LS, para luego profundizar en las características del formato.

3.1. LOCO-I

El desarrollo de LOCO-I [1] [12] surge del objetivo de diseñar un algoritmo de compresión sin pérdida para imágenes de tono continuo que ofrezca tasas de compresión muy buenas, comparables a las mejores ofrecidas por los algoritmos contemporáneos basados en codificación aritmética, pero reduciendo en gran medida su complejidad computacional. Para lograr esto, LOCO-I utiliza un modelo que combina la simplicidad de la codificación de Golomb-Rice con el potencial de compresión de los modelos de contexto.

Se realiza una recorrida de los píxeles de la imagen a codificar, avanzando por líneas horizontales de arriba hacia abajo, recorriendo cada línea de izquierda a derecha. Al procesar la muestra $t+1$ se tiene información estadística acumulada del escaneo de las t muestras anteriores, denotadas x^t. [5] En base a esta información, se estima una cierta distribución de probabilidad condicional $P(\cdot|x^t)$ para la siguiente muestra x_{t+1}. Este esquema de asignación de probabilidades se puede dividir en los siguientes componentes:

1. Una *predicción*, donde para la siguiente muestra a procesar, x_{t+1}, se estima el valor $\hat{x}_{t+1}$ en base a las muestras anteriores x^t.

[5] Esto se debe a que el algoritmo LOCO-I es de una sola pasada, es decir, se recorre la imagen secuencialmente y no se tiene información de las siguientes muestras hasta no ser procesadas.

2. La determinación de un *contexto* en el que ocurre la muestra x_{t+1}, calculado en función de muestras vecinas a x_{t+1} en x^t.

3. Un modelo probabilístico para el *error en la predicción* $\epsilon = x_{t+1} - \hat{x}_{t+1}$, condicionado en el contexto de x_{t+1}.

Las unidades de predicción y modelado en LOCO-I se basan en el esquema ilustrado en la figura 3.1. Siendo x el píxel procesado actualmente, a los píxeles *a, b, c, d* que lo rodean se los denomina *vecinos*. Es importante notar que al procesar el pixel *x*, los vecinos *a, b, c, d* ya fueron procesados, por lo que tanto el codificador como el decodificador conocen los valores de estos píxeles.

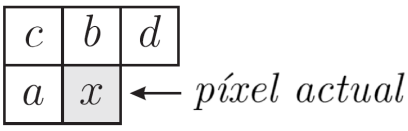

Figura 3.1: Esquema de vecinos.

La predicción consiste en un componente fijo y un componente adaptativo.

Componente fijo

Para el cálculo del componente fijo el algoritmo realiza una prueba primitiva para detectar bordes verticales u horizontales. Si ningún borde fue detectado entonces la predicción inicial es $a + b - c$, ya que ese sería el valor de x_{t+1} si el píxel actual perteneciera al "plano" definido por los tres píxeles vecinos con "alturas" *a*, *b* y *c*. Esto expresa la suavidad esperada de la imagen en ausencia de bordes. Específicamente, el predictor de LOCO-I realiza la siguiente predicción:

$$\hat{x}_{t+1} = \begin{cases} \text{mín}\,(a,b), & \text{si } c \geq \text{máx}\,(a,b) \\ \text{máx}\,(a,b), & \text{si } c \leq \text{mín}\,(a,b) \\ a + b - c, & \text{en caso contrario.} \end{cases}$$

Componente adaptativo

Para el componente adaptativo del predictor se tiene en cuenta el contexto en el que ocurre la muestra a procesar. Este contexto también es determinado en base a los vecinos de la muestra. En primera instancia se computan los gradientes locales $g_1 = d - a$, $g_2 = a - c$ y $g_3 = c - b$, que capturan el nivel de actividad (suavidad,

presencia de bordes) que rodea al píxel. Cada gradiente es luego cuantizado a un valor pequeño en base a regiones aproximadamente equiprobables. Para preservar simetría, las regiones son indexadas $-T, \cdots, -1, 0, 1, \cdots, T$, para un total de $2T+1$ regiones por gradiente, es decir, $(2T+1)^3$ contextos distintos. Asumiendo que

$$Prob\{\epsilon_{t+1} = \Delta | C_t = [q_1, q_2, q_3]\} = Prob\{\epsilon_{t+1} = -\Delta | C_t = [-q_1, -q_2, -q_3]\}$$

donde C_t representa un contexto determinado por la tripleta de gradientes cuantizados $q_j, j = 1, 2, 3$, se puede reducir la cantidad de contextos a aproximadamente la mitad invirtiendo el signo de los gradientes y de ϵ_{t+1} si el primer elemento de C_t distinto de cero es negativo A través de la combinación de contextos de signos opuestos, el número de contextos posibles se reduce a $((2T+1)^3+1)/2$.

Para JPEG-LS se utiliza $T = 4$, obteniendo un total de 365 contextos, y las regiones de cuantización definidas por defecto para un alfabeto de 8 bits son $\{0\}$, $\{1, 2\}$, $\{3, 4, 5, 6\}$, $\{7, 8, \cdots, 20\}$, $\{e | e \geq 21\}$ y sus correspondientes contrapartes negativas. Se definen de esta forma los límites de regiones de cuantización como $T_1 = 3, T_2 = 7, T_3 = 21$.

Una vez determinado este contexto, se tienen en cuenta los resultados obtenidos al procesar muestras anteriores que se encontraron en el mismo contexto. Para cada contexto se lleva la cuenta N de cuántas veces ocurrió y se acumula en una variable B los errores de predicción codificados hasta el momento para ese contexto. En función de estas estadísticas se define un *valor de corrección*, C, que es sumado a la predicción inicial con el objetivo de ajustar un eventual sesgo sistemático para este contexto.

Codificación

Se conoce que la distribución empírica de los errores de predicción en imágenes de tono continuo se aproxima en general a una distribución geométrica de dos caras (TSGD - Two-Sided Geometric Distribution). Esta distribución se define como

$$Prob_{(\theta,s)}(\epsilon) = C(\theta, s)\theta^{\epsilon+s}, \epsilon = 0, \pm 1, \pm 2, \pm 3 \cdots,$$

donde $C(\theta, s) = (1-\theta)/\theta^{(1-s)} + \theta^s$ es un factor de normalización, y $0 < \theta < 1$ y $0 \leq s < 1$. θ representa la tasa de decaimiento exponencial, mientras que s representa la desviación del centro de la distribución respecto del cero, como se puede apreciar en la figura 3.2.

A la hora de codificar estos errores, la elección de este modelo es de suma importancia si se busca lograr una baja complejidad computacional. Esto se debe a que la codificación se puede realizar eficientemente mediante el uso de una familia extendida de códigos de Golomb, que son seleccionados de forma adaptativa. Como se comentó en la sección 2.2.5, la codificación de Golomb consiste en definir un parámetro de Golomb M, y para un valor N que se quiera codificar se codifica

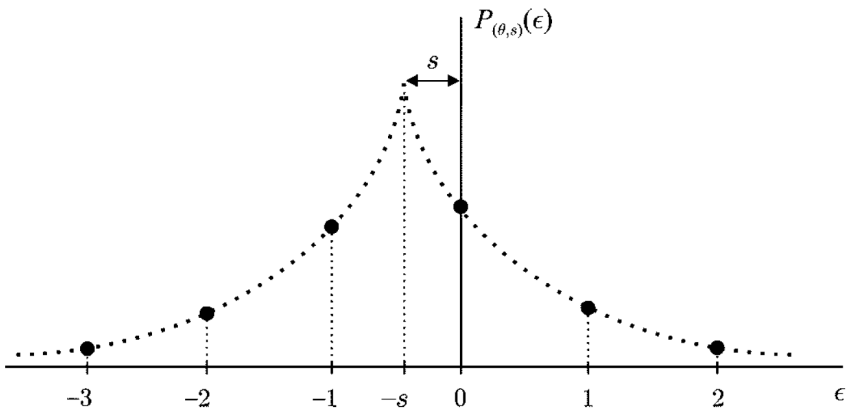

Figura 3.2: Distribución geométrica de dos caras. Figura extraída de [12].

primero el cociente $q = \lfloor N/M \rfloor$ y luego el resto $r = N \mod M$. El caso especial de códigos de Golomb con $M = 2^k$ simplifica considerablemente los procedimientos de codificación y decodificación. En este caso, la división entre M para el cálculo de q y r se puede realizar mediante la operación *shift* [6], lo cual reduce considerablemente la complejidad computacional respecto a realizar una división de enteros. Para este caso, el largo de la codificación de un entero y es $k + 1 + \lfloor y/2^k \rfloor$.

Un paso crucial en el esquema es la determinación del valor k que minimice el largo esperado del código en función de los parámetros de la TSGD. Se sabe que una buena estimación del valor óptimo de k es

$$k = \lceil \log_2 E[|\epsilon|] \rceil,$$

donde $E[|\epsilon|]$ es el promedio aritmético de la magnitud del error de predicción para el contexto actual. Para calcular este promedio, LOCO-I utiliza una variable A que acumula la suma de magnitudes de errores de predicción obtenidos en este contexto hasta el momento. De esta forma, $E[|\epsilon|]$ es calculado como A/N (siendo N el número de ocurrencias del contexto), y k se computa como

$$k = \text{mín}\,\{k' | 2^{k'} N \geq A\}$$

Para mejorar la adaptación a variaciones locales en la imagen y estadísticas no estacionarias en general, periódicamente las variables N y A son reajustadas. En

[6]Operación a nivel de bits que consiste en desplazar una cadena de bits hacia la izquierda o hacia la derecha. Aplicar esta operación sobre la representación binaria de un entero no negativo x se traduce en multiplicar $x * 2^k$ si el desplazamiento es a la izquierda o dividir $x/2^k$ si el desplazamiento es hacia la derecha, siendo k el número de bits del desplazamiento. Por ejemplo, para la representación binaria del número 12 utilizando un byte (00001100), aplicar la operación *shift* hacia la izquierda un número de bits $k = 3$ resulta en la cadena 01100000, que equivale a $12 * 2^3 = 96$.

particular, se divide a la mitad el valor de N y A cada vez que N alcanza un determinado límite N_0 predefinido. Por defecto, en JPEG-LS se utiliza $N_0 = 64$.

Es importante recordar que la codificación de Golomb permite únicamente codificar enteros *no negativos*. Inicialmente, el error ϵ en la predicción se encuentra en el rango $[-(\alpha - 1), \alpha - 1]$, siendo α el tamaño del alfabeto, aunque se observa que puede reducirse el error módulo α, llevándolo al rango $[-\alpha/2, \alpha/2 - 1]$, sin alterar el valor reconstruido por el descompresor. Debido a que el error ϵ puede tomar valores negativos LOCO-I realiza un mapeo $M(\epsilon)$ sobre el error, previo a su codificación. De esta forma se logra obtener $M(\epsilon)$ en el rango $[0, \alpha - 1]$. El mapeo realizado se define como:

$$M(\epsilon) = \begin{cases} 2\epsilon, & \epsilon \geq 0, \\ 2|\epsilon| - 1, & \epsilon < 0. \end{cases}$$

Este mapeo lleva la secuencia $0, -1, 1, -2, 2, \cdots$ de valores positivos y negativos intercalados a la secuencia de naturales $0, 1, 2, 3, \cdots$. Utilizando este mapeo la distribución de probabilidad de $M(\epsilon)$ resulta ser en general muy cercana a la geométrica, por lo que puede ser codificada eficientemente usando un código de Golomb apropiado.

Modo *run*

Para optimizar la codificación de áreas lisas en una imagen, LOCO-I define un modo *run*. El algoritmo entra en modo *run* si todos los vecinos *a, b, c, d* tienen el mismo valor, es decir, el píxel procesado se encuentra en un contexto de región plana. Al entrar en este modo se cuenta la cantidad de píxeles iguales a a desde el píxel x inclusive en adelante, hasta que la corrida es interrumpida por un valor $x \neq a$. El largo de la corrida (cantidad de muestras consecutivas con el mismo valor) se codifica mediante el uso de *block-MELCODE* [13], una técnica basada en variantes de códigos de Golomb. Durante este modo el contexto no es computado. Una vez la corrida es interrumpida, el codificador entra en un estado de "interrupción de *run*", donde la diferencia $\epsilon = x - b$ es codificada. El modo *run* también puede ser interrumpido por el final de una línea. Debido a que todas las decisiones para entrar y salir del modo *run* se basan en los píxeles anteriores, el decodificador puede reproducir estas decisiones sin la necesidad de información adicional. Por lo tanto, el decodificador sabe entrar al modo *run*, identificar y procesar la codificación del largo del *run* y de la muestra de interrupción, y salir del modo.

La figura 3.3 ilustra el flujo completo del procesamiento de una muestra de la imagen, y permite apreciar la conexión entre cada uno de los pasos detallados anteriormente.

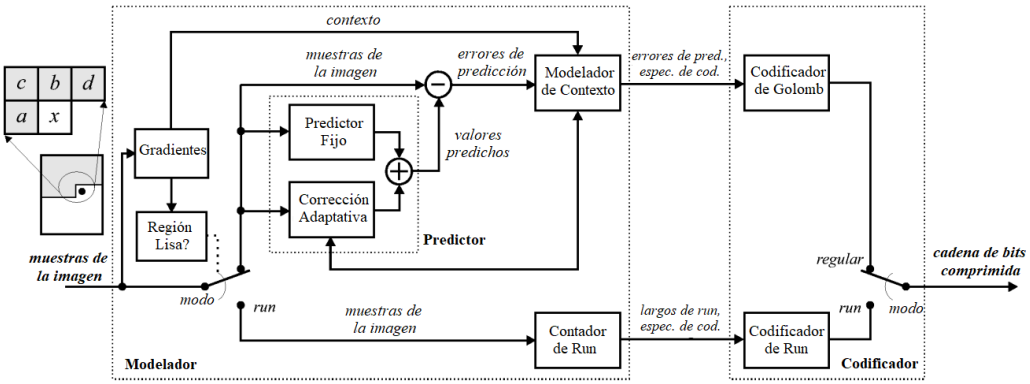

Figura 3.3: Flujo del procesamiento de las muestras en LOCO-I. Extraída de [12].

3.2. Formato JPEG-LS

El formato JPEG-LS [2], se basa en el algoritmo LOCO-I para lograr buenos niveles de compresión y baja complejidad computacional. Hereda varias características de JPEG, principalmente respecto a la estructura del archivo. A continuación se profundizará sobre algunos aspectos del formato.

3.2.1. Estructura de archivo JPEG-LS

En esta sección se detalla la estructura de los archivos que siguen la norma JPEG-LS.

Para que un archivo sea válido según la norma JPEG-LS, debe tener una estructura que siga una serie de restricciones definidas a continuación. Su estructura es bastante similar a la de JPEG, siguiendo un esquema jerárquico de estructuras anidadas que conforman al archivo. Estas estructuras son las siguientes:

- Interchange format
- Frame
- Scan
- Restart interval
- MCU

El *interchange format* es un cabezal agregado por fuera del cabezal definido en la norma, pensado para intercambiar metadata adicional entre aplicaciones. Este cabezal permite indicar la orientación de la imagen, agregar una imagen de

miniatura, entre otras cosas. Existen diferentes tipos de interchange format, como puede ser JFIF (JPEG File Interchange Format) o Exif (Exchangeable image file format). Debido al alcance establecido, en esta implementación se decidió soportar únicamente el cabezal JFIF, estandarizado a través de la norma ISO/IEC 10918-5 [14], por ser uno de los más utilizados en imágenes de internet.

El *frame* es la unidad que representa una imagen. Un archivo puede contener múltiples frames. Cada frame tiene un cabezal donde se define cuántos componentes (ver sección 2.1) tiene la imagen contenida, y el identificador de cada uno de ellos.

Los frames están compuestos por *scans*. Un frame siempre debe contener al menos un scan, aunque puede haber más. Cada scan representa una porción de los datos de la imagen, y puede contener información de uno o más componentes, especificando cuáles de ellos están contenidos a través de la información de un cabezal de scan.

También se puede definir opcionalmente *restart intervals*. En caso de existir, los restart intervals dividen un scan en segmentos, con la particularidad de que al comenzar cada segmento se reinician las estadísticas del codificador/decodificador. Esto permite aplicar una recuperación de errores, pensándolo como puntos de control de la imagen. Si se encuentra un segmento corrupto, se puede ignorar el contenido hasta leer el siguiente restart interval y reanudar el procesamiento desde allí. En este caso se perdería únicamente ese segmento y no la imagen en su totalidad.

Por último, la mínima unidad de datos utilizada en el procesamiento se denomina *MCU* (Minimum Coded Unit). En el caso más típico, un MCU corresponde a una línea de cada componente.

En la figura 3.4 se ilustra la estructura básica de una imagen en formato JPEG-LS, excluyendo el cabezal de interchange format. Como se puede observar, el formato JPEG-LS admite diferentes maneras de almacenar los datos. Cada frame puede representar una imagen distinta, o los componentes pueden estar distribuidos en distintos scans. De aquí surgen posibles aplicaciones en diferentes áreas, como por ejemplo la astronomía, permitiendo representar una secuencia de imágenes (cada una representada con un frame) dentro de una captura para luego fusionarlas con alguna técnica, y obtener una mejor calidad de imagen. Mediante múltiples frames también se pueden representar animaciones, como lo hace GIF. Las tomografías son otro caso de uso donde el almacenamiento de múltiples frames puede ser aprovechado, utilizando cada frame para representar una sección de la tomografía.

