Sistema esteganogràfic per imatges JPEG2000

Memòria del Projecte Fi de Carrera
d'Enginyeria en Informàtica
realitzat per

Álex Torres Rodríguez

i dirigit per

Josep Rifà Coma

Bellaterra, 18 de septembre de 2008

Universitat Autònoma de Barcelona

etse

Escola Tècnica Superior d'Enginyeria

El sotasignat, Josep Rifà Coma

Professor/a de l'Escola Tècnica Superior d'Enginyeria de la UAB,

CERTIFICA:

Que el treball a què correspon aquesta memòria ha estat realitzat sota la seva direcció per en

Álex Torres Rodríguez

I per tal que consti firma la present.

Signat: .....

Bellaterra, 18 de septembre de 2008

Agradecimientos

En primer lugar me gustaría agradecer a mi director de proyecto, Josep Rifà Coma, por la paciencia y el apoyo que me ha brindado durante todo el proyecto. También agradecer a mis padres y mi hermano por aguantarme en casa y haberme aportado un entorno agradable desde mi infancia hasta la fecha. A mis compañeros de la universidad, ya sean FHF, FMF, MHM o personas normales. Han sido muchos años estudiando juntos en la biblioteca, comiendo en ciencias y asistiendo a clase en general. Estoy seguro de que si no hubieran estado ahí, no habría sido lo mismo. A mis amigos de GBU por tantos buenos ratos, campamentos y otras muchas cosas que ocuparían demasiado y por tanto no escribiré. A María, por estar ahí y preocuparse por mi y el proyecto, por los ánimos dados y su amistad. Y por último al Hacedor, porque sin él nada de todo esto habría ni estaría pasando. A todos los mencionados y a los que probablemente me haya dejado, gracias.

Álex

"The most important thing in communication is to hear what isn't being said."

Peter Ferdinand Drucker

Índice

1.- La Esteganografía

La Esteganografía es la ciencia de las comunicaciones ocultas. Se trata de escribir un mensaje escondido de tal manera que nadie, además del emisor y el receptor, pueda darse cuenta de que dicho mensaje existe. A diferencia de la criptografía, que hace el mensaje ilegible y por tanto consigue que cualquiera que intercepte el mensaje no pueda comprenderlo, la esteganografía esconde el hecho de que existe un mensaje.

La ventaja de la esteganografía sobre la criptografía es que los mensajes no atraen atención sobre ellos mismos, el emisor o el receptor. Un mensaje que no está escondido, conllevará sospechas y, sin importar lo irrompible que sea su cifrado, puede ser incriminatorio en sí mismo, ya que hay países en los que la criptografía es ilegal. Con todo, esteganografía y criptografía suelen usarse juntas para asegurar la seguridad del mensaje escondido.

Los componentes principales de un mensaje esteganográfico son los siguientes:

- El Emisor: es la persona o entidad que envía el mensaje.
- El Receptor: es la persona o entidad que recibe el mensaje.
- El Mensaje: es la información que el emisor quiere hacer llegar al receptor.
- El Canal o Portador del Mensaje: es el elemento donde el mensaje será incrustado.
- El Objeto Esteganográfico: es el resultado de incrustar el mensaje en el objeto portador del mensaje. Aparentemente es igual que el objeto portador, pero no lo es ya que contiene el mensaje.
- El Método Esteganográfico: es la forma en que el mensaje es incrustado en el portador del mensaje. Sólo el el emisor y el receptor lo conocen y por tanto son los únicos que pueden recuperar el mensaje.

1.1.- Historia de la Esteganografía

La palabra esteganografía proviene del griego y significa "escritura escondida o encubierta" (*steganos*, que significa encubierto y *graphos*, que significa escritura). Su origen se encuentra en el año 440 AC. Heródoto menciona dos ejemplos de esteganografía en *Las Historias de Heródoto*:

Demarato, un ciudadano griego que vivía en Persia tras haber sido desterrado de Grecia, fue testigo de que Xerxes, el rey Persa, estaba construyendo una de las mayores flotas navales que el mundo había conocido. Xerxes iba a usar esta flota para atacar a Grecia por sorpresa. Demarato, que aún amaba su tierra natal, decidió avisar a Grecia, pero sabía que sería difícil enviar un mensaje sin que fuera interceptado. Fue entonces cuando se le ocurrió la idea de usar una tabla de cera para esconder su mensaje, ya que sabía que una tabla de cera en blanco se podría enviar a Grecia sin levantar sospechas, debido a que eran una forma de escritura común en aquel tiempo. Así pues, lo que hizo fue quitar la cera de una de las tablas, escribir el mensaje en la madera y cuando hubo acabado recubrió la tabla de madera con cera. Con la tabla de madera cubierta por la cera, el mensaje quedó cubierto también y la tabla parecía una tabla de cera normal. El mensaje escondido nunca fue descubierto por los persas y llegó a su destino en Grecia. Gracias a este mensaje, Grecia pudo derrotar a la flota invasora persa.

La otra historia es la de Histiaeo, que afeitó la cabeza de su esclavo más leal y le tatuó un mensaje en ella. Cuando el pelo le creció del nuevo y el mensaje estaba escondido lo envió a su destino. El objetivo era instigar una revuelta contra los Persas.

En la antigua china también desarrollaron sus propias técnicas esteganográficas. Escribían el mensaje en una pieza de seda. Cuando el mensaje estaba escrito en su totalidad cogían la pieza de seda, hacían una pequeña bola con ella y la cubrían de cera. Una vez finalizado ese proceso, el mensajero se tragaba la bola. La forma en que se recuperaba la bola cuando el mensajero llegaba a su destino queda para la imaginación de cada uno, porque no está documentado.

En 1499, Johannes Trithemius publicó el libro *Steganographia*, que era un tratado acerca de criptografía y esteganografía disfrazado como un libro de magia negra. Este libro fue el que prestó su nombre al campo moderno que nos ocupa.

En el siglo XVI el científico italiano Giovanni Porta desarrolló una forma de esconder el mensaje dentro de un huevo duro. Porta descubrió que había una manera de escribir en la cáscara de un huevo duro, de forma que la tinta fuera invisible fuera del huevo, pero visible en la parte interior del huevo. Esta tinta se hacía mezclando una onza de alumbre y una pinta de vinagre. La tinta traspasaría la cáscara del huevo debido a la naturaleza porosa de la misma y luego se volvería transparente, pero no sin antes escribir el mensaje en la parte interior blanca del huevo. Debido a que no había ninguna diferencia en el exterior, la única forma de encontrar y leer el mensaje era quitarle la cáscara al huevo.

Otro método usado históricamente son las tintas invisibles. Dichas tintas son substancias usadas para escribir, las cuales son invisibles al aplicarlas o al poco tiempo de hacerlo y que posteriormente se pueden hacer visibles de algún modo. Lo que suele hacerse es escribir en una hoja en blanco con la tinta invisible y luego escribir un texto cualquiera, ya que enviar una hoja en blanco puede levantar sospechas. También se puede escribir primero el texto y después, en los espacios en blanco, usar la tinta invisible para escribir el mensaje que queremos hacer llegar. Una vez el mensaje ha sido enviado, el receptor debe hacer visible el mensaje secreto, lo cual puede ser llevado a cabo de múltiples formas, dependiendo de la tinta usada: por luz ultravioleta, con calor, usando una reacción química, etc. Las tintas invisibles han sido usadas mayormente en espionaje.

La guerra siempre ha sido una motivación para la invención de técnicas de comunicación secreta. Una de ellas fue el sistema de la rejilla. Esta técnica trataba de escoger letras estratégicamente emplazadas en un texto normal y corriente. El mensaje se mandaba y el receptor podía ver el mensaje secreto usando una rejilla especial. La rejilla era sólo un trozo de madera que se pondría encima del texto y tenía los agujeros puestos donde se encontraban las letras que formaban el mensaje secreto.

La Segunda Guerra Mundial trajo consigo dos métodos esteganográficos más. La primera fue la invención de la tecnología de microfims. Fue inventada por los alemanes para enviar mensajes secretos a sus aliados. J. Edgar Hoover, el director del FBI del momento, se refirió a ella como la "obra maestra del espionaje enemigo". Dijo esto tras darse cuenta de lo mucho que los alemanes estaban usando esta tecnología y de la cantidad de información que un microfilm podía contener. Un microfilm es básicamente una imagen empequeñecida de tal forma que podría caber en un símbolo de puntuación o en el punto de una i. Podías meter una página de texto entera o una imagen completa en un microfilm, lo cual hacía que el microfilm fuera tan útil. La imagen se podía ver de nuevo ampliando el microfilm o usando un microscopio. El único inconveniente de de los microfilms es que, como aun eran una película fotográfica, brillaban si les daba la luz de una determinada manera y se podían diferenciar del resto del papel, que era mate.

La otra técnica perfeccionada en la Segunda Guerra Mundial fue el uso de null ciphers. Es una técnica similar a la técnica de rejilla, pero en este caso no se necesita una rejilla de madera para ver las letras especiales. En estos mensajes, ciertas letras de cada palabra se usaban para formar el mensaje secreto. En un null cipher se usaban textos normales para esconder el mensaje y debido a eso, normalmente pasaban las revisiones de seguridad. Lo que sigue es un ejemplo que se usa normalmente para explicar este método. Se cree que el siguiente mensaje fue enviado por un espía alemán durante la Segunda Guerra Mundial: "*Apparently neutral's protest is thoroughly discounted and ignored. Isman hard hit. Blockade issue affects pretext for embargo on by-products, ejecting suets and vegetable oils.*" Si cogemos la segunda letra de cada palabra, nos encontramos el siguiente mensaje: "*Pershing sails from NY June 1*".

1.2.- El Dilema de los Prisioneros

El problema del que trata la esteganografía también se conoce como El Dilema de los Prisioneros, formulado por Gustavus J. Simmons[1] en 1983.

En dicha situación Alice y Bob han sido arrestados y van a ser encerrados en celdas separadas. La única forma que tendrán de comunicarse será a través de mensajes que serán supervisados por el alcaide de la prisión: Walter.

Walter dejará a los prisioneros comunicarse con la esperanza de poder engañar a uno de los dos con algún mensaje que él mismo envíe (sólo en el caso del alcaide activo). Sin embargo, como sospecha de que Alice y Bob querrán coordinar algún tipo de plan de huida, los mensajes que podrán intercambiarse deberán ser totalmente comprensibles para él, por tanto no puede haber ningún tipo de mensaje cifrado.