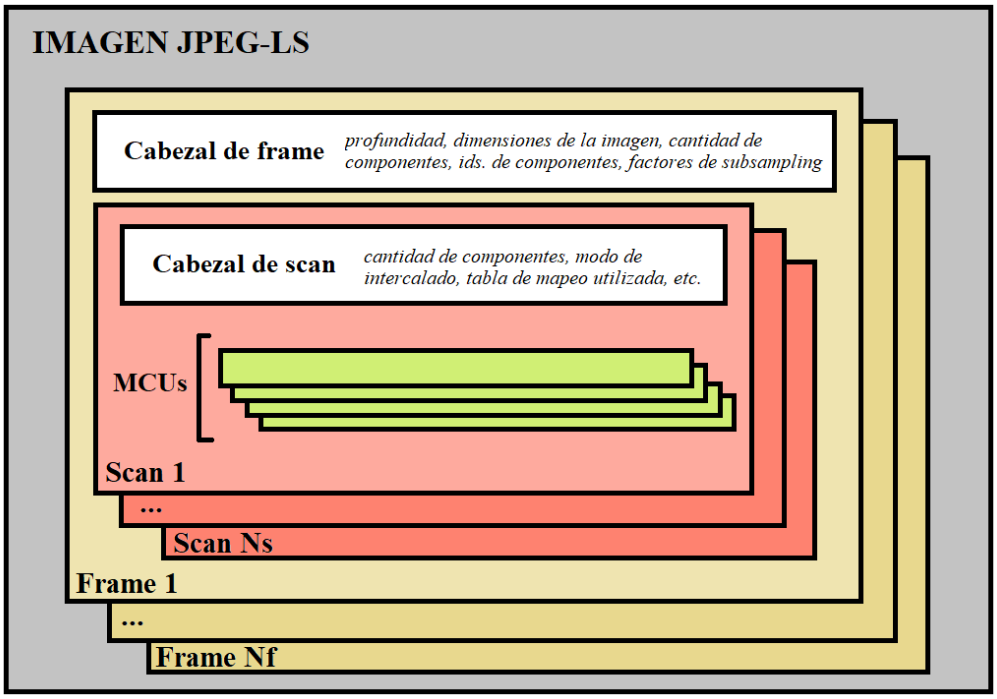

Figura 3.4: Esquema de la estructura básica de un archivo JPEG-LS. Opcionalmente, previo al primer frame se puede incluir un cabezal adicional del tipo Interchange format. A su vez, los MCUS pueden ser encapsulados en restart intervals.

3.2.2. Marcadores

Al igual que en el formato JPEG, en JPEG-LS la estructura de la imagen contiene *marcadores* (o *markers*). Un marcador es una palabra de 2 bytes con el formato 0xFFhh, que debe cumplir las siguientes condiciones para ser válido:

- El bit más significativo de hh debe ser 1.
- hh tiene que ser distinto de 0xFF.

De esta forma, 0xFFFF no es un marcador válido pero 0xFFD7 sí lo es.

Hay dos tipos de marcadores:

- Stand-alone: Son marcadores por sí solos, que no contienen ningún dato adicional. Por ejemplo `SOI` (Start Of Image), un marcador que indica el comienzo de la codificación de una imagen o el marcador `EOI` (End Of Image) que indica el fin de la codificación, son marcadores de tipo Stand-Alone.

- Marker-segment: Son marcadores que van acompañados de un segmento de datos. Por ejemplo, el marcador `SOF` (Start Of Frame), que como su nombre dice indica el comienzo de un frame, es seguido por información del frame, especificando sus dimensiones, cantidad de componentes, profundidad de las muestras, entre otros datos.

El marcador `SOI` tiene como único propósito indicar el comienzo de la codificación de la imagen, por lo que no necesita ningún dato adicional y por eso es de tipo stand-alone. Por otro lado, `SOF` no solo indica el comienzo de un frame, sino que además incluye un segmento de datos con información asociada a dicho frame, por lo que es de tipo marker-segment.

Los marcadores solo pueden aparecer en ubicaciones puntuales de la codificación. Estas ubicaciones permitidas dependen de cada marcador. La mayoría de marcadores solo pueden ocurrir previo a la codificación de un frame o scan, mientras que otros como el `RSTm` que indica el inicio de un nuevo restart interval debe ocurrir previo al comienzo de la codificación de un nuevo MCU.

En la figura 3.5 se puede apreciar la estructura de un archivo en formato JPEG-LS a partir de sus marcadores. Los marcadores `SOI` y `EOI` son mandatorios y deben ocurrir una única vez al comienzo y al final de la imagen respectivamente. El marcador LSE es opcional y se utiliza para definir distintos parámetros como los límites de cuantización de gradientes, frecuencia en el reinicio de estadísticas de contextos, tablas de mapeo, entre otros. El `SOS` (Start of Scan), indica el comienzo de un scan y como datos incluye el tipo de intercalado, la tabla de mapeo utilizada (en caso de usar una), entre otros. Cada frame o scan puede ser precedido opcionalmente por marcadores `LSE` para la redefinición de algunos parámetros, definición de tablas de mapeo, transformadas de color, entre otros. El `largo` indica el tamaño en bytes del segmento de datos del marcador.

A continuación se muestra una lista de los marcadores de JPEG-LS que fueron heredados de la especificación de JPEG original.

Marcador	**Descripción**	**Valor**
SOI	Comienzo de imagen	0xFFD8
EOI	Fin de imagen	0xFFD9
SOS	Comienzo de scan	0xFFDA
DNL	Define el número de lineas	0xFFDC
DRI	Define el restart interval	0xFFDD
RSTm	Reseteo (FFD0-FFD7)	0xFFD0
COM	Comentario	0xFFFE
APP0	Marcador JFIF	0xFFE0

Se presenta a continuación una lista con los marcadores definidos en la norma JPEG-LS. Un detalle importante a mencionar, es que en la norma de JPEG

Marcador	Descripción	Valor
SOF	Comienzo de frame JPEGLS	0xFFF7
LSE	Extensión de JPEGLS	0xFFF8
LSE_PARAMS	Redefinición de parámetros	0x01
LSE_MAPTABLE	Definición de tablas de mapeo	0x02
LSE_XMAPTABLE	Continuación de tablas de mapeo	0x03
LSE_OVERSIZE_DIMS	Definición de imágenes sobredimensionadas	0x04
LSE_COLOR_TRANS	Definición de transformadas de color	0x0D

también se define el marcador `SOF`, pero con un valor diferente al especificado por JPEG-LS.

El marcador `LSE` debe estar compuesto por el marcador en si mismo, seguido del largo del segmento y sus datos. Los datos del segmento están compuestos por el identificador, cuyos valores posibles están especificados con el prefijo `LSE` en la tabla anterior, y los parámetros que dependen de cada identificador.

3.2.3. Múltiples componentes

La codificación mediante el algoritmo LOCO-I está definida para un único componente. Para imágenes de múltiples componentes, la sintaxis de JPEG-LS soporta tanto modos *intercalados* como *no intercalados*. Para el caso del modo sin intercalado, cada componente es codificado en un scan separado, como si se tratara de una imagen en escala de grises. Por otro lado, JPEG-LS ofrece dos modos de intercalado: por líneas o por muestras. El intercalado por líneas consiste en codificar en su totalidad el primer componente de la primera línea de la imagen, luego el segundo componente de la primera línea, hasta completar la codificación de todos los componentes, para luego repetir el proceso para codificar la siguiente línea. El intercalado por muestras recorre las líneas y para cada píxel codifica todos sus componentes antes de procesar el siguiente píxel.

En el caso de existir múltiples componentes en un scan las estadísticas de contexto se comparten para todos los componentes. De todas formas, los procedimientos de predicción y modelado de contextos se realizan independientemente para cada muestra, es decir, las muestras de un componente no se utilizan para computar los contextos de las muestras de los otros componentes.

En estos casos, la posible correlación entre los planos de color se usa de forma limitada, únicamente como consecuencia de compartir las estadísticas de contextos entre componentes. De todas formas, para algunos espacios de color, como RGB, se puede lograr una buena decorrelación de los componentes mediante la aplicación de transformadas de color simples sin pérdida (explicada en la sección 3.2.4) como un preprocesamiento a la codificación. Esto permite aprovechar la redundancia entre componentes, mejorando la tasa de compresión, aunque cada componente (transformado) se codifique por separado.

Figura 3.5: Esquema de la estructura de una imagen JPEG-LS a partir de sus marcadores.

3.2.4. Funcionalidades

A continuación se entrará en detalle sobre los conceptos detrás de algunas de las funcionalidades que ofrece JPEG-LS.

Submuestreo

Las imágenes están compuestas por píxeles, que a su vez están compuestos típicamente por tres componentes (RGB). El submuestreo ocurre cuando en una imagen hay menos información de un componente que de los demás, es decir, ese componente tiene menos muestras que los otros.

Además de la popular representación de colores a traves del espacio de color RGB, los colores también se pueden representar a traves del esquema Y'CbCr. En este espacio de color Y' es el componente de luminancia, que representa la luminosidad del píxel, mientras que Cb y Cr son los componentes de *crominancia* y representan el contenido espectral.

Debido a que el ojo humano es mucho más sensible a variaciones de luminosidad que de color, los algoritmos de compresión de imágenes se pueden aprovechar de esto mediante el aumento de la información almacenada asociada a la luminosidad de la imagen con respecto a la información asociada a su color. De esta forma, al trabajar en el espacio de color Y'CbCr se puede reducir a la mitad la información asociada a la crominancia y el ojo humano apenas distinguiría la diferencia. Este esquema es denominado 4:2:2, ya que por cada 4 muestras del componente Y', se almacenan 2 muestras de los componentes Cb y Cr. A estos valores se les llama *factores de submuestreo*. La aplicación del submuestreo en este caso logra reducir a dos tercios el espacio necesario para almacenar la imagen (previo a la compresión).

Esta funcionalidad es soportada por JPEG y también por JPEG-LS.

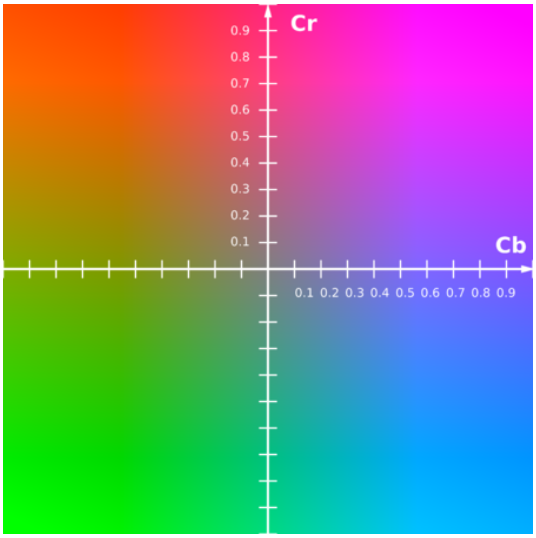

Figura 3.6: Representación gráfica de los componentes CbCr para Y'=0.5

Tablas de mapeo

El objetivo detrás de las tablas de mapeo es restringir los posibles colores de una imagen a un número máximo (típicamente 256) de colores definidos en una *paleta de color*. De esta forma, en lugar de almacenar para cada píxel el valor de los tres componentes asociados a su color, alcanzaría con almacenar un único valor que identifique al color en la paleta. Para el caso de 256 colores, utilizando tablas de mapeo el espacio de almacenamiento de la imagen se reduce a un tercio de la versión original, una reducción muy significativa.

Pero esta reducción del espacio de la imagen viene acompañada, como es de esperar, de una reducción en la calidad debido a la aproximación que se le aplica al color de cada píxel para asociarlo a un color de la paleta definida. Si la imagen original ya tenía 256 colores o menos, la calidad no se ve afectada. Pero si la imagen tenía miles o millones de colores (una imagen estándar con 3 componentes de 8 bits de profundidad puede tener más de 16 millones de colores posibles) entonces la reducción de calidad es notoria. En la figura 3.7 se puede apreciar el resultado de aplicar esta técnica para una imagen de muchos colores.

Como consecuencia, la aplicación de paletas de color es particularmente útil en imágenes simples, con poca variedad de colores como gráficos o logos, ya que reduce considerablemente su espacio sin afectar la calidad en estos casos.

La elección de los colores de la paleta es un problema importante, que influye en gran manera sobre la imagen resultante. Para solucionar este problema existen algoritmos que, en base a la imagen original, definen los colores de la paleta de forma de minimizar la pérdida de calidad para el ojo humano.

En caso de definir una paleta de colores, la información correspondiente se debe incluir como metadata previo a la codificación de la imagen propiamente dicha.

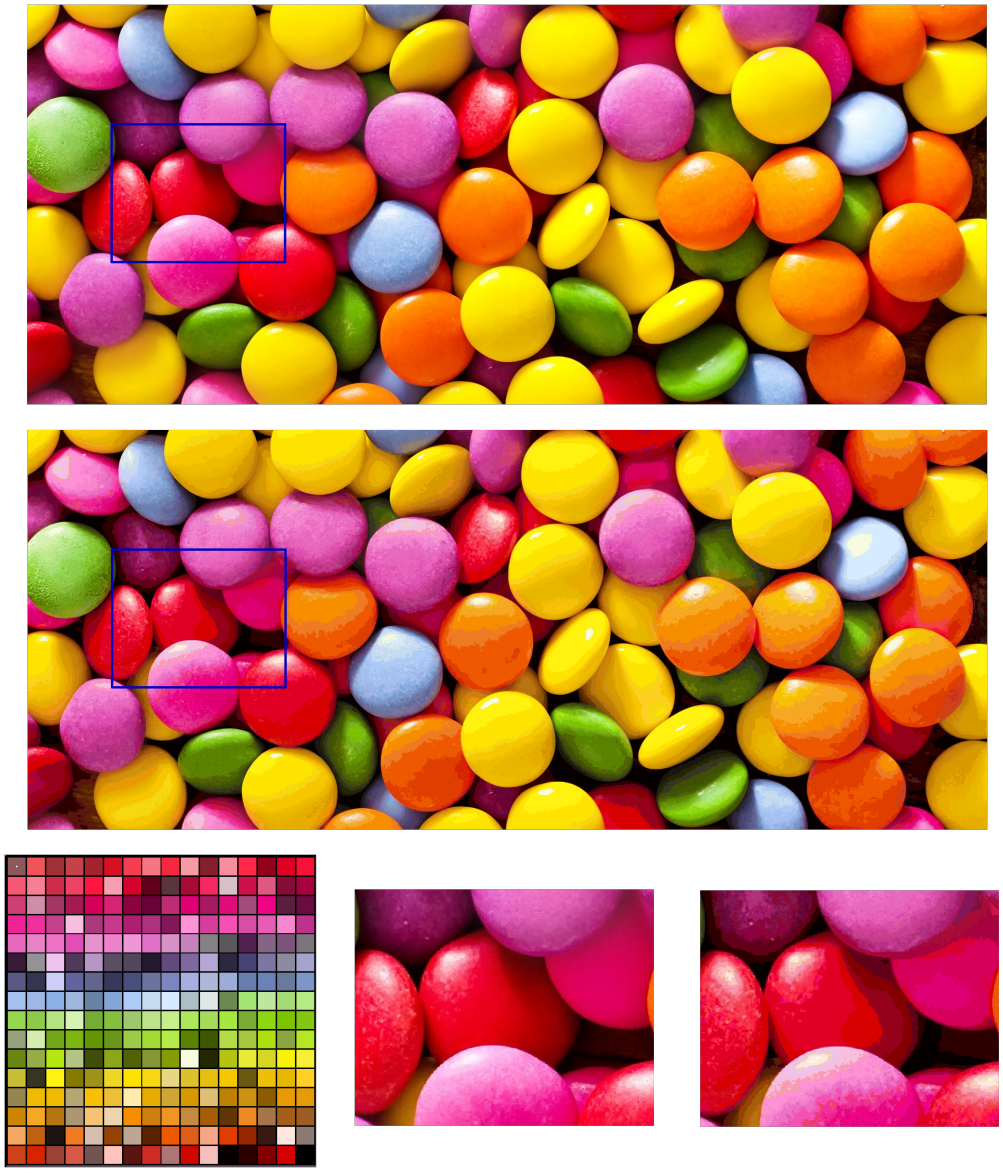

Figura 3.7: Comparación entre una imagen cruda y una imagen paletizada con 256 colores. En la parte superior se puede apreciar la imagen original (arriba) junto a su versión paletizada (debajo). A su vez, en la parte inferior se observa la paleta de colores utilizada (izquierda) y una ampliación del recuadro marcado en azul en la imagen original (centro) y en la paletizada (derecha), lo cual permite apreciar en detalle la pérdida de calidad que puede producir esta técnica.

El formato GIF está definido en torno a esta funcionalidad, pudiendo codificar únicamente imágenes de hasta 256 colores definidos en una paleta. JPEG-LS incorpora la funcionalidad de imágenes paletizadas como una configuración opcional, con el nombre de *tablas de mapeo*.

Transformadas de color

Se conoce que en imágenes naturales los componentes de un espacio de color (por ejemplo RGB) pueden mantener una cierta correlación, es decir, los valores de cada componente no son independientes de los valores de los demás.

La aplicación de transformadas de color tiene como objetivo minimizar la correlación entre los componentes para así poder codificar cada uno separadamente, sin generar una cantidad excesiva de información redundante. El objetivo se puede lograr en mayor o menor medida, dependiendo de múltiples factores de cada caso en particular.

Las transformadas de color consisten en aplicar para cada píxel una operación reversible entre los tres componentes, obteniendo como resultado otros tres componentes con menor correlación. Para obtener los componentes originales, el decodificador simplemente debe aplicar la operación inversa sobre los componentes de cada píxel.

Una de las transformadas de color más simples y a la vez más populares para el espacio de color RGB consiste en aplicar la siguiente transformación: $(R, G, B) \rightarrow (R - G, G, B - G)$. Esta operación es aplicada resolviendo las diferencias módulo α (siendo α el tamaño del alfabeto) en el intervalo $[-\lfloor \alpha/2 \rfloor, \lceil \alpha/2 \rceil - 1]$ (como se mencionó en la sección 3.1 para errores de predicción, esta reducción modular no afecta la invertibilidad). De esta forma, si para un píxel los valores RGB son (100, 120, 150), y el tamaño del alfabeto es 256, el resultado de la transformada es (-20, 120, 30). Esta operación es aplicada a cada píxel de la imagen a forma de preprocesamiento. Aplicando esta transformada recién descrita se pueden obtener, en algunos casos, mejoras en la tasa de compresión de entre el 25 % y 30 % frente a codificar sin su aplicación. De todas formas, algo importante a destacar es que dicha aplicación no garantiza una mejora en este sentido, sino que depende de factores de cada imagen como la resolución, si es una imagen natural o artificial, entre otros. Por lo tanto, la transformada de color es opcional, y el usuario debe decidir en cada caso si le conviene utilizarla o no.