Los prisioneros aceptan las condiciones para poder comunicarse a riesgo de ser engañados por el alcaide (una vez más, sólo en el caso del alcaide activo). Pero debido a que tienen que establecer el plan de huida, deberán establecer una forma de comunicarse de manera secreta, a través de un canal subliminal, ya que Walter monitoreará dichos mensajes. Además, como son conscientes de que el alcaide querrá engañarlos introduciendo mensajes fraudulentos, sólo intercambiarán mensajes si se les permite autenticarlos.

Debido a que el alcaide siempre tiene acceso a los mensajes, la única forma para Alice y Bob de saber que un mensaje es auténtico es introduciendo información redundante, la cual ha sido previamente acordada, en los mensajes. Así pues, cuando el receptor interprete dicha información sabrá que el mensaje es genuino. Si, en cambio, esa información no existe, querrá decir que el mensaje no lo es.

El alcaide está satisfecho ya que no hay nada en el mensaje que le pueda hacer sospechar que existe un canal subliminal entre los dos prisioneros. Es trabajo de los prisioneros establecer este canal subliminal, modificando el mensaje de manera totalmente inocua, para que Walter no sospeche, pero de alguna forma previamente acordada por ellos para poder escaparse de la prisión. Y esa es la característica más importante: la diferencia entre el objeto original y el objeto modificado ha de ser estadísticamente indetectable, lo que quiere decir esto es que Walter no debe detectar ningún tipo de diferencia.

Como ya se ha ido mencionando antes, se pueden diferenciar dos casos en esta situación:

- Alcaide pasivo: Walter se limita a monitorear y supervisar que en los mensajes no haya ningún tipo de información sospechosa.
- Alcaide activo: Además de supervisar los mensajes, en algún momento puede llegar a cambiar parte o todo el mensaje y hacerse pasar por uno de los dos prisioneros.

1.3.- La Esteganografía digital

La esteganografía digital se basa en dos principios:

- El primero es que los ficheros que contienen imágenes o sonido digitalizados pueden ser alterados de alguna forma sin que pierdan su funcionalidad.
- El otro principio trata sobre la imposibilidad de los seres humanos de distinguir pequeños cambios en el color de las imágenes o la calidad del sonido, lo cual hace posible el uso de objetos que contienen información redundante, como por ejemplo un archivo de sonido de 16 bits o una imagen de 24 bits.

También intenta, como la esteganografía clásica, mantener el método y el objeto portador en secreto. Para que la esteganografía permanezca sin detectar, el objeto portador debe mantenerse en secreto, ya que si muestra y se lo compara con el objeto esteganográfico, podría demostrarse fácilmente que ha sido modificado de alguna forma y por tanto, con el tiempo, revelar el mensaje escondido.

La teoría de la información nos permite ser más específicos cuando queremos definir que un sistema esteganográfico sea perfectamente seguro. Christian Cachin [2] propuso un modelo de teoría de la información para esteganografía que considera la seguridad de sistemas esteganográficos contra alcaides pasivos. En este modelo se asume que el adversario conoce el sistema de codificación pero no conoce la clave secreta. Su trabajo es crear un modelo de la distribución de la probabilidad de todos los posibles portadores y todos los posibles objetos esteganográficos resultantes. Una vez creado debe usar teoría de detección de señales para decidirse entre dos hipótesis: que el mensaje contenga información escondida o que el mensaje no contenga información escondida. Un sistema es perfectamente seguro si no existe ninguna regla de decisión que permita escoger una de las dos hipótesis mejor que una elección al azar.

En esteganografía digital se pueden usar muchos tipos de portadores, pero los más utilizados son imágenes, audio y vídeo.

Las imágenes digitales son candidatas excelentes para ser convertidas en portadoras del mensaje, ya que los bits de dicho mensaje secreto se pueden insertar como ruido en los bits empleados para codificar la imagen. Hay varias formas de codificar imágenes digitales por tanto veremos algunos casos.

- **GIFs & BMPs**: En estas imágenes se codifica cada pixel de la imagen con el indice de una tabla que muestra la representación de los colores que se emplean en la imagen. Es decir, que el valor de cada pixel es un índice a dicha tabla, no el color en sí mismo. Por tanto, si intentamos insertar el mensaje modificando los valores de esos pixels, la imagen resultante podría ser muy diferente de la original.

  La solución propuesta para combatir esto consta de tres partes: modificar la paleta de colores para que tenga la mitad de colores; poner entre dos colores A y B, adyacentes en la paleta, un color A' que sea cromáticamente muy cercano a A y, por último, incrustar el mensaje secreto en el bit menos significativo de cada índice.

  El precio que se paga con esta solución es tener la mitad de los colores en la paleta, pero la parte buena es que la densidad de información escondida es bastante alta, 1 bit / pixel.
- **JPEG & imágenes codificadas con DWT:** El problema que tienen estos formatos es que, debido a una parte de sus algoritmos llamada cuantización, son formatos de compresión con pérdida. Esto quiere decir que el valor de los pixels puede ser modificado por el proceso de codificación / decodificación. Por tanto, debemos descartar el inyectar información directamente en los pixels de la imagen. Por fortuna, los últimos pasos de la codificación son sin perdida, haciendo posible incrustar el mensaje secreto.

  La solución que se propone es la siguiente: seleccionar los coeficientes que serán usados para incrustar los bits del mensaje; incrustar los bits del mensaje secreto en los bits menos significativos de estos coeficientes y, por último, codificar estos coeficientes.

  Se ha de tener en cuenta que el primer paso reduce la cantidad de coeficientes utilizables, pero es necesario hacerlo, ya que este tipo de algoritmos agrupan el peso de la imagen en algunos coeficientes y muchos de ellos son 0. Por tanto, es recomendable modificar sólo los coeficientes que superen un umbral predefinido.

Incrustar mensajes secretos en audio digital es normalmente un poco más difícil que hacerlo en otros medios, pero para conseguir esconder mensajes en audio de manera satisfactoria se han desarrollado una serie de métodos. La potencia de estos métodos varía y van desde algoritmos que insertan información en forma de ruido o algoritmos que aprovechan métodos de proceso de señales para esconder la información. Algunos de estos métodos son los siguientes:

- **Codificación LSB**: Esta es la forma más simple. Se substituye el bit menos significativo de cada muestra por el bit del mensaje secreto que necesitamos. Hay algunas implementaciones que cambian los dos bits menos significativos, pero para hacer eso se debe de tener en cuenta el archivo de audio que vamos a usar, ya que el ruido podría ser audible.
- **Codificación de Paridad**: En lugar de dividir la señal en muestras individuales, este método divide la señal en regiones de muestras e incrusta cada bit del mensaje secreto en el bit de paridad de las regiones. Si el bit de paridad de la región seleccionada no concuerda con el bit que iba a ser incrustado, el proceso cambia el bit menos significativo de una de las muestras de la región. Así, el emisor tiene mas de una elección al incrustar el bit del mensaje secreto y la señal puede ser cambiada de una forma más discreta.
- **Codificación de Fase**: La codificación de fase se apoya en el hecho de que los humanos no perciben los componentes de fase del sonido tanto como el ruido. En lugar de introducir ruido, esta técnica incrusta los bits del mensaje como desplazamientos de fase en el espectro de fase de una señal digital, consiguiendo una codificación inaudible en términos de señales percibidas como ruido.
- **Spread Spectrum**: Este método trata de esparcir la información secreta a través del espectro de frecuencia de una señal de audio tanto como sea posible. Es análogo a un sistema que, usando la codificación LSB, esparciera los bits del mensaje secreto de forma aleatoria a través del archivo de sonido. Sin embargo, al contrario que en la codificación LSB, este método esparce el mensaje secreto en el espectro de frecuencia usando un código independiente de la señal en sí. Hay dos versiones: de secuencia directa, donde el mensaje se esparce usando una constante y luego se modula con una señal

pseudoaleatoria, y de esquemas de salto de frecuencia, donde el espectro de frecuencia del archivo de audio es alterado de forma que salta rápidamente entre frecuencias.

- **Inserción de eco**: En este método la información se incrusta introduciendo un eco en la señal discreta. Para introducir los datos de manera satisfactoria se modifican tres aspectos del eco: la amplitud, el ritmo de desvanecimiento y el retardo con respecto a la señal original. Los tres parámetros se establecen por debajo del umbral de la capacidad del oído humano para que el eco no se pueda escuchar fácilmente. Además, se modifica el retardo para que represente el mensaje binario a incrustar. Un valor de retardo representa un 1 y dos un 0.

Esteganografía digital en vídeo

Cuando se quiere esconder información dentro de un vídeo, uno de los métodos que se pueden usar es el DCT (Discrete Cosine Transform). Lo que hace este método es modificar cada uno de los frames del vídeo, pero solo de forma que no sea distinguible por el ojo humano. De hecho, lo que hace el método es alterar valores de ciertos lugares de las imágenes, normalmente redondeándolos hacia arriba. Por ejemplo, si una parte de una imagen tiene un valor de 6,667 lo redondeará a 7. Con este método se trabaja de manera similar que cuando se usan métodos de esteganografía en imágenes, la diferencia radica en que la información se esconde en cada frame del vídeo. Como es lógico, si se pretende que la información escondida dentro del vídeo no se note, la cantidad de información escondida deberá ser menor, ya que si escondemos una gran cantidad de información los cambios en el vídeo serán evidentes.

2.- El portador del mensaje: JPEG2000

2.1.- Introducción y características

JPEG 2000 es un estándar de compresión de imágenes basado en transformación de ondas (*wavelet*). Con este tipo de transformación se consigue un nivel de compresión mejor que con la transformada discreta del coseno del antiguo JPEG. Fue desarrollado por el Joint Photographic Experts Group (JPEG) y es parte de la Organización Internacional por la Estandarización (ISO). Aunque dicho estándar fue estructurado inicialmente en seis partes, en el año 2001 se propusieron seis más. Son las siguientes:

- **JPEG2000-1 Core coding system**: descripción mínima del decodificador y una sintaxis simple de code-stream. Esta parte tiene un número limitado de opciones para facilitar el intercambio entre aplicaciones.
- **JPEG2000-2 Extensions**: extensiones del núcleo de codificación que proveen características adicionales de codificación avanzada que se pueden usar para mejorar el rendimiento o para manipular tipos de datos inusuales. Ademas provee un formato de archivo mejorado.
- **JPEG2000-3 Motion JPEG2000**: extensiones provistas para la manipulación de secuencias de imágenes.
- **JPEG2000-4 Conformace testing**: información para que las implementaciones de JPEG2000 se ajusten y adapten a la norma.
- **JPEG2000-5 Reference software**: dos implementeaciones del núcleo del sistema: JJ2000, en Java y JasPer, en C.
- **JPEG2000-6 Compound image file format**: información adicional del formato de archivo.
- **JPEG2000-7**: Esta parte ha sido abandonada.
- **JPEG2000-8 Secure JPEG2000**: descripción de una sintaxis code-stream para interpretar imágenes seguras y el proceso normativo para usar herramientas JPSEC .
- **JPEG2000-9 Interactivity tools, APIsand protocols**: descripción del protocolo de transmisión JPIP, el cual se usa para transmitir imágenes JPEG2000.

- **JPEG2000-10 Volumetric JPEG2000**: codificación de datos volumétricos, y características mejoradas para tratar datos en coma flotante.
- **JPEG2000-11 Wireless JPEG2000**: descripción de técnicas de protección de errores para code-streams de JPEG2000 destinados a detectar y corregir errores producidos durante la transmisión del code-stream.
- **JPEG2000-12 ISO basemedia file format**: definición del archivo base de ISO, proveyendo así un formato que facilita intercambio, edición y gestión.

JPEG2000 empezó a desarrollarse en marzo 1997, cuando los responsables del estándar hicieron un llamado a todo aquel que quisiera contribuir de manera técnica en el proceso. Este llamado lo hicieron con una lista de las características deseadas del núcleo del sistema de codificación:

- **Mejor rendimiento en bit-rates bajos**: debía tener alto rendimiento en todos los bit-rates, sobretodo en los bajos, con respecto al estándar JPEG previo.
- **Compresión de doble nivel y tono continuo**: codificación independiente de diferentes componentes de la imagen, cada una con una profundidad de 1 a 16 bits y usando una arquitectura unificada.
- **Transmisión progresiva a través de precisión en los pixels y la resolución**: para incrementar la calidad d la imagen o la resolución cuando se transmite el code-stream a través de enlaces de comunicación.
- **Compresión con y sin pérdida**: los dos tipos de compresión deben estar establecidos en una única arquitectura.
- **Acceso y proceso aleatorio del code-stream**: debe tener la capacidad de acceder y decodificar áreas especificas de la imagen sin tener que decodificar todo el code-stream.
- **Resistente a errores de bit**:capacidad de gestionar errores en bits introducidos en el code-stream.
- **Poder construir la imagen secuencialmente**: la arquitectura de codificación no tendría que necesitar cargar la imagen entera para producir el code-stream.

Después de dos años de selección y evaluación de los algoritmos propuestos, una versión modificada de *Embedded Block Coding with Optimized Truncation* (EBCOT), por D. Taubman [3], se adoptó como el núcleo del sistema. Las características más importantes del núcleo son:

- **Comprime una vez, descomprime de muchas maneras:** una de las características más importantes de JPEG2000 es la posibilidad seleccionar un área determinada de una imagen y decodificarla a diferente calidad y/o resolución sin tener que procesar todo el code-stream.
- **Edición / proceso de la imagen comprimida**: operaciones simples como la rotación, redimensión o la reducción de calidad de una imagen pueden hacerse sin tener que recodificarla.
- **Progresiónes**: JPEG2000 define la transmisión progresiva por cualidad, resolución, localización espacial y componente. Estas progresiones pueden mezclarse, produciendo ordenes de progresión más potentes. La reorganización de un code-stream en otra orden de progresión no implica tener que recodificar la imagen.
- **Imágenes de poca profundidad de bit**: los componentes con valor binario pueden codificarse ajustando los parámetros de codificación. Sin embargo, el escalado de resolución y calidad no está disponible. Además, las imágenes que contentan un número limitado de colores se pueden codificar como un componente simple gracias a la definición de tablas de búsqueda.
- **Codificación según la región de interés**: se pueden priorizar regiones de la imagen codificándolas primero sin tener que modificar el proceso de codificación.

2.2.- Fases de JPEG2000

JPEG2000 está dividido en cuatro fases principales: transformación, codificación, reorganización del code-stream y control del bit-rate.

- **Transformación de los datos**: en esta fase, que es la primera, se preparan las muestras de la imagen para que en la siguiente fase, que es la fase de codificación, haya mejores

resultados. También en esta fase se modifica el rango de del valor de las muestras para simplificar ciertas partes de la implementación.

Cuando las operaciones de color y la transformada de onda (*wavelet transform*) se procesan usando coma flotante, el rendimiento de la codificación se incrementa considerablemente, pero en ese caso la compresión con perdida no es posible. Para permitir comprensión con y sin perdida sin sacrificar rendimiento en la codificación, se permite que estas operaciones se puedan hacer en coma flotante y enteros, produciendo así dos caminos de codificación: reversible e irreversible.

La codificación según la región de interés también se lleva a cabo en esta fase. Se usa una operación simple que da prioridad a los coeficientes de una región dada, de tal forma que en la fase de codificación se codifican esa región primero.