En la extensión del formato JPEG-LS [3] se define el soporte para transformadas de color, que permite al decodificador invertir la transformada de color aplicada sobre la imagen, obteniendo así la imagen original.

Capítulo 4

Arquitectura e implementación

En este capítulo mostramos los elementos más importantes considerados en el diseño e implementación de la biblioteca. Mencionamos aspectos generales de la arquitectura, así como el detalle de las estructuras definidas en la implementación y la función que cumple cada una.

4.1. Lenguaje, sistemas operativos

Uno de los objetivos es el de brindar soporte a diversos casos de aplicación. En este sentido tomamos medidas tanto en términos de compatibilidad de versiones utilizadas, así como recursos necesarios para soportar la biblioteca. En base a los objetivos planteados, desarrollamos la biblioteca en lenguaje C, en particular utilizando una versión de C bien establecida y soportada por la mayoría de los dispositivos ya existentes sin necesidad de una actualización. La versión de C utilizada es C99 (ISO/IEC 9899:1999).

Esta biblioteca soporta los sistemas operativos Windows, Linux y macOS. Los requerimientos de software necesarios para la compilación de la biblioteca son el compilador *gcc* y el utilitario *make*. Más detalles sobre la compilación de la biblioteca se pueden encontrar en el documento README [7].

4.2. Características de la solución

En esta sección detallamos las características de la implementación desarrollada. Siguiendo con los objetivos planteados, la solución busca satisfacer fácilmente las necesidades de la mayoría de los usuarios, así como también adaptarse a diversos escenarios para contemplar la mayor cantidad de casos de uso posibles. Además de la biblioteca propiamente dicha, se brindan ejecutables para la compresión y descompresión de imágenes puntuales.

[7] https://github.com/andresk727/libJLS/blob/main/README.md

4.2.1. Modularidad

La solución está implementada en módulos, cada uno encapsulando sus funcionalidades particulares. Se implementó el módulo `common`, que implementa las funcionalidades comunes. Por otro lado se implementaron los módulos `encoder` y `decoder` para el codificador y decodificador respectivamente, cada uno implementando sus funcionalidades particulares, y utilizando el módulo `common`. De esta forma, el codificador y decodificador son unidades independientes entre sí, que pueden estar en una misma biblioteca o en bibliotecas completamente separadas.

4.2.2. Profundidad de bits

El término profundidad de bits, o profundidad de color, se refiere a la cantidad de bits usados para la representación del color de un componente de un píxel en una imagen no comprimida. Si bien en esta implementación se admiten imágenes con una profundidad de entre 2 y 16 bits, cabe destacar que para su compilación se debe diferenciar entre los casos de 2-8 bits y de 2-16 bits. Esta restricción se debe a aspectos de optimización, ya que a nivel de implementación se utiliza un tipo de datos diferente para representar una muestra. Para esto se utiliza el tipo `JLS_SAMPLE`, del cual entramos en detalle en secciones posteriores. Esta distinción se realiza mediante la definición de una macro, aprovechando la particularidad de que su evaluación y reemplazo se realiza en tiempo de compilación, por lo que en consecuencia se compila una biblioteca optimizada para cada caso.

4.2.3. Interfaz

En cuanto a interfaz de usuario, ofrecemos dos opciones diferentes, diseñadas para distintos tipos de usuario y casos de uso.

- Interfaz simplificada.
- Interfaz extendida.

Detallamos sobre cada una de estas en la sección 4.5.

4.2.4. Funcionalidades

A continuación listamos las principales funcionalidades soportadas por la implementación.

- Procesamiento de imágenes con profundidad de entre 2 y 16 bits.
- Soporte para imágenes con canal alpha (transparencia, definido en la sección 4.2.5).
- Submuestreo (definido en la sección 3.2.4).

- Tablas de mapeo (definidas en la sección 3.2.4).
- Transformadas de color (definidas en la sección 3.2.4).
- Procesamiento de imágenes por líneas.
- Soporte para cabezal JFIF.

4.2.5. Transparencia

El formato JPEG-LS no soporta nativamente ningún tipo de imágenes con transparencia, ni de forma binaria como ofrece GIF (cada píxel puede ser completamente transparente o completamente opaco), ni como un componente adicional de la imagen. Sin embargo, en la implementación de libJLS se brinda soporte a este tipo de imágenes, permitiendo añadirle un componente adicional dedicado a la transparencia. Dado que la norma no soporta este tipo de imágenes, se implementaron mecanismos internos para codificar y decodificar las imágenes de este tipo.

La biblioteca dispone de funciones para agregar un componente de transparencia adicional, llamado `canal alpha`. Este componente es codificado en el último scan, independientemente a los demás componentes. Para identificar una imagen con transparencia, la implementación codifica un comentario (a través del uso del marcador `COM`) con la palabra `ALPHA` en el cabezal de la imagen. Cuando se decodifica la imagen, si el decodificador encuentra un comentario indicando que la imagen contiene un canal de transparencia, sabe que debe procesar el último scan de forma especial. Esta interpretación del codificador y decodificador, es particular de esta implementación y no está contemplada bajo la norma.

4.2.6. Configuraciones incompatibles

Existen combinaciones de funcionalidades que no están admitidas en la biblioteca, ya sea por restricciones propias de la norma, o por decisiones de implementación. Por ejemplo, las transformadas de color y el submuestreo no son compatibles mutuamente debido a que para aplicar una transformada de color es necesario que todos los componentes tengan las mismas dimensiones. Algo similar ocurre en los otros casos, por ejemplo, para las tablas de mapeo y transformadas de color o submuestreo, que tampoco son compatibles. Algunas otras configuraciones tienen restricciones muy puntuales y se encuentran detalladas en el anexo 8.1.

4.3. Flujo de operaciones

En esta sección describimos el flujo de operaciones típico para hacer uso de la biblioteca. También presentamos los parámetros de codificación, indicando cuáles

pueden ser modificados por el usuario. Para una guía más detallada, revisar el documento de guía de usuario, en el anexo 8.1.

4.3.1. Inicialización

El primer paso que se debe realizar al procesar una imagen es la inicialización de la estructura de control (definida en la sección 4.4.1), tanto para el caso del codificador como del decodificador. En esta instancia, se reserva memoria y se inicializa la estructura de datos utilizada.

4.3.2. Configuración

La configuración se realiza mediante la modificación parámetros, que son metadatos de la imagen u opciones de procesamiento; las funciones de compresión y descompresión tienen diferentes comportamientos en función de los parámetros especificados.

A la hora de comprimir una imagen, estos son algunos de los parámetros que acepta el compresor:

- Dimensión de la imagen.
- Cantidad de componentes.
- Profundidad de bits.
- Modo de intercalado.
- Factor de submuestreo (opcional).
- Tabla de mapeo (opcional).
- Transformadas de color (opcional).

Una vez que se inicializa correctamente la estructura para el procesamiento, se deben configurar los parámetros de codificación según la imagen a procesar.

En este sentido el proceso de decodificación es más sencillo para el usuario, ya que la información asociada a estos parámetros se obtiene del cabezal de la imagen codificada.

4.3.3. Procesamiento

Luego de que se configura la estructura con los parámetros requeridos se inicia el proceso de codificación/decodificación. Una vez comenzado el procesamiento, los parámetros no pueden ser modificados.

4.3.4. Finalización

Al finalizar el procesamiento del codificador o decodificador, es necesario liberar la memoria reservada inicialmente para la estructura de control. La biblioteca dispone de funciones específicas para realizar esta tarea.

4.4. Estructura de control y tipos de datos

En esta sección mostramos la estructura de control utilizada y los tipos de datos definidos en la implementación, detallando el uso de cada uno de ellos.

4.4.1. Estructura de control

La estructura en la que se centra toda la implementación es `jls_control_struct`. Es una estructura de control común a la codificación y decodificación utilizada durante todo el procesamiento. Si bien la estructura no es visible directamente al usuario, la API provee funciones para su inicialización y manipulación, permitiendo mantener el estado del codificador o decodificador entre llamadas a funciones de la biblioteca.

A continuación se muestran y detallan los campos de la estructura de control con las que el usuario puede interactuar a través de la API. Si bien existen otros campos pertenecientes a esta estructura, estas son de uso interno para mantener el estado del codificador y decodificador.

```
struct jls_control_struct
{
  // USER-ACCESIBLE PARAMETERS

  // Image to be processed
  JLS_SAMPLE_ROW *image_buffer;

  // Image dimensions
  uint32_t image_width;
  uint32_t image_height;

  // Sample depth in the range of (2-16)
  unsigned int sample_depth;

  // Interleaved mode. Values may be 0 (plane), 1 (line) or 2
    (sample)
  unsigned int ilv_mode;

  // Boundaries of the regions in gradient quantization
  unsigned int quant_regions_boundaries[T_QUANTIZATION];
```

```
// Current component/channel
unsigned int channels;

// Number of possible sample values (256 by default)
int range;

// Value of N at which encoding/decoding statistics are
 halved
int reset;

// Subsampling factors
uint8_t *subsampling_h;
uint8_t *subsampling_v;

// Restart interval
uint32_t rest_interval;

// Used for buffer access. Values may be 0 (plane), 1 (line
 ) or 2 (sample)
unsigned int buffer_ilv;
// Buffer dimensions
uint64_t buffer_width;
uint64_t buffer_height;
// Used for buffer access
int64_t buffer_current_row;
int64_t buffer_current_column;

// Array of pointers to the loaded mapping table
struct jls_mapping_table *mapping_table;
// True if the user wants to automatically map the index
 buffer to a multiple-component image at the end
// of the decoding process (if a mapping table was loaded).
  False if the user prefers to keep access to the
// original index buffer and the mapping table instead.
bool auto_map_image;

// Color transform
struct jls_inv_color_transform_data
      current_color_transform;

// Indicates if a JFIF marker should be written or was read
bool jfif_marker;
// JFIF parameters (in case a JFIF marker is present)
struct JFIFParameters *jfif_params;

// Every read/written comment
struct jls_comments *comments;
```

```
    // Alpha channel
  bool using_alpha_channel;
  // Buffer for alpha channel
  JLS_SAMPLE *alpha_buffer;

  // Read/Written bytes
  uint64_t bytes_compression;

  // Exit code of last execution
  int exit_code;
}
```

- `image_buffer`: buffer con los datos a procesar. Si se está codificando contiene las muestras de la imagen cruda a codificar, y si se esta decodificando contiene las muestras de la imagen cruda decodificada. En el caso de procesamiento por líneas, contiene la porción actual de líneas que se están procesando.
- `image_width`: ancho de la imagen.
- `image_height`: altura de la imagen.
- `sample_depth`: profundidad de bits, en el rango 2-16.
- `channels`: cantidad de componentes.
- `ilv_mode`: modo de intercalado 3.2.3 para el procesamiento (0 Plane, 1 Line, 2 Sample).
- `quant_regions_boundaries`: arreglo que define los límites (T_1, T_2, T_3, definidos en la sección 3.1) de las regiones para la cuantización de gradientes.
- `range`: tamaño del alfabeto (valor máximo para las muestras). Por defecto, este valor equivale a $2^p - 1$, siendo p la profundidad de bits, pero puede ser redefinido a un valor menor si se sabe que ninguna muestra supera dicho valor.
- `reset`: valor límite N_0 en el que al ser alcanzado por el contador N, las estadísticas de contexto A, B y N (definidos en la sección 3.1) son divididas a la mitad.
- `subsampling_h` y `subsampling_v`: array con factores de submuestreo (definidos en la sección 3.2.4) para cada componente.
- `rest_interval`: cantidad de MCUs dentro de un restart interval.
- `buffer_ilv`: modo de intercalado en el que están organizadas las muestras de `image_buffer`. Este modo puede ser distinto al `ilv_mode` ya que, por ejemplo, el buffer puede contener las muestras intercaladas por muestras pero el usuario puede querer codificar la imagen con intercalado por líneas.

- `buffer_width`: ancho del buffer que almacena los datos de la imagen.
- `buffer_height`: cantidad de filas del buffer que almacena los datos de la imagen.
- `mapping_table`: referencia a la tabla de mapeo cargada [8].
- `auto_map_image`: si se usa una tabla de mapeo de colores, indica si la imagen debe ser mapeada automáticamente al ser decodificada, o si debe retornar el array de índices a la tabla. Por defecto mapea automáticamente.
- `current_color_transform`: referencia a la transformada de color aplicada (si la hay).
- `jfif_marker`: indica si existe un marcador JFIF.
- `jfif_params`: parámetros del marcador JFIF.
- `comments`: lista de comentarios dentro de la imagen codificada.
- `using_alpha_channel`: indica si la imagen procesada contiene un canal de transparencia.
- `alpha_buffer`: en caso de tener un canal de transparencia, esta variable mantiene una referencia al buffer que contiene las muestras de ese canal.
- `exit_code`: código del resultado de la ejecución.

Dado que el lenguaje C no soporta variables privadas, para evitar que el usuario modifique indebidamente la estructura de control utilizamos un *tipo opaco* para declararla en la API. Esto consiste en declarar la estructura en el cabezal público sin especificar sus campos, los cuales serán especificados en un cabezal interno, al que el usuario no tiene acceso. Una simplificación de esta técnica se puede apreciar en el siguiente pseudocódigo.

```
/// libJLS.h (publico para el usuario)
struct jls_state_struct;
typedef struct jls_state_struct *jls_state_struct_p;


/// jlstypes.h (privado para el usuario)
struct jls_control_struct
{
    /// Campos de la estructura de control
```

[8] La norma soporta la carga de múltiples tablas de mapeo, con la posibilidad de aplicar una distinta en cada scan. Debido al alcance establecido, la implementación realizada solo soporta la carga de una única tabla.

```
}


/// libJLS.c (privado para el usuario)
int jls_encode_image(jls_state_struct_p s_struct_p,
user_parameters);
{
    struct jls_control_struct *c_struct =
        (struct jls_control_struct *)s_struct_p;
    // (...)
}


/// Codigo del usuario
jls_state_struct_p s_struct_p;
// Inicializacion y configuracion de la estructura
// (...)
jls_encode_image(s_struct_p);
```

De esta forma el usuario puede interactuar con la estructura de control (inicializarla, pasarla como parámetro de múltiples funciones, liberarla) únicamente a través de las funciones que ofrece la API (algunas de estas funciones son mencionadas en la sección 4.5.1), ya que no tiene acceso a sus campos. Esto favorece considerablemente a la robustez de la biblioteca, disminuyendo el factor de error que puede introducir el usuario al manejarla.

Para comenzar el proceso de codificación, el usuario debe necesariamente especificar la imagen cruda que desea codificar, el modo de intercalado del buffer que la contiene, sus dimensiones, el número de componentes y el modo de intercalado de la codificación, además del archivo o el buffer donde quiere que quiere almacenada la imagen codificada. Los demás parámetros tienen valores definidos por defecto, aunque como se mencionó previamente, estos valores pueden ser redefinidos a través de funciones que ofrece la API. En caso de que el usuario no tenga la imagen completa en memoria, puede codificar la imagen por líneas indicándole a la API las líneas a codificar. En esta situación el usuario puede no conocer de antemano la altura de la imagen, por lo que la biblioteca permite comenzar la codificación sin haber especificado la altura, para luego hacerlo eventualmente mediante el uso del marcador DNL (Define Number of Lines), que puede ocurrir al finalizar cualquier MCU.

Para el caso del decodificador, el usuario únicamente debe proveer el buffer donde desea almacenar la imagen decodificada, y especificar el archivo o el buffer donde se encuentra la imagen a decodificar. Todos los demás campos son obtenidos por el decodificador a través de los cabezales de frames y scans, o del contenido de los marcadores LSE presentes en la imagen codificada. Una vez culminado el proceso de decodificación, el usuario puede acceder a los datos de la imagen decodificada a través de una estructura pública que contiene únicamente los datos relevantes para el usuario, la cual mostramos a continuación.

```
struct jls_dec_img_data
{
    /// Image width
    uint32_t image_width;

    /// Image height
    uint32_t image_height;

    /// Image sample depth, in range 2-16
    unsigned int sample_depth;

    /// Channels
    unsigned int channels;

    /// Coding interleaved mode
    unsigned int ilv_mode;

    /// Subsampling horizontal factors
    uint8_t *subsampling_h;

    /// Subsampling vertical factors
    uint8_t *subsampling_v;

    /// Indicates if the image is indexed to a mapping
table (should be mapped to obtain the actual image)
    bool uses_mapping_table;

    /// JFIF parameters (NULL in case no JFIF data was
read)
    struct jls_jfif_params *jfif_params;

    /// Every comment read
    struct jls_comments *comments;

    /// Reference to the buffer containing the decoded
alpha channel (NULL if the decoded image doesn't have an
alpha channel)
    JLS_SAMPLE *alpha_buffer;
}
```

Esta estructura puede ser obtenida a través de una función que ofrece la API, y es cargada internamente a partir de los campos de la estructura de control.

4.4.2. Tipos

Dado que la representación interna de cada tipo de datos puede variar entre los diferentes sistemas operativos o compiladores (lo cual podría implicar diferentes errores en el manejo de datos), y tomando en cuenta su relevancia en términos de

eficiencia y optimización, se decidió utilizar tipos que no fueran ambiguos y que brinden la certeza de que su implementación no varía en los diferentes ambientes.