Antes de la codificación la imagen se divide en una estructura jerárquica. Esta operación no se considera parte de la fase de codificación porque no se dan operaciones relacionadas con la codificación, sin embargo juega un papel importante en el sistema de codificación.

- **Codificación de los datos**: Las partes más pequeñas producidas por esta división jerárquica se llaman bloques de código (*codeblocks*), que son pequeños bloques de coeficientes que son codificados de manera independiente. Esta fase de codificación usa dos codificadores: un codificador fraccional de bit plane y un codificador aritmético, y produce un code-stream para cada bloque de código. El proceso que se lleva a cabo en esta fase también se llama codificación Tier-1.
- **Reorganización del code-stream**: En esta fase tiene lugar la construcción final del code-stream de JPEG 2000. Implica una reorganización de los code-streams generados en la fase anterior de cada bloque de código y la codificación de alguna información de estos bloques de código. La codificación de esta información también se conoce como codificación Tier-2.
- **Control de bit-rate**: La función de esta fase es controlar el bit-rate o la distorsión del code-stream final producido por la codificación. Cuando se selecciona un bit-rate, el control del bit-rate optimiza la calidad del code-stream final, minimizando la distorsión. En cambio, cuando se selecciona una calidad determinada, esta fase optimiza el bit-rate minimizando su longitud.

Además, el control del bit-rate se ocupa de la construcción de capas de calidad, las cuales son estructuras del code-stream y su importancia radica en que proveen escalabilidad de calidad, lo cual es una característica fundamental de JPEG2000.

El estándar define una guía para el control del bit-rate tal y como se propone en EBCOT, sin embargo esta fase puede implementarse de diferentes maneras modificando las fases anteriores. Por esta razón desde el año 2002 han aparecido más de veinte metodos diferentes de control del bit-rate.

En la siguiente figura, extraida de [4], se puede apreciar cada una de las fases de JPEG2000.

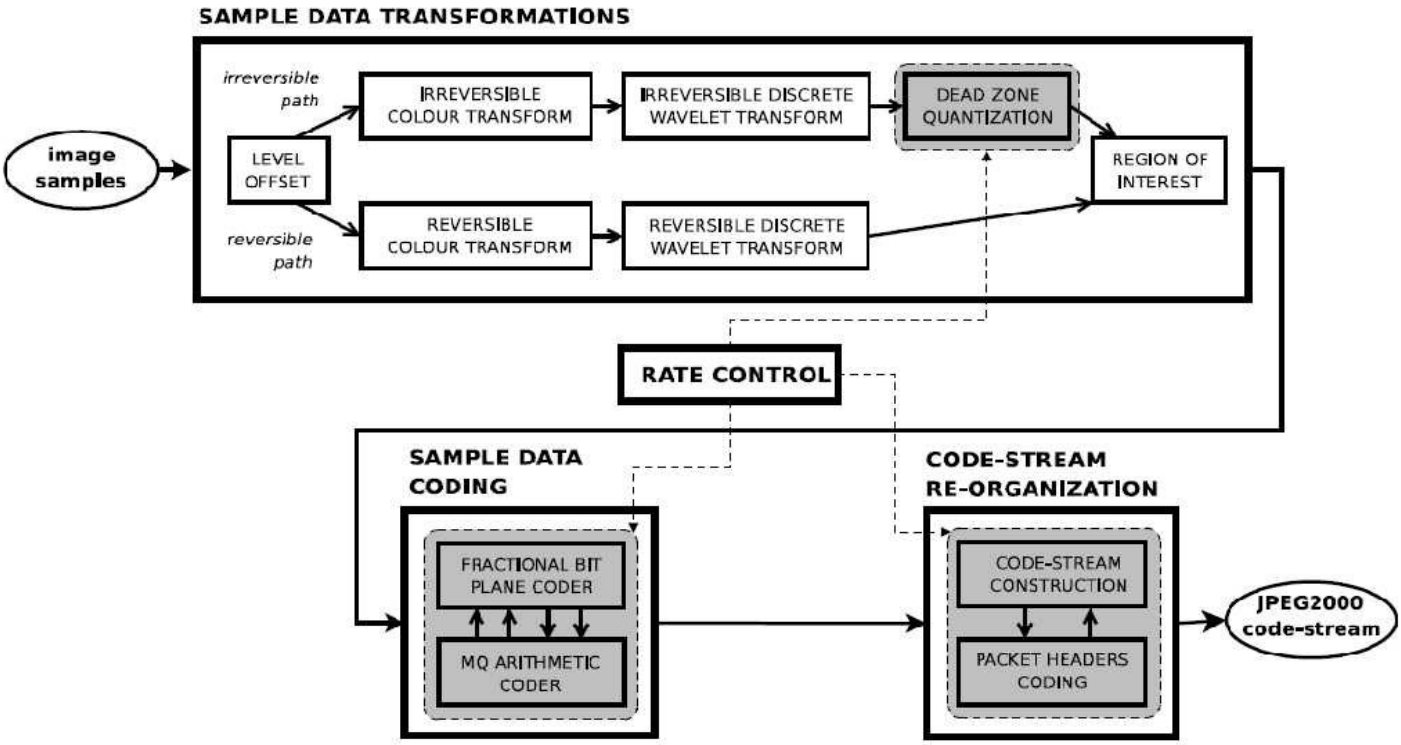

Fig. 1: Fases de JPEG2000

2.3.- Implementación utilizada: BOI

BOI es una implementación del estándar JPEG2000. Aunque hay otras aplicaciones que incorporan el estándar JPEG2000, BOI fue programado y diseñado para proveer una base que ayudara a probar y desarrollar nuevas ideas dentro del núcleo de codificación de JPEG2000. Esa es una de las razones por la cual BOI ha sido escogido para realizar este proyecto. Su estructura modular lo hacían idóneo para añadir los métodos esteganográficos en dicha estructura sin tener que modificar ninguna parte principal de la implementación.