Los tipos más utilizados son:

- `uint8_t`
- `uint16_t`
- `uint32_t`
- `uint64_t`
- `int64_t`

Como se mencionó anteriormente, en la biblioteca se admiten imágenes con profundidad de color de entre 2-16 bits. Considerando la eficiencia, una buena optimización implica definir diferentes tipos de datos para representar las muestras de una imagen, dependiendo de la profundidad de color que se vaya a utilizar. Esto se debe a que, para imágenes de hasta 8 bits, cada muestra puede ser almacenada en datos de 1 byte, mientras que para imágenes de entre 9-16 bits es necesario utilizar 2 bytes. No sería muy eficiente en términos de memoria utilizar siempre el tipo de mayor tamaño.

De esta forma, se define el tipo `JLS_SAMPLE`, implementado con tipos de datos diferentes según el modo en el que se haya compilado la biblioteca.

La definición del tipo queda en función de la macro de compilación `TWO_BYTES_SAMPLE`:

```
#ifdef TWO_BYTES_SAMPLE
    typedef uint16_t JLS_SAMPLE;
#else
    typedef uint8_t JLS_SAMPLE;
#endif
```

Para el caso de imágenes con profundidad de bits entre 2-8 bits, el tipo de dato utilizado es `uint8_t`, mientras que para imágenes de entre 9-16 bits se utiliza `uint16_t`.

Por otro lado se define el tipo `JLS_SAMPLE_ROW`, que consiste en un array de elementos de tipo `JLS_SAMPLE`. Este tipo de datos se utiliza para representar al buffer de la imagen a procesar.

4.5. Diseño de API

En esta sección comentamos los aspectos considerados para el diseño de la API, detallando las funciones ofrecidas al usuario a través de la biblioteca.

Nos referimos al usuario como el que hará uso de las bibliotecas. Este tiene acceso a las bibliotecas y las puede compilar por su cuenta. A su vez, tiene acceso a los cabezales de los archivos `.h` que definen las APIs y contienen exclusivamente las estructuras y datos a los que el usuario está autorizado a acceder. Los binarios de las bibliotecas, y estos cabezales públicos son lo que el usuario incorpora a su proyecto.

Ofrecemos una interfaz simplificada de la biblioteca para aquellos usuarios que desean utilizarla sin conocer demasiados detalles. Con esta interfaz se utilizan los parámetros de codificación por defecto. También ofrecemos una interfaz completa para los usuarios que buscan tener más control sobre el proceso de codificación o decodificación, con la posibilidad de alterar diferentes parámetros de procesamiento y utilidades aplicables a las imágenes (por ejemplo, transformadas de color). A su vez, dentro de esta interfaz avanzada se ofrece la opción de procesar la imagen por líneas. Esto le permite al usuario realizar el procesamiento de la imagen sin tenerla almacenada en su totalidad, realizando llamados a la biblioteca a medida que recibe fragmentos de ella. Esta funcionalidad resulta particularmente útil, por ejemplo, cuando se trabaja con imágenes de grandes dimensiones en un dispositivo que no tiene suficiente memoria disponible para almacenar la imagen en su totalidad.

Las funciones que ofrece la biblioteca están separadas en tres categorías, bajo los directorios `common`, `encoder` y `decoder`. En `common` se encuentran las funciones que son comunes al proceso de codificación y decodificación, como puede ser la función para la inicialización de estructuras. En `encoder` y `decoder` se encuentran las funciones específicas de cada uno de estos procesamientos.

4.5.1. Principales funciones

A continuación describimos las principales funciones de la interfaz de la biblioteca, indicando sus utilidades. Cabe destacar que todas las funciones utilizan como parámetro la estructura de control 4.4.1. Una guía de uso más detallada se puede encontrar en el anexo 8.1.

Tanto el codificador como el decodificador disponen de funciones para la inicialización de la estructura de control como paso inicial y para su liberación al finalizar el procesamiento.

```
// Reserva memoria e inicializa las estructuras de datos
   correspondientes al codificador y decodificador
   respectivamente
int jls_init_encode_struct(jls_state_struct_p *s_struct_pp);

int jls_init_decode_struct(jls_state_struct_p *s_struct_pp);
```

```
// Finaliza el proceso de codificacion o decodificacion, y
   libera las estructuras de datos previamente inicializadas
int jls_finalize_encoder(jls_state_struct_p s_struct_p);

int jls_finalize_decoder(jls_state_struct_p s_struct_p);
```

Estas funciones se corresponden con las de las etapas de inicialización y finalización respectivamente mencionadas en la sección 4.3.

Codificación

Estas funciones se corresponden a la configuración del codificador. Si bien el orden del llamado entre las funciones que implican configuración del codificador no es importante, estas deben ser llamadas luego de la función de inicialización y antes de comenzar el procesamiento de la entrada. Si se aplica alguna configuración durante el procesamiento, se tendrá un resultado que no es el esperado.

Carga la dimensión de la imagen en la estructura de estado. Su uso es obligatorio, no se aplican valores por defecto.

```
void jls_set_image_dimensions(jls_state_struct_p s_struct_p,
    uint32_t image_width, uint32_t image_height);
```

Carga la cantidad de canales en la estructura de estado. Su uso es obligatorio, no se aplican valores por defecto.

```
void jls_set_channels_count(jls_state_struct_p s_struct_p,
    unsigned int channels);
```

Carga el modo de intercalado en la estructura de estado. En caso de no aplicarse, su valor por defecto es el correspondiente al intercalado por líneas.

```
void jls_set_ilv_mode(jls_state_struct_p s_struct_p,
    unsigned int ilv_mode);
```

Carga la profundidad de bits en la estructura de control. El valor esperando debe estar en el rango 2-16. En caso de no utilizarse, su valor por defecto es 8 o 16, dependiendo de la configuración de compilación de la biblioteca.

```
void jls_set_sample_depth(jls_state_struct_p s_struct_p,
    unsigned int depth);
```

Setea en la estructura de estado el buffer correspondiente a los datos de la imagen de entrada, indicando en qué modo se encuentran organizados los datos y el ancho de cada linea. Su uso es obligatorio.

```
void jls_set_image_buffer(jls_state_struct_p s_struct_p,
    JLS_SAMPLE *image_buffer,  unsigned int buffer_ilv,
    uint64_t line_width);
```

Mientras que para la etapa de procesamiento, se cuenta con las siguientes funciones.

Comienza el proceso de codificación, escribiendo los marcadores `SOI` y `SOF`, además de marcadores `LSE` en caso de ser necesario.

```
int jls_start_encoder(jls_state_struct_p s_struct_p);
```

Para la codificación del contenido de la imagen se dispone de las siguientes funciones.

```
// Interfaz simplificada, aplica la codificacion segun los
   parametros recibidos
int jls_encode(
    JLS_SAMPLE *image_buffer,
    unsigned int buff_ilv,
    const char *out_filename,
    uint32_t image_width,
    uint32_t image_height,
    unsigned int ilv_mode,
    unsigned int channels);

// Interfaz completa, comienza el procesamiento, requiere la
   estructura de control previamente inicializada
int jls_encode_with_config(jls_state_struct_p s_struct_p);

// Interfaz completa, codifica las lineas recibidas por
   parametro
int jls_encode_lines(jls_state_struct_p s_struct_p,
    JLS_SAMPLE_ROW *rows_p,
    uint64_t num_lines);
```

Decodificación

La decodificación es más sencilla para el usuario, ya que requiere muy poca configuración debido a que la información necesaria está contenida en la propia imagen codificada. De esta forma, su única tarea es gestionar la memoria necesaria para las estructuras utilizadas durante el procesamiento. A continuación mostramos las principales funciones a utilizar durante el proceso de decodificación. Deben ser llamadas luego de `jls_init_decode_struct`.

```
// Lee el cabezal del frame, procesa los marcadores escritos
   en esa seccion, y almacena la metadata en la estructura de
    control. Es necesario leer el cabezal en una funcion
   aparte para asi obtener las dimensiones de la imagen y
   cantidad de componentes, para poder reservar la memoria
   necesaria para almacenar la imagen decodificada
int jls_read_header(jls_state_struct_p s_struct_p);

// Version simplificada, requiere la estructura de control
   previamente inicializada
int jls_decode(jls_state_struct_p s_struct_p);

// Version completa. Prepara el procesamiento del siguiente
   scan, mediante la lectura del cabezal del scan y la
   inicializacion de multiples variables internas
int jls_start_decode_scan(jls_state_struct_p s_struct_p);

// Version completa. Decodifica las lineas recibidas por
   parametro
int jls_decode_lines(jls_state_struct_p s_struct_p,
    JLS_SAMPLE_ROW *row_pointer,
    uint64_t num_lines);
```

Tanto para la codificación como para la decodificación, en el manual de usuario se presentan algunos ejemplos de uso para la versión simplificada de la biblioteca, además de un listado completo de las funciones de que ofrece la API. Ejemplos de uso de todas las funcionalidades para la versión completa de la biblioteca se pueden encontrar en el archivo `example.c`.

4.6. Organización de directorios

En esta sección se muestra y detalla la estructura de directorios utilizados para la biblioteca. Se toma como referencia el repositorio de Git [9].

Algunas de las decisiones tomadas, como se mencionó en las secciones anteriores, tiene efecto directo en la estructura de los directorios para mantener la coherencia y organización.

```
/
├── contrib ..... herramientas y bibliotecas externas
└── libJLS
    └── include
        ├── common
        └── encoder
```

[9] https://github.com/andresk727/libJLS

```
        │   └── decoder
        ├── src
        │   ├── common
        │   ├── encoder
        │   └── decoder
        ├── images
        └── test
    ├── doc
        ├── doxygen
        ├── USERGUIDE.md ..... guia de uso
        └── example.c
    ├── README.md
    └── LICENSE
```

En el directorio contrib, se encuentra el código fuente de algunas herramientas utilizadas principalmente en la etapa de testeo. Aquí se puede encontrar la implementación de la biblioteca *pnm*, utilizada para la lectura de imágenes en ese formato. Adicionalmente se encuentra la biblioteca *genimg*, utilizada para la generación de imágenes aleatorias en función de diferentes parámetros (se profundiza al respecto en la sección 5.2.1).

Bajo el directorio `libJLS` se encuentra el código fuente de la implementación desarrollada. Dentro se encuentran `include` con los archivos `.h` con las declaraciones de las funciones y `src`, en donde están los archivos `.c` que contienen la implementación del código. Aquí también se ve reflejada una de las decisiones tomadas en el proceso de implementación, como lo es la división entre funcionalidades comunes al procesamiento de codificación y decodificación, y por otro lado las específicas a cada uno de estos. Siguiendo este lineamiento, se optó por agrupar los archivos de `include` y `src` en tres subdirectorios de acuerdo a lo mencionado. Los subdirectorios son `common`, `encoder` y `decoder`.

Siguiendo dentro del directorio `libJLS`, se encuentran `images` y `test`, que contienen algunas imágenes de prueba y códigos fuentes de ejecutables para testeo, respectivamente.

En el directorio `doc` se encuentra toda la documentación de la biblioteca, incluyendo la guía de usuario y la documentación detallada generada en Doxygen.

4.7. Herramientas utilizadas

A continuación listaremos las herramientas que fueron utilizadas para el desarrollo de la biblioteca:

- **Git/Github:** Utilizado para manejar el versionado del proyecto.
- **VirtualBox:** Decidimos utilizar máquinas virtuales para realizar el testeo de la biblioteca en Linux. En particular, la versión de Linux instalada fue Ubuntu 18.04.
- **Visual Studio:** El Debuger y el Profiler que tiene incorporado el IDE fueron utilizados durante el trabajo de desarrollo y mejora de rendimiento de la biblioteca, en la plataforma Windows.
- **Visual Studio Code:** El editor de texto utilizado para la mayoría de tipos de archivo, desde código, makefiles, documentación, etc.
- **CMake:** Utilizado para controlar los procesos de compilación en las diferentes plataformas.
- **Valgrind:** Herramienta para la detección precisa de leaks de memoria y otros problemas relacionados con su manejo. También utilizamos su herramienta *Callgrind* para resolver problemas de rendimiento mediante el análisis de los ciclos de CPU que consume cada sección del código.
- **Notepad++:** Editor de texto usado para el análisis bit a bit de los archivos .jls generados por la biblioteca y su comparación con los archivos generados por la implementación de referencia.
- **Aseprite:** Editor de imágenes pixel-art, usado para la inspección de imágenes por píxeles, con herramientas que permiten visualizar los valores de cada componente de cada píxel con facilidad.
- **LaTeX:** El sistema utilizado para la generación de este documento.
- **Doxygen:** Utilizado para generar la documentación de la biblioteca para el usuario.

4.7.1. Doxygen - Generación de la documentación

Como mencionamos en secciones anteriores, la documentación de la biblioteca para el usuario se genera utilizando la herramienta *Doxygen*. Para generar esta documentación se puede utilizar la herramienta de interfaz gráfica de Doxygen, o la herramienta por línea de comandos.

La ejecución de la herramienta por línea de comandos, se basa en un archivo de configuración con los parámetros que se utilizan para generar la documentación.

El archivo de configuración mínimo requiere el nombre del proyecto, directorios de origen y salida de documentos, como se ejemplifica a continuación:

```
PROJECT_NAME           = libJLS
OUTPUT_DIRECTORY       = %path%/libJLS/doc
EXTRACT_ALL            = YES
INPUT                  = %path%/libJLS/libJLS
RECURSIVE              = YES
GENERATE_LATEX         = NO
```

En el directorio */doc/* se encuentran diferentes archivos de configuración, a modo de ejemplo:

Configuración mínina
doxygen_config_min.txt

Configuración extendida
doxygen_config_base.txt

Configuración utilizada
doxygen_config.txt

Y en el directorio */doc/doxygen/html/* se encuentra el archivo *index.html*, el cual contiene la documentación completa de la biblioteca, generada por la herramienta en formato de página web. En la figura 4.1 se muestra una captura de ejemplo de lo mencionado.

A continuación se muestra un ejemplo de la generación del archivo de configuración, y de la documentación.

```
// generates template config file
$ doxygen -g "example_config_file.txt"

// generate documentation based on config file
$ doxygen "project_config.txt"
```

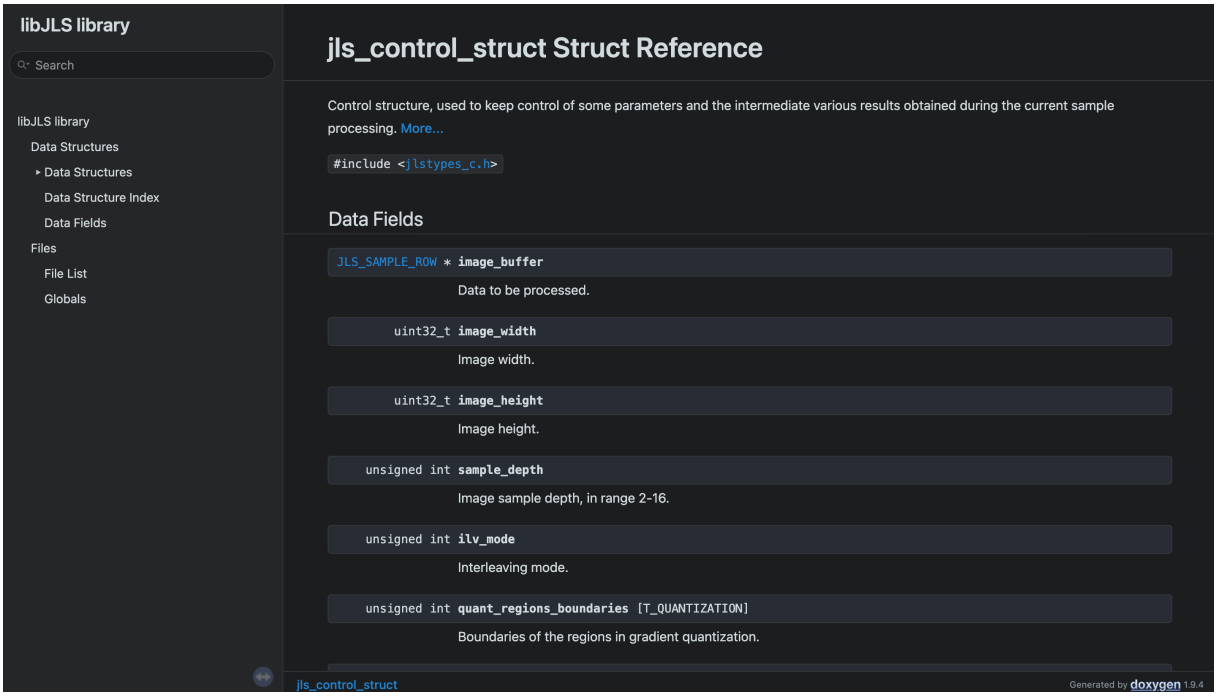

Figura 4.1: Documentación generada con Doxygen.

Capítulo 5

Evaluación de la biblioteca

En este capítulo explicamos los criterios tomados en cuenta para evaluar la biblioteca y validar el cumplimiento de los objetivos. Detallamos los procesos de testeo realizados y sus variaciones, herramientas utilizadas y la comparación con bibliotecas similares.

5.1. Aspectos considerados

Los factores que se toman en cuenta al momento de evaluar la biblioteca están alineados con los objetivos. De esta forma, se desprenden algunos aspectos especificados a continuación.

Corrección y cumplimiento de la norma
: Para la verificación de este punto se utiliza una implementación de referencia, que implementa la norma JPEG-LS y que comparte varias de las funcionalidades implementadas. Se detalla su comparación posteriormente.

Eficiencia computacional
: La validación de este punto se realiza en comparación con la implementación de referencia, y frente a bibliotecas de otros formatos.

Robustez
: A nivel de usuario, se cuenta con un módulo específico para el manejo de códigos de salida durante la ejecución, con un control detallado y códigos específicos de las diferentes situaciones.

Testeo con imágenes aleatorias
: Para el testeo de la librearía, se realizaron pruebas con una amplia variedad de imágenes de prueba. Estas imágenes fueron generadas de forma aleatoria variando sus características, como las dimensiones, largos de cadenas de píxeles iguales consecutivos (que provocan la entrada al modo *run*), o "suavidad" de la imagen.

5.2. Corrección y cumplimiento de la norma

En esta sección detallamos los aspectos considerados para validación de la corrección y cumplimiento de la norma JPEG-LS.

5.2.1. Generación de imágenes

La generación de imágenes aleatorias es un aspecto importante al momento de evaluar la liberaría, así se pueden tener suficientes entradas variadas para hacer una correcta validación de la implementación en diferentes escenarios, con imágenes de diferentes características como pueden ser la cantidad de canales, dimensiones, colores, etc.

Para generar imágenes de prueba utilizamos la herramienta *genImg*, que permite generar imágenes aleatorias sin compresión, en función de diversos parámetros. La ejecución de esta herramienta se realiza mediante la consola de comandos, y se encuentra disponible en el repositorio del código como parte de la solución.

Algunos de los parámetros que pueden ser modificados para la generación de imágenes aleatorias son:

- Dimensiones de la imagen.
- Profundidad.
- Suavidad de la imagen (rango de 0-10).
- Cantidad de sólidos (zonas lisas) en la imagen y sus dimensiones.
- Probabilidad de entrar en modo *run* y su largo.

5.2.2. Comparación con implementación de referencia

Con las funcionalidades propias de la biblioteca que fueron implementadas se puede validar la consistencia interna de los procesos de codificación y decodificación. Es decir, se codifica una imagen utilizando el compresor de la biblioteca, la salida generada vuelve a ser procesada pero esta vez por el decodificador, y finalmente se compara la salida del decodificador con la imagen original, las cuales deben coincidir. De todas formas, esto no asegura la conformidad con la norma JPEG-LS.

Para contemplar este último punto, utilizamos como referencia la implmentación desarrollada por Gadiel Seroussi. Esta fue la implementación usada por los responsables del diseño del algoritmo LOCO-I durante su desarrollo para hacer pruebas sobre del algoritmo. Dicha implementación fue utilizada como referencia para asegurar el cumplimiento de la norma JPEG-LS.

En este caso se agrega una validación al procedimiento detallado anteriormente en el esquema. Luego de codificar la imagen de entrada con la biblioteca

también se la codifica con la implementación de referencia, y se verifica que los archivos generados sean idénticos. Es decir, no solo se compara que la imagen de salida del decodificador sea igual a la entrada original, sino que también se valida que la imagen comprimida generada por la biblioteca sea igual a la imagen codificada por la implementación de referencia.

Existen algunas funcionalidades que no son soportadas por la implementación de referencia, por lo que esta no puede ser usada para validar el cumplimiento de la norma si se utilizan estas funcionalidades. Para estos casos verificamos la consistencia interna, realizando pruebas exhaustivas con diversas entradas.

5.2.3. Pruebas realizadas

La ejecución de pruebas para validar la correcta codificación y decodificación, así como también la conformidad frente a la norma de JPEG-LS, fueron realizadas a lo largo del proceso de desarrollo con cada nueva funcinalidad, así como también luego de tener la implementación completa.

En esta sección, mencionamos los tipos de pruebas y variaciones realizadas para asegurar el correcto funcionamiento de la herramienta.

Pruebas exhaustivas

La ejecución de pruebas exhaustivas fue realizada con una herramienta auxiliar, llamada *jls_exhaustive_test*, en conjunto con la herramienta de generación de imágenes aleatorias. Esta herramienta automatiza la ejecución de pruebas con diferentes parámetros de codificación, permitiendo alterar las combinaciones de funcionalidades activas en cada ejecución.

A continuación se listan los parámetros que se prueban de forma exhaustiva.

- Profundidad de las muestras.
- Componentes de la imagen.
- Dimensiones, incluyendo imágenes sobredimensionadas (de dimensiones mayores a 2^{16}, cuyos largos se representan con 3-4 bytes).
- Modos de intercalado.
- Procesamiento por líneas.
- Submuestreo, tablas de mapeo, y transformadas de color.
- Imágenes con canal alpha generado aleatoriamente.
- Redefinición de restart interval.

- Redefinición de los límites de cuantización de gradientes.
- Redefinición del valor máximo para las muestras.

Casos extremos

Se realizaron pruebas llevando ciertos parámetros a sus valores extremos. Un ejemplo es el de las dimensiones de la imagen, en donde los extremos son el valor 1 y 65535, valor máximo representable usando 2 bytes. Si se utilizan imágenes sobredimensionadas (de dimensiones mayores a 2^{16}), este límite superior pasa a ser 2^{24} o 2^{32}, utilizando 3 o 4 bytes para su representación respectivamente.

Otros casos extremos que fueron utilizados para las pruebas fueron:

- Imágenes de 1 píxel (1x1).
- Imágenes con 1 píxel en algunas de sus dimensiones.
- Imágenes con 65535 píxeles en alguna de sus dimensiones.
- Imágenes que comienzan en modo RUN.

5.3. Comparación con otras bibliotecas

Parte de la evaluación de la biblioteca consiste en su comparación respecto a las demás bibliotecas disponibles para compresión de imágenes sin pérdida. Por un lado se cuenta con la implementación de referencia de JPEG-LS, mientras que por otro lado se compara con las bibliotecas públicas de WebP y PNG. Debido a su característica de solo soportar imágenes de hasta 256 colores, decidimos no realizar comparaciones con GIF.

Las imágenes utilizadas para las pruebas comparativas fueron tomadas de un paquete de imágenes fotográficas de prueba disponible públicamente, armado específicamente para la evaluación e investigación de algoritmos de compresión [10].

Las entradas se pueden categorizar en tres grupos, de acuerdo a sus características.

- 8 bits en TC (tono continuo) - 12 imágenes.
- 16 bits en TC (tono continuo) - 12 imágenes.
- 8 bits generadas computacionalmente - 3 imágenes.

Para las comparaciones, tomamos en cuenta el tiempo de ejecución de codificación y decodificación, además de la tasa de compresión de cada biblioteca simplemente a modo ilustrativo, ya que esta tasa es una característica de la norma, no de la implementación que realizamos.

[10] Este paquete de imágenes se puede obtener desde el siguiente sitio: `http://www.imagecompression.info/test_images`

Cabe destacar que la biblioteca webP no soporta imágenes de 16 bits.

5.3.1. Ambiente de hardware para las pruebas

Las pruebas comparativas de rendimiento entre las bibliotecas se realizaron en un ambiente con las siguientes especificaciones.

- Sistema operativo: macOS Big Sur.
- Procesador: 2,5Ghz Intel Core i7.
- Memoria: 16 GB 1600Mhz DDR3.

5.3.2. Tiempos de ejecución

La comparación respecto a los tiempos de ejecución se realizó calculando exclusivamente el tiempo de procesamiento, ya sea codificación o decodidifación, sin tener en cuenta otras tareas realizadas durante la ejecución como inicialización, escritura de archivos, mensajes en consola, etc.

En las figuras 5.1 y 5.2 se muestran los tiempos promedios de codificación y decodificación respectivamente, para cada grupo de imágenes. Dado que cada imagen tiene distintas dimensiones y por tanto distinto tamaño, se normalizaron los valores expresándolos en megabytes por segundo (MB/s).

Analizando los resultados de la figura 5.1, se destaca a primera vista la dominancia de la implementación de referencia respecto a las demás, codificando una mayor cantidad de megabytes por segundo. Le sigue la implementación realizada por nosotros, con cerca del doble de velocidad respecto a PNG, y aproximadamente seis veces más rápido que WebP.

Por otro lado, en la figura 5.2 se analizan resultados opuestos a los anteriores. En términos de decodificación, tanto libPNG como webP son superiores por una gran diferencia frente al algoritmo de JPEG-LS. Esto era de esperar, dado que el algoritmo LOCO-I es relativamente simétrico en cuanto a tiempos de compresión y descompresión, y fue optimizado especialmente para el proceso de compresión, pensando en situaciones donde el algoritmo se ejecuta en dispositivos de capacidad computacional limitada, como ser una cámara digital. Por otro lado, los algoritmos de PNG y WebP tienen tiempos de decodificación mucho menores a los de codificación.

En el anexo se muestran los tiempos de codificación para cada una de las pruebas realizadas utilizando las diferentes bibliotecas.

5.3.3. Tasa de compresión

Más allá de que la tasa de compresión es una característica del algoritmo y no de la implementación, presentamos los resultados de tasas de compresión obtenidos

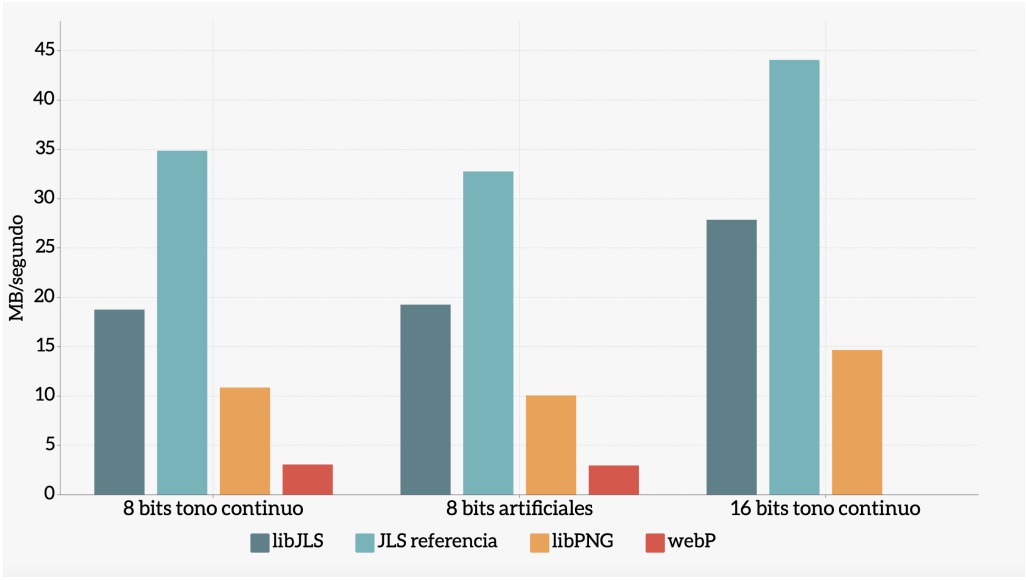

Figura 5.1: Velocidad de codificación en MB/segundo.

en las pruebas realizadas, simplemente a modo de comparar el desempeño del algoritmo frente al de PNG y WebP.

La tasa de compresión está expresada en bits por muestra. Para calcular la tasa de compresión se toma el tamaño de la imagen expresado en bits, y se lo divide por la cantidad de muestras, calculado como $altura * ancho * cantidad\_componentes$ de cada imagen. Los valores mostrados en la figura 5.3 son los promedios de la tasa de $bits/muestra$ de la imagen codificada codificado.

Como se puede apreciar en la figura 5.3, los resultados obtenidos para el conjunto de imágenes de prueba varían bastante dependiendo de la categoría de las imágenes. En el caso de imágenes de tono continuo de 8 bits, las tres bibliotecas se comportan practicamente igual. Siguiendo con imágenes de 8 bits artificiales (generadas computacionalmente), se ve una leve mejora en el desempeño de la biblioteca libJLS frente a las dos restantes, las cuales retornaron resultados similares. Finalmente, en el caso de 16 bits, nuevamente se refleja un mejor desempeño de libJLS frente a libPNG. Debido a su incompatibilidad con 16 bits de profundidad, webP queda excluída de este último análisis.

libJLS frente a la implementación de referencia

Dadas las diferencias obtenidas entre la implementación desarrollada y la implementación de referencia, es relevante comentar los posibles motivos de dichas diferencias y qué acciones se tomaron al respecto para minimizarlas.

Un aspecto en el que se diferencian ampliamente las implementaciones es en la arquitectura y distribución de los archivos dentro del proyecto. En la implemen-

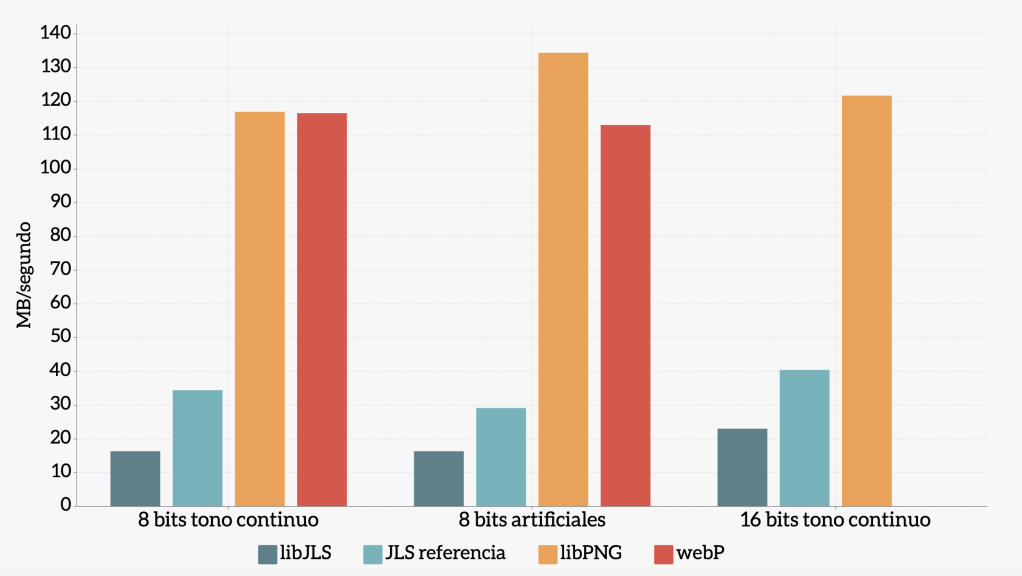

Figura 5.2: Velocidad de decodificación en MB/segundo.

tación desarrollada se agrupó, a nivel de estructura de los archivos y a nivel de proyectos durante la compilación, en tres categorías diferentes: common, encoder y decoder. Esta distinción se realizó con el fin de cumplir el objetivo de lograr un código modular y fácil de extender o modificar, pero tiene un impacto negativo en términos de rendimiento debido al funcionamiento del compilador durante el proceso de compilación. Esta estructura impide que el compilador pueda optimizar algunos llamados o definir funciones críticas como inline, y por tanto resulta en un aumento del sobrecosto por llamados entre funciones.

Realizando un análisis más detallado con herramientas de profiling [11], se llegó al resultado de que aproximadamente un 25 % del tiempo total de ejecución, se corresponde a sobrecosto en llamado a funciones. Debido a la baja modularidad de la implementación de referencia, el sobrecosto en llamado a funciones es considerablemente menor.

Para forzar a que el compilador inserte funciones *inline* en algunos lugares críticos se define la siguiente macro, que varía en función del compilador utilizado. De esta forma se logra que en tiempo de compilación, ese fragmento del código se inserte en el lugar donde es llamado en lugar de llamar al código objeto. Utilizando esta técnica se logró reducir aproximadamente un 10 % del tiempo de sobrecosto.

```
#ifdef COMPILER_GCC
    #define INLINE __attribute__((always_inline)) inline
#else
```

[11] Se utilizó el profiler incluido en Visual Studio, y la herramienta **Callgrind**.

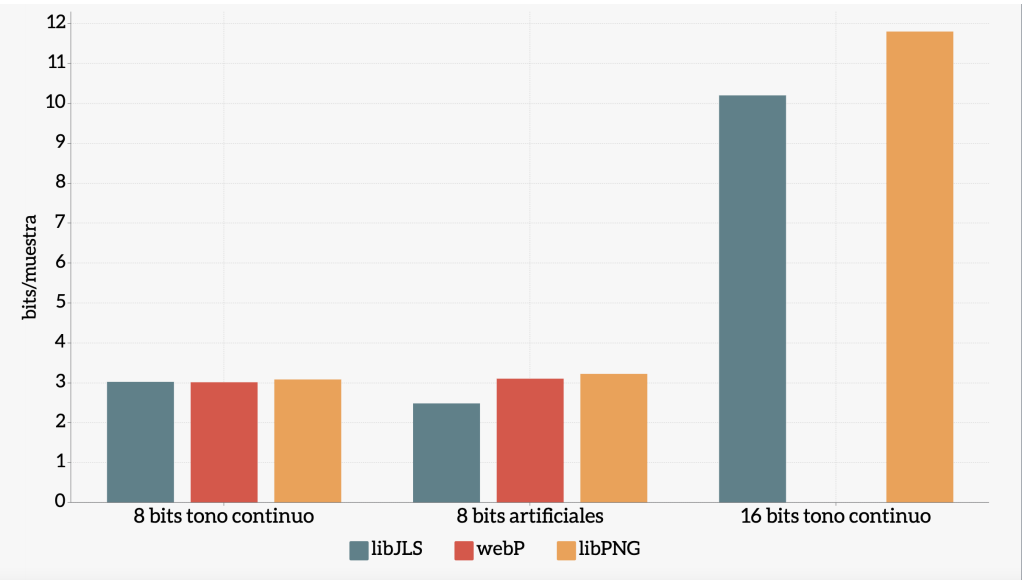

Figura 5.3: Tasas de compresión en bits/muestra.

```
        #define INLINE __forceinline
    #endif
```

libJLS frente a libPNG

Analizando los tiempos de procesamiento de la implementación de referencia con la de libPNG, se pueden ver resultados opuestos cuando se trata de la codificación y decodificación. Para la codificación, en términos numéricos libJLS presenta un rendimiento 2 veces mejor que la biblioteca libPNG, y en los tres grupos de imágenes se mantiene esa proporción. En el caso de la decodificación se invierten las proporciones, y en el caso de imágenes de tono continuo (ya sea 8 o 16 bits), libJLS presenta un rendimiento 3 veces menor que la biblioteca de libPNG. Para imágenes artificiales, esta proporción aumenta desfavorablemente para libJLS, presentando un rendimiento 4 veces menor que libPNG. Esto es esperable ya que JPEG-LS fue diseñado con la codificación de imágenes de tono continuo en mente. Esta diferencia entre los tiempos de compresión y descompresión de libPNG se debe a que, a diferencia de LOCO-I, el algoritmo utilizado por PNG difiere en gran medida entre el proceso de codificación y decodificación, obteniendo mucho mejores tiempos en el segundo caso.