BOI además provee diferentes funcionalidades que pueden ser controladas a través de parámetros. Más de treinta parámetros permiten combinar los diferentes procesos para producir una gran cantidad de tipos de imágenes comprimidas. Se ha añadido un parámetro a este grupo que controla el activar o no la esteganografía en la compresión de la imagen. También incorpora algoritmos para que el tratar con diferentes tipos de datos sea lo suficientemente flexible, para que sea capaz de controlar y monitorear las fases de compresión, para extraer estadísticas, etc.

Pero debido a todo lo expuesto, BOI es una plataforma para probar nuevas ideas y facilitar el desarrollo de las mismas, no debería integrarse en ningún tipo de aplicación que necesite ayuda para codificar / decodificar archivos JPEG2000, ya que hay otras implementaciones que están programadas con ese enfoque y resultarán en un rendimiento mayor.

BOI se divide en 2 aplicaciones: el codificador y el decodificador. Ambos programas están encapsulados en archivos jar que se pueden usar separadamente usando la JVM. Cada aplicación acepta su propio juego de argumentos que están explicados con detalle en la documentación de BOI que puede encontrarse en [5].

La distribución incluye shell scripts (BOIcode y BOIdecode) para facilitar la ejecución tanto para entornos GNU/Linux como para Windows. BOI utiliza una cantidad considerable de memoria, por tanto es recomendable especificar la cantidad máxima de memoria disponible, usando el parámetro -Xmx de la JVM, en los shell scripts mencionados anteriormente.

Por ejemplo, el archivo **BOIcode.bat** usado durante el proceso del proyecto ha sido el siguiente:

```
java -Xmx1024m -classpath dist\BOI.jar BOI.BOICode -i lena.pgm -o lena -of 1 0 2 -stg 1
```

Donde los parámetros significan lo siguiente:

- **-i**: establece el nombre del archivo de entrada. Los formatos de imagen válidos en BOI son: pgm, ppm, pbm, jpg, tiff, png, bmp, gif y fpx.
- **-o**: establece el nombre del archivo de salida. El nombre se ha de escribir sin extensión, ya que la extensión viene determinada por el parámetro siguiente.
- **-of**: indica el tipo de archivo que se generará. Está definido por tres bits: el primero indica el t ipo de bitstream; el segundo el orden de la progresión y el tercero indica el tipo de cabecera usado (y la extensión que tendra el archivo).
- **-stg**: este último parámetro indica si los métodos esteganográficos estarán activados, 1, o desactivados, 0.

El archivo **BOIdecode.bat** usado durante el proceso del proyecto ha sido el siguiente:

```
java -Xmx1024m -classpath dist\BOI.jar BOI.BOIDecode -i lena.jp2 -o lena.jpg -stg 1
```

Donde los parámetros y su valor son equivalentes a los del shell script del codificador. Siendo la única diferencia que en el parámetro que establece el archivo de salida, se ha de especificar una extensión para el formato de imagen que sea válido. Los formatos de imagen válidos son los siguientes: raw/img (si se usa esta opción, el argumento -og es obligatorio), pnm / pgm / ppm (pgm se usa cuando la imagen tiene 1 componente, ppm cuando tiene tres componentes y pnm para ambos casos), tiff, jpg y bmp.

3.- Códigos correctores de error y Objetivos

Como ya se ha visto en las secciones anteriores, la esteganografía se puede presentar de muchas maneras. En este proyecto, sin embargo, nos centraremos en la esteganografía en imágenes. Más concretamente en esteganografía en imágenes JPEG2000.

La primera aproximación para esconder información en imágenes JPEG2000 es usando el primero de los métodos expuestos en el punto cuatro de esta misma memoria: el LSB (*Least Significant Bit*).

Un método más interesante es el llamado codificación de matrices (*matrix encoding*) el cual está explicado en [7]. Basado en este método está el segundo método expuesto en este proyecto: el F5, presentado por Andreas Westfeld en [8].

El método de codificación de matrices está basado en códigos binarios lineales y, siguiendo [9], daremos unas cuantas definiciones. Sean n y t enteros positivos donde $t \leq n$, y sea X un conjunto finito. Un protocolo esteganográfico del tipo *[n,t]* sobre X resulta en un par de mapas $X^n \times X^n \rightarrow X^n$ y $r: X^n \rightarrow X^t$ tal que $r(e(s,v)) = s$ para todo $s \in X^t$ y $v \in X^n$. Los mapas e y r son los mapas de incrustación y recuperación respectivamente. El radio del protocolo viene definido por $p = \max \{ d(v,e(s,v)) \mid s \in X^t, v \in X^n \}$ donde d es la distancia de Hamming.