Considerando la tasa de compresión, la biblioteca de libJLS presenta un mejor rendimiento que libPNG, aunque ambos son bastante similares. Cuando se trata de imágenes de 8 bits de tono continuo, el rendimiento es practicamente el mismo, y en las otras dos categorías la diferencia es aproximadamente un 15 %.

libJLS frente a webP

En este caso, el análisis se centrará en los casos de 8 bits, ya que la biblioteca de webP no brinda soporte a imágenes con 16 bits de profundidad. Con esta comparación ocurre algo similar a lo mencionado en la comparación anterior. Para la codificación, la biblioteca de webP presenta un rendimiento 6 veces menor que la biblioteca libJLS para los dos grupos de imágenes, mientras que en la decodificación este resultado se invierte e incluso aumenta, presentando libJLS un rendimiento 7 veces menor que webP.

En cuanto a la tasa de compresión, para imágenes de 8 bits de tono continuo practicamente no se presentan diferencias, mientras que para imágenes artificiales la biblioteca webP presenta tasas un 25 % inferiores a las de libJLS.

Capítulo 6

Trabajos futuros

Más allá de haber logrado los principales objetivos del proyecto, hay algunos aspectos que por distintas razones no se llegaron a resolver. A continuación presentaremos una breve lista de los elementos más trascendentes que quedaron pendientes.

6.1. Mejora de rendimiento

A pesar del esfuerzo invertido en mejorar el rendimiento de la biblioteca, no logramos alcanzar el objetivo de obtener tiempos de ejecución similares a la implementación de referencia. De todas formas, estos resultados son en gran medida consecuencia del balance entre modularidad y rendimiento que tuvimos que mantener durante el desarrollo. Más allá de que aún hay margen de mejora en este aspecto, manteniendo el nivel actual de modularidad y claridad del código, difícilmente los tiempos de ejecución se puedan aproximar a los de la implementación de referencia.

6.2. Near-lossless

Una característica de LOCO-I es que además de compresión sin pérdida, cualidad que lo identifica, también ofrece la alternativa de comprimir imágenes con pérdida. Esta funcionalidad no fue implementada principalmente debido a los límites de tiempo establecidos. Decidimos darle prioridad a la compresión sin pérdida, que es la principal fortaleza del formato JPEG-LS, como su propio nombre indica. De todas formas, la arquitectura de la solución tiene flexibilidad para la incorporación de nuevas funcionalidades, por lo que no implicaría un esfuerzo demasiado grande.

6.3. Múltiples frames

El formato JPEG-LS soporta la combinación de múltiples frames para formar animaciones, funcionalidad en la que brilla GIF y en la que es más popularmente utilizado. WebP también soporta animaciones, mientras que PNG por su lado ofrece una extensión (APNG) para soportar este tipo de imágenes. Al igual que con near-lossless, esta funcionalidad no fue implementada debido a que en el marco de tiempo dedicado al desarrollo de la biblioteca le dimos prioridad a la codificación de imágenes de un único frame, siendo estas las más comúnmente utilizadas.

6.4. Extensiones

En el año 2002 se publica una nueva norma [3] a modo de complemento de la norma original que define el formato JPEG-LS, donde se presentan una serie de extensiones con el objetivo de atacar algunos problemas detectados en la versión original, además de ofrecer algunas nuevas funcionalidades que se adaptan a las necesidades de los usuarios. De estas extensiones, en el proyecto se implementaron las *transformadas de color*, que permite la reconstrucción de una imagen a la cual se le aplicó una transformación sobre los píxeles para lograr una mejor decorrelación de los componentes del espacio de color.

Las demás extensiones, por distintos motivos, no fueron implementadas. La más importante de estas es la incorporación de la posibilidad de variar el método de codificación para aplicar codificación aritmética en lugar de codificación de Golomb. Esta variante permite lograr una mejor tasa de compresión, principalmente para imágenes artificiales, a cambio de un aumento moderado en la complejidad computacional del código. De todas formas, al igual que en el caso de near-lossless, la modularidad de la implementación facilita la incorporación de nuevas funcionalidades como esta.

Otras extensiones ofrecen pequeñas variaciones en la codificación de Golomb, la posibilidad de aplicar codificación de largo fijo y mejorar la predicción para imágenes con histogramas dispersos.

La implementación de estas extensiones fue relegada a un trabajo futuro con el objetivo de focalizarse en las funcionalidades principales del formato.

Por último, algunas extensiones ofrecen mejoras o variaciones sobre la compresión con pérdida, que no aplican para la biblioteca desarrollada al no ser implementada esta funcionalidad.

Capítulo 7

Conclusiones

El resultado de este proyecto es una biblioteca que ofrece todas las funcionalidades de compresión sin pérdida definidas en la norma base de JPEG-LS [2], a través de una implementación robusta, un código claro y bien documentado y una documentación completa del producto. Más allá de los aspectos mencionados en el capítulo anterior, los principales objetivos fueron logrados.

Invertimos un gran esfuerzo en mejorar el rendimiento del programa. Luego de varias iteraciones mejoramos considerablemente los tiempos de ejecución respecto a versiones iniciales de la implementación, pero aún así los resultados indican que queda margen para seguir mejorando. Como contraparte, el código está correctamente modularizado y mantiene un orden y una claridad que facilita el trabajo a futuro que se le pueda realizar. A su vez la documentación interna es clara y completa, por lo que sumado a lo anterior, implementar funcionalidades adicionales como las presentadas en [3] no debería resultar un problema mayor.

La biblioteca fue sometida a un testeo exhaustivo completo, combinando todas las funcionalidades posibles, en distintos sistemas operativos y con una gran variedad de imágenes, sin haber fallas detectadas. Esto, sumado al manejo de errores implementado, permite afirmar que se cumplió el objetivo en cuanto a robustez de la biblioteca desarrollada.

Logramos un muy buen nivel de documentación de la biblioteca, ofreciendo un manual de usuario completo, con código de ejemplo para todas las funcionalidades y las distintas posibilidades que ofrece la biblioteca. Sumado a lo anterior, se ofrece la documentación del código a través de la herramienta Doxygen que permite una visualización más cómoda de todo lo que ofrece la API. Por último, la documentación incluye también un README a modo de paneo general para un usuario que recién se descargó la biblioteca.

Definitivamente aún queda trabajo por hacer, pero en términos generales se obtuvo una biblioteca comparable con las ofrecidas por los formatos más populares en la industria.

Capítulo 8

Anexos

8.1. Manual de usuario

1. USER GUIDE FOR LIB-JLS

2. Overview

This user's guide gives an introduction to the JPEG-LS standard, mentioning its main features and components. This first section mentions aspects of the standard only, without going into implementation details. It also briefly explains some concepts used throughout the guide, and implementation restrictions against the standard.

After that, general aspects of the API are explained, detailing the data structures common to the encoder and decoder.

There are also separate sections for the simplified API, and for the Full API for encoding and decoding. Each of these has specific example codes.

> For a detailed usage example, see the code in the "doc/**example.c**" file.

3. JPEG-LS standard

3.1. JPEG-LS format

A JPEG-LS image consists of a sequence of markers and frames. The frames in turn are organized into smaller units. The file may optionally contain an exchange information header (e.g. JFIF), and such a header is usually defined outside the standard. However, the LIB-JLS library supports writing and reading the JFIF header, as detailed in JFIF section.

When stored as a file, the extension used for this file type is `.jls`.

The elements that compose an image of this type are described below.

- JFIF
- Frame
    - Scan
        - Restart Interval
        - MCU

3.1.1. JFIF (JPEG File Interchange Format)

The JFIF (JPEG File Interchange Format) is an image file format following the T.871 standard. It defines adicional specifications for the container format that contains the image data encoded with the JPEG algorithm.

As JPEG-LS images are part of the JPEG group of standards, the JFIF container also allows image data encoded with the JPEG-LS algorithm.

This extension is designed for an easy exchange of images across multiples platforms and applications, and defines some details that are not specified in the [JPEG standard]. i.e.: image orientation or a thumbnail image.

3.1.2. Frame

A frame is a representation of an image. A `.jls` file may contain multiple frames, and each frame can contain one or more scans.

3.1.3. Scan

A scan represents a portion of the data that contains one or more image components. In turn, a scan can be organized into smaller pieces described below.

3.1.4. Restart interval

With the restart intervals it is possible to divide each scan into independent segments where statistics are collected separately, i.e., the accumulated statistical parameters of the encoder/decoder are reset after each restart interval. Thus, a segment defined by reset intervals can be compressed and decompressed independently, without any dependence on other segments.

3.1.5. MCU

The MCU (Minimum Code Unit) is the smallest group of samples that can be coded. It depends on the interleaving mode, which is described later.

3.2. General concepts

This section gives a very brief introduction to the concepts used throughout the user's manual. For full detailed information, see T.87 (or T.870 in case of color transforms).

> **NOTE:** The terms *channels* and *components* are used interchangeably.

3.2.1. Markers

A marker is a two-byte word that has the format `0xFF` **`hh`** with the following restrictions:

- **`hh`** most significant bit must be `1`.
- **`hh`** cannot be equal to `0xFF`.

> e.g.: 0xFFFF is not a valid marker, but 0xFFD7 is.

There are two types of markers:

- Stand-alone: a two-byte marker by itself, without any associated data segment.
- Marker-segment: a two-byte marker with an associated data segment.
    - The two-byte marker identifier is followed by two bytes indicating the length of marker-specific payload data that follows.

NOTE: whenever the value `0xFF` is to be written as part of the encoded image, the encoder adds an extra zero bit, which allows the decoder to differentiate it from the beginning of a marker.

3.2.2. JPEG and JPEG-LS markers

The following is a list of JPEG markers inherited by JPEG-LS that are supported in this implementation.

	Description	**Value**
SOI	Start of image	`0xFFD8`
EOI	End of image	`0xFFD9`
SOS	Start of scan	`0xFFDA`
DNL	Define number of lines	`0xFFDC`
DRI	Define restart interval	`0xFFDD`
RSTm	Restart marker (FFD0-FFD7)	`0xFFD0`
COM	Comment	`0xFFFE`
APP0	JFIF marker	`0xFFE0`

> NOTE: SOI and EOI are stand-alone markers, all others are marker-segment.

The following is a list of supported JPEG-LS-specific markers.

	Description	Value
SOF	Start of JPEG-LS frame (*)	`0xFFF7`
LSE	JPEG-JLS extension marker	`0xFFF8`
LSE_PARAMS	Marker for redefining parameters	`0x01`
LSE_MAPTABLE	Marker for defining mapping tables	`0x02`
LSE_XMAPTABLE	Continuation marker for defining mapping tables	`0x03`
LSE_OVERSIZE_DIMS	Marker for defining oversized images	`0x04`
LSE_COLOR_TRANS	Marker for defining color transform	`0x0D`

> (*) The original JPEG standard also defines the SOF marker, but with a different value.

The LSE marker segment must be composed of the marker itself, followed by the length of the marker segment and data segment. The data segment is composed of an identifier whose possible values are in the previous table with the prefix '*LSE*', and the parameters that vary depending on each identifier.

3.2.3. Interleaving modes

The standard defines three modes of operation, which can be selectd by the user. The operation mode may affect the computation performance and the compression ratio.

The interleaving mode used is written on the scan header by the encoder, so that the decoder can know the used mode.

> For the description below, assume a three-component image (RGB).

Plane interleaved

All lines of the same component are processed before processing the next component.
The pixels are processed as follows:

```
RRR...RRR
⋮
RRR...RRR            // end of first channel

GGG...GGG
⋮
GGG...GGG            // end the second channel

BBB...BBB
⋮
BBB...BBB            // end the third channel
```

Line interleaved

Each line of every component is processed before the next image line.
The pixels are processed as follows:

```
RRR ... RRR
GGG ... GGG
BBB ... BBB                 // first image line

RRR ... RRR
GGG ... GGG
BBB ... BBB                // second image line

...

RRR ... RRR
GGG ... GGG
BBB ... BBB                // last image line
```

Sample interleaved

In this case the processing occurs at the sample level. One sample of every component is processed before moving on to the next pixel.
The pixels are processed as follows:

```
RGB RGB ... RGB RGB     // first line
RGB RGB ... RGB RGB     // second line

...

RGB RGB ... RGB RGB     // last line
```

3.2.4. Mapping tables

Mapping tables are used to support a procedure in which each sample value emitted by the decoder is interpreted as an index into a mapping table (also known as color palette) from where the actual sample value is reconstructed.

The encoding process receives the data mapped to a single component as input, representing the index of the pixel in the mapping table. The decoder performs the reverse process, using the decoded value and the mapping table to recover the original component values.

3.2.5. Color transform

Color transforms consist of applying a reversible operation to the components of each pixel, with the aim of reducing the correlation between components. To obtain the original components, the decoder must simply apply the inverse operation on the components of each pixel.

3.2.6. Subsampling

Subsampling is used when some components of an image are sampled with different rates (for example, each line contains twice as many green samples as red or blue samples). For such a case, it is allowed to explicitly define a vertical and an horizontal subsampling factor for each image component.

3.2.7. Specification of parameters

JPEG-LS handles different parameters that are specified in each marker segment, defined to correctly interpret the information contained in the image.

The standard defines the number of bytes used for each parameter and the allowed range for each one, which are detailed as follows.

> N_f is the number of components in frame.
> N_s is the number of components in Scan.

Frame header parameters

Parm	Size	Valid range
Frame header length	2 bytes	$8 + 3\ N_f$
Sample precision	1 byte	2 - 16
Image height	2 bytes	1 - 65 535
Image width	2 bytes	1 - 65 535
Number of components N_f	1 byte	1 - 4
Component identifier (*)	1 byte	0 - 255
Horizontal subsampling factor (*)	4 bits	1 - 4
Verticals subsampling factor (*)	4 bits	1 - 4
~~Quantization table destination selector~~ (*)	1 byte	0

Scan header parameters

Parm	Size	Valid range
Scan header length	2 bytes	$6 + 2 N_s$
Components in scan N_s	1 byte	1 - 4
Scan component selector (*)	1 byte	0 - 255 (#)
Mapping table id (*)	1 byte	0 - 255
Start of spectral selection (near)	1 byte	1 - 7
End of spectral selection (interleaved)	1 byte	
~~Successive approximations~~	1 byte	0

> (*) Data repeated for each of the components.
> ~~Strikethrough~~ fields are not used in JPEG-LS.
> (#) These value shall be member of the set of component identifiers specified in the frame header.

3.3. Implementation limitations

This section mentions restrictions of the implementation, which are not restrictions of the JPEG-LS standard itself.

Some of this restrictions refer to combination of parameter values. The following is a list of parameters that can be modified using the library.

- Dimensions (considering oversized).
- Components.
- Image depth.
- Mapping table, color transform and subsampling.
- Redefine quantization thresholds.
- Redefine maxval.
- Redefine reset.
- Image with alpha channel.
- Restart interval.
- Processing by lines.
- Comments.
- DNL marker.

In case the encoding process are started with an incompatible combination of parameters, an execution exit code indicating the error is returned. When decoding, the data received is validated to provide greater security, although if the JLS input data is generated correct, the decoder should not detect invalid parameters.

The following are the restrictions that must be taken into account in terms of parameters. Parameters that are not explicitly mentioned do not have any restrictions.

- Mapping table, color transform and subsampling are for exclusive use with each other, none can be used simultaneously.

- Mapping table cannot be used if the max value is not the default.
- Images with alpha channel cannot be processed if mapping tables or color transforms are applied.
- Subsampled images with alpha channel, must follow this restriction:
    - When subsampling is used and the image has an alpha component, the subsampling factor for the alpha channel cannot be greater than the largest factor defined for the other image components. This occurs because different values are initialized and calculated taking into account the factor of the original components, and the alpha channel is set separately.
    - The exit code returned in this case is `EXIT_ERROR_ALPHA_CHANNEL_SUBSAMP_FACTOR`.
- Color transform, processing by lines, and plane interleaved cannot be decoded.
    - In the conditions mentioned above, it is possible to encode an image but not to decode it. The restriction comes up because in order to apply the inverse color transform when decoding, the complete component is needed, but this is not possible in plane interleaved since it is decoding by lines and the complete component is not available.
    - The exit code returned in this case is `EXIT_ERROR_DECODE_LINES_PLANE_COLOR_TRANSFORM`.
- Mapping table and processing by lines cannot be decoded.
    - This restriction comes from the fact that when decoding by lines, the dimensions of the image are not known beforehand, and therefore the necessary memory cannot be allocated to return the buffer with the mapped samples.
    - The exit code returned in this case is `EXIT_ERROR_DECODE_LINES_MAPPING_TABLE`.
- Non default image depth is incompatible with mapping table and color transform.
- Mapping table is incompatible with comments and DNL marker.
- DNL marker and not decoding by lines.
    - If the encoded image uses the DNL marker, it cannot be decoded without line processing, because the image dimension is not known beforehand to allocate memory for the necessary structures.
    - The exit code returned in this case is `EXIT_ERROR_DECODE_DNL_WITHOUT_LINES_PROCESSING`.

4. General API description

All public structures, type definitions and macros, can be found in the `jlspublictypes_c.h` file.

The user is the one who uses the libraries. He has access to the library(s) and can compile them on his own. In turn, he has access to the `.h` file headers that define the APIs and contain only the structures and data that the user is authorized to access. The binaries of the libraries, and these public headers are what the user incorporates into his project.

4.1. Macro definition

There are some macro definitions that must be taken into account before using the library.

4.1.1. DEBUG_MODE

The `DEBUG_MODE` macro definition allows one to activate the debug mode, where different messages are printed during image processing.

```
#define DEBUG_MODE 0

#if DEBUG_MODE
  #define PRINT(a)
    if (print_condition)
    printf a
#else
  #define PRINT(a) (void)0
#endif
```

The implementation defines a function `PRINT`, which invokes `printf` in debug mode and is `void` otherwise.
In the former case, the invocation to printf depends on the global variable `print_condition`, used to turn messages on or off in specific code fragments.

4.1.2. Others

Other macro definitions that can be modified by the user are mentioned below.

- `BUFFER_SIZE`: defines the size (in bytes) of the buffer used internally, which may affect the computation performance. This value must be greater than zero.
- `FIRST_CHANNEL_ID`: defines the first channel-id value when writing frame header. The other channels identifiers are consecutive starting from this value. By default, its value is 1.

4.2. Structures

This section shows the main structures of interest used in the implementation.

4.2.1. Processing state

This structure is a data type opaque to the user, used to maintain the state of the encoding or decoding process. The user has library functions to initialize and release this structure, and is responsible for maintaining the state during processing, using it as a function parameter throughout each library function call.

The type that is public to the user is defined as a pointer to the internal data structure. In this way, the user can declare and use variables of this type, but access to its content is restricted.

4.2.2. Decoded image data

This structure is defined only in the decoder, and is used to return the metadata of the decoded image to the user. It contains the fields that are necessary to interpret the decoded image correctly.

The data type used in the decoder API is `jls_dec_img_data_struct_p`, which is defined as a pointer to `struct jls_dec_img_data`.

```
struct jls_dec_img_data
{
  /// Image width
  uint32_t image_width;
  /// Image height
  uint32_t image_height;
  /// Image sample depth, in range 2-16
  unsigned int sample_depth;
  /// Channels
  unsigned int channels;
  /// Subsampling horizontal factors
  uint8_t *subsampling_h;
  /// Subsampling vertical factors
  uint8_t *subsampling_v;
  /// Indicates if the image is indexed to a mapping table (should be mapped to obtain
the actual image)
  bool uses_mapping_table;
  /// JFIF parameters (NULL in case no JFIF data was read)
  struct jls_jfif_params *jfif_params;
  /// Every comment read
  struct jls_comments *comments;
  /// Reference to the buffer containing the decoded alpha channel (NULL if the decoded
image doesn't have alpha channel)
  JLS_SAMPLE *alpha_buffer;
};
```

This structure can be obtained through a function provided by the API, and is loaded from the state structure.

```
// Creates variable of type 'jls_dec_img_data_struct_p'
jls_dec_img_data_struct_p img_data;

// The struct is allocated and the metadata of the decoded image
// is loaded in the struct which must be passed by reference in the
// parameters
jls_get_decoded_image_data(s_struct_p, &img_data);

// Frees memory struct
jls_destroy_dec_img_data(img_data);
```

4.2.3. Mapping tables

The structure used to represent a mapping table consists of a table identifier, a field to indicate the size (in bytes) of each entry of the table (1 byte typically), and a pointer to an array with the table entries.

```
typedef uint8_t *TABLE_ENTRY;

struct jls_mapping_table
{
  // Table ID
  uint8_t tid;
  // Entry size (in bytes)
  uint8_t entry_size;
  // Actual table
  TABLE_ENTRY *table;
};
```

The library has the following function to use mapping tables.

```
void jls_load_mapping_table(jls_state_struct_p *s_struct_p, uint8_t tid, uint8_t
entry_size, TABLE_ENTRY *table);
```

4.2.4. Mapping tables

The structure used to represent a mapping table consists of a table identifier, a field to indicate the size (in bytes) of each entry of the table (1 byte typically), and a pointer to an array with the table entries.

```
typedef uint8_t *TABLE_ENTRY;
struct MappingTable
{
  // Table ID
  uint8_t tid;
  // Entry size (in bytes)
  uint8_t entry_size;
  // Actual table
  TABLE_ENTRY *table;
};
```

The library has the following function to use mapping tables.

```
void jls_load_mapping_table(jls_state_struct_p s_struct_p, uint8_t tid, uint8_t
entry_size, TABLE_ENTRY *table);
```

4.3. Initialization

Before starting the image processing, either for encoding or decoding, the state structure has to be initialized using the following functions:

```
// initializes the control for encoding
int jls_init_encode_struct(jls_state_struct_p *s_struct_p);

// initializes the control for decoding
int jls_init_decode_struct(jls_state_struct_p *s_struct_p);
```

With this functions, the state structures for encoding and decoding are initialized, respectively, allocating memory for internal usage, and setting default parameter values.

4.4. End of processing

The completion of the encoding and decoding process, respectively, is signaled by invoking the following functions.

```
void jls_finalize_encoder(jls_state_struct_p s_struct_p);

void jls_finalize_decoder(jls_state_struct_p s_struct_p);
```

These functions release memory allocated for internal usage, and in the case of the encoder, the `EOI` end-of-image marker is written.

The memory allocated for the state structure is freed by using the following function.

```
void jls_destroy_state_struct(jls_state_struct_p s_struct_p);
```

4.5. Exit codes

The library has a function to get the current exit code, and another function to get the detailed message. Most of the library functions return an exit code. Also, it is possible to obtain the last thrown exit code at any time during the execution using these functions.

```
// function header
int jls_get_exit_code(jls_state_struct_p s_struct_p);
char *jls_print_exit_code(int exit_code);
```

Code example:

```
int exit_code = jls_get_exit_code(s_struct_p);
char * message = jls_print_exit_code(exit_code);

message = jls_print_exit_code(EXIT_ERROR_INVALID_DIMENSIONS);
printf(message);

// Output:
// The specified value for the dimensions is not within the valid range.
```

The exit codes are categorized into certain numerical ranges depending on the type of error.

Code 0 is used for successful execution, and in the range of 1-9 are general errors.

Code	Value	Description
0	EXIT_OK	The execution finished successfully.
1	EXIT_ERROR_MEMORY_NOT_ALLOCATED	Memory could not be allocated.
2	EXIT_ERROR_INCOMPATIBLE_DEPTH_WITH_BUILD	The specified bit depth is not compatible with the project that was built.

In the range of 10-15 are the errors detected in I/O handling.

Code	Value	Description
10	EXIT_ERROR_OPENING_IO	EError opening I/O (stream or image).
11	EXIT_ERROR_READING_FILE	Error reading file, check pnm lib.
12	EXIT_ERROR_EMPTY_IO	Opened I/O is empty.
13	EXIT_ERROR_WRITING	Error writing I/O data, written bytes missed.

The errors detected in the headers, whether JFIF, frame or scan, are in the range of 30-40.

Code	Value	Description
30	EXIT_ERROR_INVALID_GRADIENT	The gradient is not valid.
31	EXIT_ERROR_INVALID_DIMENSIONS	The specified value for the dimensions is not within the valid range.
32	EXIT_ERROR_INVALID_COMPONENTS	The specified value for components is not within the valid range.
33	EXIT_ERROR_INVALID_SUBSAMPLING_F	The specified value for subsampling factors are not within the valid range.
34	EXIT_ERROR_INVALID_ILV_MODE	Invalid interleaved mode was selected. (0)PLANE (1)LINE (2)SAMPLE.
35	EXIT_ERROR_INVALID_BIT_DEPTH	The specified value for bit depth is not in the valid range.
36	EXIT_ERROR_INVALID_PROCESSING_MODE	The specified value for processing mode (1 encode, 2 decode) are not valid.
37	EXIT_ERROR_JFIF_HEADER	Error when reading JFIF header, APP0 marker found but was not JFIF.

In the range 50-70 are errors related to marker processing.

Code	Value	Description
50	EXIT_ERROR_UNRECOGNIZED_MARKER	An unrecognized marker was decoded.
51	EXIT_ERROR_MARKER_INVALID_DRI_MCU	The specified value for the number of MCUs in the DRI segment is not valid.
52	EXIT_ERROR_MARKER_INVALID_LOCATION	Marker found in a invalid location.
53	EXIT_ERROR_MARKER_DNL_EXPECTED	Image height was not set, DNL marker is required.
54	EXIT_WARNING_UNSUPPORTED_LSE_MARKER	An unsupported LSE marker was decoded. The content of the marker will be ignored.
55	EXIT_ERROR_MARKER_LSE_MAPPING_TABLE_ID	Mapping table continuation marker table ID does not match with the previous marker table ID.
56	EXIT_ERROR_MARKER_LSE_MAPPING_TABLE_SIZE	Mapping table continuation marker entry size does not match with the previous marker entry size.
57	EXIT_WARNING_MARKER_RST_CORRUPTED_DATA	Part of image data was lost. Decoding was skipped to the nearest RST marker.
58	EXIT_ERROR_MARKER_APP_NOT_SUPPORTED	APP marker different of JFIF not supported.
59	EXIT_ERROR_MARKER_LSE_INVALID_RANGE_VALUE	The specified value to redefine the range is smaller than the values that have already been decoded.
60	EXIT_ERROR_MARKER_DNL_INVALID_RANGE_VALUE	The specified value to number of lines in DNL marker is not in valid range.

Finally, in the range 80-90 are errors for incompatible features combination.

Code	Value	Description
80	EXIT_ERROR_INVALID_CONFIG	The parameter settings used are not compatible. See user guide for more details.
81	EXIT_ERROR_ALPHA_CHANNEL_SUBSAMP_FACTOR	The subsampling factor for the alpha channel cannot be greater than the greatest factor of the other components
82	EXIT_ERROR_DECODE_LINES_PLANE_COLOR_TRANSFORM	Line-by-line decoding is not allowed when using color transformations and PLANE interleaved.
83	EXIT_ERROR_DECODE_LINES_MAPPING_TABLE	Line-by-line decoding is not allowed when using mapping tables.
84	EXIT_ERROR_DECODE_DNL_WITHOUT_LINES_PROCESSING	Image height was no set. If the encoded image uses a DNL marker, it must be decoded by lines.

4.6. Alpha channel

The alpha channel is used for images with transparency. An important detail to mention is that the JPEG-LS standard does not contemplate the special treatment of transparency components, but it is a particularity of the implementation. The aspects taken into account in the implementation for the special treatment of this component, are explained below.

With this library, the alpha channel data is always written in a separate scan from the other components.

The library has a function to load the alpha channel.

```
/**
 * @brief Sets the buffer for alpha channel encoding.
 */
void jls_set_alpha_channel(jls_state_struct_p s_struct_p, JLS_SAMPLE *alpha_buffer);
```

Before start writing the frame header, a comment with the word `ALPHA` is written to indicate to the decoder that the encoded image has transparency. Note that this is outside the JPEG-LS standard, and it is specific to this implementation.

After the decoding process, if the processed image contains transparency, the alpha channel buffer will be available through the `jls_dec_img_data` struct as well (see decoded image data).

```
// (...) Initilization of the state struct and image buffer allocation

// Decodes the image
jls_decode(s_struct_p);

jls_dec_img_data_struct_p img_data;
jls_get_decoded_image_data(s_struct_p, &img_data);

if (img_data->alpha_buffer != NULL)
{
  // Access alpha channel through img_data->alpha_buffer
}
```

4.7. Color transform

The library implements most frequently used color transforms, and also allows the user to explicitly define a custom transform.

4.7.1. Predefined color transforms

The `JLS_COLOR_TRANSFORM` enum type represents the predefined color transformations that can be requested by the user. The color transforms are applied on the encoder side, which is responsible for indicating to the decoder the transformation that was used.

- **DrGDb**
  - R' = R - G
  - G' = G
  - B' = B - G
- **mRDgDb**
  - R'= R
  - G' = G - R
  - B' = G - B
- **mLDgEb**
  - Dg = R - G
  - L = R - ⌊Dg / 2⌋
  - Db = B - L
- **mLDgDb**
  - Dg = R - G
  - L = R - ⌊Dg / 2⌋
  - Db = G - B
- **mRCT**
  - R' = R - G
  - B' = B - G
  - G' = G + ⌊(R'+ B') / 4⌋

When encoding an image using color transforms, the encoding process does not automatically apply the transform of the user's choice. Instead the library provides a function to apply a color transform to an image buffer, which can be then encoded as usual.

```
// function header
void jls_apply_color_transform(jls_state_struct_p s_struct_p);
```

The transformation to be applied must be set beforehand using the following function.

```
// function header
void jls_set_preset_color_transform(jls_state_struct_p s_struct_p, JLS_COLOR_TRANSFORM
color_transform_id);
```

4.7.2. Custom color transform

A custom color transform is determined by two data structures, which define an inverse color transform to be applied by the decoder.

```
struct jls_inv_color_transform_data
{
  uint16_t max_trans; // Number in the range 0 <= MAXTRANS <= 2^p - 1
  uint8_t nt; // Number of components participating in the transform
  uint8_t *c_ids; // IDs of the nt components participating in the transform

  // The coefficients used to perform the inverse color transform of each component.
  // The order of this components is the order in which they are recovered.
  // E.g.: (R-G, G, B-G) is recovered by first calculating G = G, then R = R + G, then
B = B + G. So the coefficients are ordered (G, R, B).
  struct jls_inv_color_transform_comp_coeff *c_coefficients;
};

struct jls_inv_color_transform_comp_coeff
{
  uint8_t center; // 0 if the center is 0, 1 if the center is HALFTRANS (MAXTRANS/2)
  uint8_t norm; // Exponent used during the transformation operations

  // The coefficients used to perform the inverse color transform of this component
  // The order of this components is the order in which are recovered.
  // E.g.: (R-G, G, B-G) is recovered by first calculating G = G, then R = R + G, then
B = B + G. So the coefficients are ordered (G, R, B).
  uint16_t *coefficients;
};
```

The following library function is used to set a custom transformation before encoding.

```
// function header
void jls_set_custom_color_transform(jls_state_struct_p s_struct_p, struct
jls_inv_color_transform_data *color_transform);
```

4.8. Subsampling

The subsampling is set with the following library function.

```
// function header
void jls_set_subsampling_factors(jls_state_struct_p s_struct_p, uint8_t
*subs_factors_h, uint8_t *subs_factors_v);
```

The parameters `subs_factors_h` and `subs_factors_v` correspond to the horizontal and vertical subsampling factors for each component. Both parameters must consist of an array with the same number of elements as the number of components in the image. The subsampling factors can only be {1, 2, 4}.

Code example:

```
// image with 3 components
// horizontal subsampling factor: R:4 - G:1 - B:2
// vertical subsampling factor:   R:1 - G:2 - B:2
uint8_t subs_factors_h[3] = {4, 1, 2};
uint8_t subs_factors_v[3] = {1, 2, 2};

// assumes initialized state structure
jls_set_subsampling_factors(s_struct_p, subs_factors_h, subs_factors_v);
```

An image with this subsampling factors, horizontally, for each sample of the G component, has 4 samples of the R component and 2 of the B component. Vertically, for each sample of the R component, there are 2 samples of the G and B components.

4.9. Comments

Comments are character strings that can be written to the encoded image, without the encoder processing their values as if they were image data. This feature is inherited from the JPEG baseline standard, supported for encoding and decoding.

To represent the comments a linked list is defined, consisting of the `jls_comment` structure composed of a `char*` type field and a reference pointer to the next comment, and the `jls_comments` structure that has a reference pointer to the first and last comment, and a comment counter.

```
struct jls_comment
{
  char *comment;
  int length;
  struct jls_comment *next_comment;
};

struct jls_comments
{
  struct jls_comment *first_comment;
  struct jls_comment *last_comment;
  int comments_count;
};
```

The library has the following function writing comments.

```
// encoder function header (called in between MCUs encoding)
void jls_write_comment(jls_state_struct_p s_struct_p, char *comment, int
comment_length);
```

When this function is called, the `COM` marker is written following the specifications mentioned in the markers section. The value of the `COM` marker is written, followed by the length of the segment, which is equal to the length of the comment received plus the 2 bytes used to write this value, and finally the character string.

During the decoding process, each time a `COM` marker is read, it is processed and the comment is stored in the state structure and can be accessed by the user through the decoded image metadata (see decoded image data).

Code example:

```
// initialized state structures

// encoding
jls_write_comment(s_struct_p, "Start writing scan for component id 1", 37);
// ...
jls_write_comment(s_struct_p, "Metadata x1=v1", 14);

// decoding
jls_dec_img_data_struct_p img_data;
// get decoded image data and access to comments field
jls_get_decoded_image_data(s_struct_p, &img_data);

comment = img_data->comments->first_comment;
for (int i = 0; i < img_data->comments->comments_count; i++)
{
  printf("-> %s \n", comm->comment);
  comm = comm->next_comment;
}

// Output:
// -> Start writing scan for component id 1
// ...
// -> Metadata x1=v1
```

5. Simplified API

The simplified interface is designed for easy implementation of typical encoding and decoding use cases.

In the case of the encoder, a single function call allows the image to be encoded and saved to an output file. The only necessary parameters are the image buffer, image dimension, channels, the interleaving mode to be used, and the name of the output file.

```
// function header
int jls_encode(JLS_SAMPLE_ROW *image_buffer, const char *out_filename, uint32_t
image_width, uint32_t image_height, unsigned int ilv_mode, unsigned int channels);
```

Code example:

```
JLS_SAMPLE_ROW * image_buffer;

// (...)
// load the image buffer
// (...)

int exit_code = jls_encode(image_buffer, "encoded_image.jls", 1024, 720, SAMPLE_ILV,
3);

if (exit_code == EXIT_OK)
{
  // do something (or not)
}
```

In the case of the decoder, a little more work has to be done, as the buffer must be allocated previously depending on the header information of the image (image dimensions and channels). So the first step is to initialize the state struct, then indicate the file or stream to decode and read the header. With that information, now the buffer can be allocated, and then the rest of the image can be decoded.

Code example:

```
// Init the state struct
jls_state_struct_p s_struct_p;
jls_init_decode_struct(&s_struct_p);


// In case I/O data is get from file
if (data_form_file)
{
  // Indicate the .jls file to decode
  jls_set_io_file(s_struct_p, "compressed_example_filename.jls");
}
// In case I/O data is get from buffer
else
{
  jls_set_io_stream(s_struct_p, &io_buffer, stream_size);
}

// Reads the header information, needed to allocate the image buffer memory
jls_read_header(s_struct_p);

// Allocate memory for the image buffer
uint64_t buffer_size;
jls_get_buffer_size(s_struct_p, &buffer_size);
image_buffer_out = malloc(buffer_size);
jls_set_image_buffer(s_struct_p, image_buffer_out, buffer_ilv_mode, buffer_width);
```

```
// Decodes the image
int exit_code = jls_decode(s_struct_p);

if (exit_code == EXIT_OK)
{
  // do something with s_struct_p info
}

jls_destroy_state_struct(s_struct_p);
```

6. Full API

For fine control of all the encoding and decoding parameters, a full version of the API is available. It is explained throughout the user guide. This API allows for the encoding and decoding of images to process images either at once or in a line-by-line fashion for a reduced memory footprint.

For more detailed API documentation, see Doxygen documentation.

6.1. Full encoding API

Important features in the coding process are described, and code examples are given for each of them.

6.1.1. JFIF header

In the JFIF `APP0` marker segment some parameters of the image are specified. In this implementation a very basic read/write of this extension is performed, which allows for the specification of the density (refers to whether the ratio is 1 to 1 on the X and Y axis, pixel per inch, or pixel per centimeter), and the density according to the axis. During the decoding process, the JFIF data read can be accessed through the `jls_dec_img_data` structure.

The `jls_jfif_params` structure is used to group all the data included in this header.

```
struct jls_jfif_params
{
  uint8_t major_version;
  uint8_t minor_version;
  uint8_t density_unit;
  uint16_t x_density;
  uint16_t y_density;
};
```

To write the JFIF header when encoding, the library has a function to set the header parameters.

```
int jls_set_jfif_params(jls_state_struct_p s_struct_p, uint8_t major_version, uint8_t
minor_version, uint8_t density_unit, uint16_t x_density, uint16_t y_density);
```

The following is a example of how these functions should be used, assuming that the `jls_state_struct_p` structure has already been initialized.

```
// (...)
// major_version and minor_version = 1
// density_unit = 50
// x_density = 10, y_density = 20
jls_set_jfif_params(s_struct_p, 1, 1, 50, 10, 20);
```

On the decoder side, the information becomes available after decoding in the `jls_state_struct_p` structure, with no need for any specific action from the user. If the boolean field `jfif_marker` is set to true, the JFIF header information is available in the `jls_jfif_params` structure within the `jls_state_struct_p` structure.

6.1.2. Frame header

Each frame starts with a `SOF` marker (Start of frame) and a series of predefined parameters, each one with its own range of allowed values, and length in bytes for that value.

Calling the `jls_start_encoder` function starts the writing of the encoded data.

```
int jls_start_encoder(jls_state_struct_p s_struct_p);
```

First the SOI image start marker is written. Immediately after this, the JFIF header is written in case the user has activated it, as mentioned in the JFIF section.

Then the SOF marker is written with the parameters specified below:

- `Lf` Frame header length, written using two bytes.
- Sample precision.
- Number of lines and samples per line: in case of oversized dimensions, writes zero as the dimensions are written in a LSE marker after this.
- `Nf` Number of image components in frame.

After these parameters, specific data for each component is written.

- Component identifier
- Horizontal subsampling factor
- Vertical subsampling factor
- Quatization table (mapping table) destination selector T_{qi} (0 if no mapping table is used)

`LSE` markers are then written in case the user has activated any of the functionalities defined in this marker. See markers section.

To verify the range of allowed values and bytes used to write each of the parameters, see parameters section.

When decoding, the read of frame header is performed by the `jls_read_header` function, which reads the data from the image and set the values in the state struct (see processing state).

```
int jls_read_header(jls_state_struct_p s_struct_p);
```

6.1.3. Scan header

The scan header specifies the identifier of the components it contains.

Although the JPEG-LS standard has no restrictions on the interleaving modes and number of scans, in this implementation the number of scans is determined as follows, depending on the interleaved operation mode selected.

- Plane interleaved: one scan is written for each component.
- Line interleavead : a single scan is written with all components.
- Sample interleaved: a single scan is written with all components.

Whatever interleaving mode is chosen, if the image has an alpha channel, it is written in a separate scan.

For both encoding and decoding the functions return an exit code indicating the result of the execution.

```
int jls_start_encode_scan(jls_state_struct_p s_struct_p);
int jls_start_decode_scan(jls_state_struct_p s_struct_p);
```

6.1.4. Restart interval definition

The use of restart interval allows for partial recovery from decoding errors. If for some reason the decoding fails (e.g., due to a corrupted encoded stream), the missed portion of the image can be skipped until the next restart interval marker is found, where the encoding state is reset and decoding may resume from that point.

The library function provided for this purpose is `jls_set_restart_interval`. The first parameter corresponds to a pointer to the state structure, and the second parameter to the number of MCUs to be processed between each encoding state reset. This value is written to the `DRI` marker segment.

```
void jls_set_restart_interval(jls_state_struct_p s_struct_p, uint16_t
restart_interval);
```

6.1.5. Parameter settings

Before starting the encoding process, some fields corresponding to the image to be encoded must be configured. There are also encoding parameters, which allow the user to alter the default behavior of the compressor.

The fields that can be modified by the user are listed below.

- Image dimensions, width and height.

These fields are mandatory and must be set by the user. The image height may be omitted by passing the value 0, in which case the use of the `DNL` marker is assumed.

```
void jls_set_image_dimensions(jls_state_struct_p s_struct_p, uint32_t image_width,
uint32_t image_height);
```

- Image components.

The count of components in image must be set by the user.

```
void jls_set_channels_count(jls_state_struct_p s_struct_p, unsigned int channels);
```

- Interleaving modes.

The interleaving mode is represented by macros, with the following names and values.

Interleaving mode	Macro	Value
Plane interleaved	PLANE_ILV	0
Line interleaved	LINE_ILV	1
Sample interleaved	SAMPLE_ILV	2

The interleaving mode must be set by the user. By default, the `LINE_ILV` mode is used.

The library has the following function to set the interleaving mode, which must be called before starting the encoding process.

```
void jls_set_ilv_mode(jls_state_struct_p s_struct_p, unsigned int ilv_mode);
```

- Sample depth.

The image sample depth must be set by the user. Its default value is 8 or 16, depending on the mode the library was built.

```
void jls_set_sample_depth(jls_state_struct_p s_struct_p, unsigned int depth);
```

- Image buffer.

Setting the image buffer is mandatory. A pointer to the `JLS_SAMPLES` array is expected, which represents the image data. It will be interpreted as lines of length `line_width` organized in the `buffer_ilv` interleaving mode.

```
void jls_set_image_buffer(jls_state_struct_p s_struct_p, JLS_SAMPLE *image_buffer,
unsigned int buffer_ilv, uint64_t line_width);
```

- Restart interval MCUs.

Indicates after how many MCUs the Restart interval marker will be written. By default, this parameter is set to 0, which disables the use of restart intervals.

```
void jls_set_restart_interval(jls_state_struct_p s_struct_p, uint16_t
restart_interval);
```

- Reset redefinition.

Set the reset value of `N` at which encoding/decoding statistics are halved. The default value is 64.

```
void jls_set_reset(jls_state_struct_p s_struct_p, int reset);
```

- Maximum value redefinition.

This field holds the number of possible sample values. This default value is 256.

```
void jls_set_max_val(jls_state_struct_p s_struct_p, unsigned int max_val);
```

- Quantization thresholds redefinition
  Gives the option of redefining the quantization thresholds. The default values for `T1`, `T2` and `T3` are 3, 7 and 21 respectively.

```
void jls_set_quant_thresholds(jls_state_struct_p s_struct_p, unsigned int t1, unsigned
int t2, unsigned int t3);
```

- Color transform

```
void jls_set_preset_color_transform(jls_state_struct_p s_struct_p, JLS_COLOR_TRANSFORM
color_transform_id);

void jls_set_custom_color_transform(jls_state_struct_p s_struct_p, struct
jls_inv_color_transform_data *color_transform);
```

- Mapping table

```
void jls_load_mapping_table(jls_state_struct_p s_struct_p, uint8_t tid, uint8_t
entry_size, TABLE_ENTRY *table);
```

- Subsampling

```
void jls_set_subsampling_factors(jls_state_struct_p s_struct_p, uint8_t
*subs_factors_h, uint8_t *subs_factors_v);
```

- JFIF

```
void jls_set_jfif_params(jls_state_struct_p s_struct_p, struct jls_jfif_params
*jfif_params);
```

- Alpha channel

```
void jls_set_alpha_channel(jls_state_struct_p s_struct_p, JLS_SAMPLE *alpha_buffer,
uint32_t width, uint32_t height);
```

After having initialized the structure, the user is free to redefine any of the encoding parameters. Once the encoding process starts, parameters cannot be modified anymore.

Code example:

```
// Encoder
jls_state_struct_p s_struct_p;
jls_init_encode_struct(&s_struct_p);

jls_set_image_dimensions(s_struct_p, 1024, 720);
jls_set_ilv_mode(s_struct_p, LINE_ILV);
jls_set_quant_thresholds(s_struct_p, 2, 10, 18);

jls_encode_with_config(s_struct_p);
```

6.2. Full decoding API

For the decoder the API is simpler, since it obtains all the required information from the JPEG-LS encoded image, and does not require any configuration by the user.

Code example for the decoder use:

```
// Defines state struct
jls_state_struct_p s_struct_p;

// Starts decode process, for "encoded.jls" image
jls_init_decode_struct(&s_struct_p);
jls_set_jls_file(s_struct_p, "../images/encoded.jls");
jls_read_header(s_struct_p);

// Allocates memory for output
uint64_t buffer_size;
jls_get_buffer_size(s_struct_p, &buffer size);
image_buffer_out = malloc(buffer_size);
if (image_buffer_out == NULL)
{
  return EXIT_ERROR_MEMORY_NOT_ALLOCATED;
```

```
}

// Set reference for allocated memory to state struct
jls_set_image_buffer(s_struct_p, image_buffer_out, buffer_ilv_mode, buffer_width);

// Decodes the "encoded.jls" image
jls_decode(s_struct_p);

printf("Execution result: %s \n\n",
jls_print_exit_code(jls_get_exit_code(s_struct_p)));


// Gets decoded image data
jls_dec_img_data_struct_p img_data;
jls_get_decoded_image_data(s_struct_p, &img_data);

// Use img_data fields
//    img_data->image_width
//    img_data->image_height
//    img_data->channels
//    (...)

// Gets comments from img_data
struct jls_comment *current_comment = img_data->comments->first_comment;

// Frees memory
jls_destroy_state_struct(s_struct_p);
jls_destroy_dec_img_data(img_data);
```

7. Compilation mode

There are some macro definitions that must be taken into account before compiling the library.

7.1. TWO_BYTES_SAMPLE

The `TWO_BYTES_SAMPLE` macro allows one to compile the library for images of up to 8 bits per pixel dimension, or up to 16 bits.
By default the macro is not defined and compiles for images of up to 8 bits. To compile it for images up to 16 bits the macro must be defined in `jlspublictypes_c.h`.

Code example:

```
// build for 8-bits
#undef TWO_BYTES_SAMPLE

// build for 16-bits
#define TWO_BYTES_SAMPLE
```

The two-bytes-per-sample mode also works for 2-8 bits-depth images, so this compilation mode can be used to process images of any depth.
The one-byte-per-sample mode only works for 2-8 bits-depth images, but with better performance than the two-bytes mode.

8.2. Comparación - Tiempos de compresión

(segundos)	**libJLS**	**JLS referencia**	**webP**	**libPNG**
artificial.ppm	0,34	0,16	3,24	0,53
bridge.ppm	2,25	1,24	6,70	2,20
building.ppm	7,88	4,30	37,98	9,71
cathedral.ppm	1,17	0,68	5,86	2,89
deer.ppm	2,19	1,14	7,09	1,86
fireworks.ppm	0,84	0,49	7,92	2,50
flower.ppm	0,62	0,37	4,62	1,74
hdr.ppm	1,23	0,66	11,11	3,81
leaves.ppm	1,21	0,68	6,84	2,32
nightshot.ppm	1,30	0,77	13,26	3,95
spider.ppm	2,26	1,16	18,18	6,87
tree.ppm	5,42	3,39	19,61	6,75
artificial_longRun.ppm	0,59	0,36	3,91	1,22
artificial_smoth.ppm	2,36	1,28	23,45	7,53
artificial_solids.ppm	1,08	0,65	5,43	1,54
artificial16.ppm	0,88	0,48	3,36	1,70
bridge16.ppm	2,54	1,67	7,46	3,63
building16.ppm	9,01	5,98	40,55	13,61
cathedral16.ppm	1,34	0,88	6,69	2,52
deer16.ppm	2,43	1,68	7,04	2,99
fireworks16.ppm	1,21	0,76	7,92	4,99
flower16.ppm	0,69	0,43	4,60	2,04
hdr.ppm	1,39	0,87	11,43	2,61
leaves16.ppm	1,40	0,91	6,28	2,07
nightshot16.ppm	1,55	0,96	13,38	4,45
spider16.ppm	2,62	1,67	17,93	5,49
tree16	6,32	4,20	20,62	9,41

8.3. Comparación - Tasas de compresión

	libJLS	**webP**	**libPNG**
artificial.ppm	90.32 %	94.59 %	90.80 %
building.ppm	52.36 %	53.86 %	44.37 %
tree.ppm	46.13 %	44.62 %	41.89 %
bridge.ppm	43.40 %	41.93 %	40.68 %
cathedral.ppm	52.82 %	53.13 %	47,96 %
deer.ppm	34.71 %	35.11 %	33.65 %
fireworks.ppm	80.99 %	78.25 %	73.42 %
flower.ppm	74.61 %	75.95 %	68.24 %
hdr.ppm	69.28 %	68.71 %	62.31 %
leaves.ppm	51.03 %	52.75 %	43.63 %
nightshot.ppm	72.56 %	70.78 %	64.31 %
spider.ppm	77.33 %	76.70 %	70.29 %
artificial16.ppm	76.93 %	97.29 %	79.58 %
building16.ppm	25.86 %	76.93 %	14.92 %
tree16.ppm	23.17 %	72.31 %	13.04 %
bridge16.ppm	21.51 %	70.96 %	11.63 %
cathedral16.ppm	27.46 %	76.57 %	17.88 %
deer16.ppm	16.48 %	67.56 %	9.08 %
fireworks16.ppm	57.95 %	89.12 %	51.29 %
flower16.ppm	41.99 %	87.97 %	27.08 %
hdr.ppm	34.86 %	84.36 %	21.59 %
leaves16.ppm	25.46 %	76.38 %	15.24 %
nightshot16.ppm	40.97 %	85.39 %	26.50 %
spider16.ppm	41.24 %	88.35 %	26.45 %
artificial_smoth.ppm	84.98 %	73.21 %	69.45 %
artificial_solids.ppm	67.63 %	66.37 %	66.31 %
artificial_longRun.ppm	54.08 %	43.71 %	43.14 %

Bibliografía

[1] G. Sapiro M. J. Weinberger G. Seroussi. "LOCO-I: A Low Complexity, Context-Based, Lossless Image Compression Algorithm". En: *Proc. 1996 Data Compression Conference* (Marzo 1996), págs. 140-149.

[2] *Information technology - Lossless and near-lossless compression of continuous-tone still images - Baseline*. ISO/IEC 14495-1 ITU Recommend. T.87. Junio 1998.

[3] *Information technology - Lossless and near-lossless compression of continuous-tone still images - Extensions*. ISO/IEC 14495-2 ITU Recommend. T.870. Marzo 2002.

[4] *Information technology — Computer graphics and image processing — Portable Network Graphics (PNG): Functional specification*. ISO/IEC 15948. Marzo 2004.

[5] *Information technology - Digital compression and coding of continuous-tone still images - Requirements and guidelines*. ISO/IEC 10918-1 ITU Recommend. T.81. Septiembre 1992.

[6] J. A. Thomas T. M. Cover. *Elements of Information Theory*. 2.ª ed. Wiley-Interscience, John Wiley & Sons, Inc, 2006.

[7] D. Huffman. "A method for the construction of minimum redundancy codes". En: *Proc. IRE* 40 (Septiembre 1952), págs. 1098-1101.

[8] J. Rissanen. "Generalized Kraft inequality and arithmetic coding". En: *IBM J. Res. Develop.* 20.3 (Mayo 1976), págs. 198-203.

[9] S. W. Golomb. "Run-length encodings". En: *IEEE Transactions on Information Theory* IT-12 (Julio 1966), págs. 399-401.

[10] A. Lempel J. Ziv. "A Universal Algorithm for Sequential Data Compression". En: *IEEE Transactions on Information Theory* IT-23.3 (Mayo 1977), págs. 337-343.

[11] T. Welch. *A Technique for High-Performance Data Compression*. IEEE Computer, Junio 1984.

[12] G. Sapiro M. J. Weinberger G. Seroussi. “The LOCO-I Lossless Image Compression Algorithm: Principles and Standarization into JPEG-LS”. En: *IEEE Transactions on Information Theory* 9.8 (Agosto 2000), págs. 1309-1324.

[13] F. Ono R. Ohnishi Y. Ueno. “The efficient coding scheme for binary sources”. En: *IECE Jpn* 60-A (Diciembre 1977), págs. 1114-1121.

[14] *Information technology - Digital compression and coding of continuous-tone still images: JPEG File Interchange Format (JFIF).* ISO/IEC 10918-5 ITU Recommend. T.871. Mayo 2011.