El mapa de incrustación de un protocolo esteganográfico [n,t] con radio p (protocolo [n,t,p] para acortar) nos permite esconder t símbolos de información en una cadena de n símbolos cambiando, como mucho, p de esos símbolos.

Un clase importante de protocolos esteganográficos se pueden definir gracias a la teoría de códigos. Los códigos correctores de errores se usan normalmente para corregir errores o borrado de datos en las transmisiones de datos. Podemos encontrar una descripción de la relación entre códigos correctores y esteganografía en [10], por Zhang y Li, donde nos muestran que hay una relación entre los códigos correctores perfectos y los códigos donde se encuentra la máxima longitud de incrustación (*maximum length embeddable codes o MLE*

codes). La mayor parte de los códigos usados en esteganografía son lineales y la existencia de una matriz de control ayuda a la hora de diseñar buenos protocolos esteganográficos.

Sea C un código lineal [n, n – t] sobre un cuerpo finito $GF(q)$, por tanto es un subespacio lineal de $GF(q)^n$ de dimensión $k = n – t$. El radio de recubrimiento p del código C se define como $p = \max_{v \in X^n} \{d(v,C)\}$ donde $d(v,C)$ es la distancia de Hamming mínima desde el vector v al código C. Sea $v = (v_1,v_2,...,v_n) \in X^n$, el conjunto $\text{supp}(v) = \{i \mid x_i \neq 0\}$ se llama el soporte del vector v.

Sea $X = GF(q)$ y sea H la matriz de control de C. Para cada $v \in X^n$, el vector $H \cdot v^T$ es el síndrome de v, donde v^T es el vector v como un vector columna. El conjunto $C + v = \{ x + v \mid \forall \ x \in X^n \}$ es la clase de todos los vectores en X^n con el mismo síndrome $u = H \cdot v^T$. Podemos seleccionar uno de los vectores con el peso mínimo en la clase $C + v$, lo llamaremos l_u y, aunque no es único, lo llamaremos líder de la clase.

Sea r: $X^n \rightarrow X^t$ el mapa lineal cuya matriz de control es H, por tanto para todo $v \in X^n$, $r(v) = H \cdot v^T$. Así pues, r es el mapa de recuperación de un protocolo esteganográfico que llamaremos lineal para enfatizar que el mapa de recuperación r es un mapa lineal.

El algoritmo de incrustación para calcular $e(s,v)$ para un protocolo esteganográfico lineal es el siguiente:

Algoritmo basado en clases

(1) calcular $u := r(v) – s$

(2) definir $e(s,v) := v - l_u$, donde l_u es el líder de la clase de todos los vectores en X^n con el mismo síndrome u. Por tanto, $r(l_u) = u$.

Podemos formular el algoritmo mencionando como una tabla (una tabla líder-síndrome) que conste de q^t vectores l_u de longitud n y asociar a cada uno de ellos su valor de síndrome correspondiente.

Cuando el código C es el código de Hamming, el algoritmo de los cosets y la función de recuperación r coincide con el algoritmo F5 desarrollado en [8] y que será explicado más tarde en el punto 4. Es sencillo ver que, en este caso, el coste computacional del algoritmo es O(n).

El radio de recubrimiento *p* de un código es un parámetro importante en esteganografía que nos lleva a considerar códigos de cobertura (*covering codes*). La definición de estos códigos y su relación con la esteganografía podemos encontrarla en [11] donde se describe el rendimiento de algunos de estos códigos que han sido usados en esteganografía.

Hay dos parámetros que ayudarán a evaluar el rendimiento de un protocolo esteganográfico [n,t,p]:

- **La media de distorsión** (*Average Distorsion*): D = $\frac{R_a}{n}$, donde R_a es el número de cambios esperados en un mensaje con distribución uniforme.
- **Grado de incrustación** (*Embedding Rate*): $\frac{t}{n}$.

Por lo general, si dos protocolos tienen grados de incrustación iguales, el que tenga la media de distorsión más pequeña será mejor que el otro.

En este proyecto sólo tendremos en cuenta el caso de alcaide pasivo. P. Moulin y Y. Wang probaron en [6] que existe un límite de capacidad teórico a la hora de esconder información. Es el máximo grado de incrustación al que se puede llegar con un protocolo esteganográfico dada una media de distorsión *D* y teniendo en cuenta que no hay ningún tipo de error (es decir, que los únicos bits que sufren cambios son los cambiados por el protocolo esteganográfico). Normalmente es difícil calcular esta capacidad máxima, pero en algunos casos es posible. Por ejemplo en el caso de Bernoulli $(\frac{1}{2})$, es decir, los únicos símbolos que se usarán en el mensaje son los componentes del conjunto $X = \{0,1\}$ y además cumplen la distribución de Bernoulli con $p = \frac{1}{2}$.

Si *D* es la media de distorsión, entonces la máxima capacidad para esconder información, *C(D)*, para este caso es:

$C(D)=H(D)=-D\log_2(D)-(1-D)\log_2(1-D)$, donde $0\le D\le\frac{1}{2}$ y H es la función entropía.

En el siguiente gráfico podemos ver la representación de la fórmula anterior.

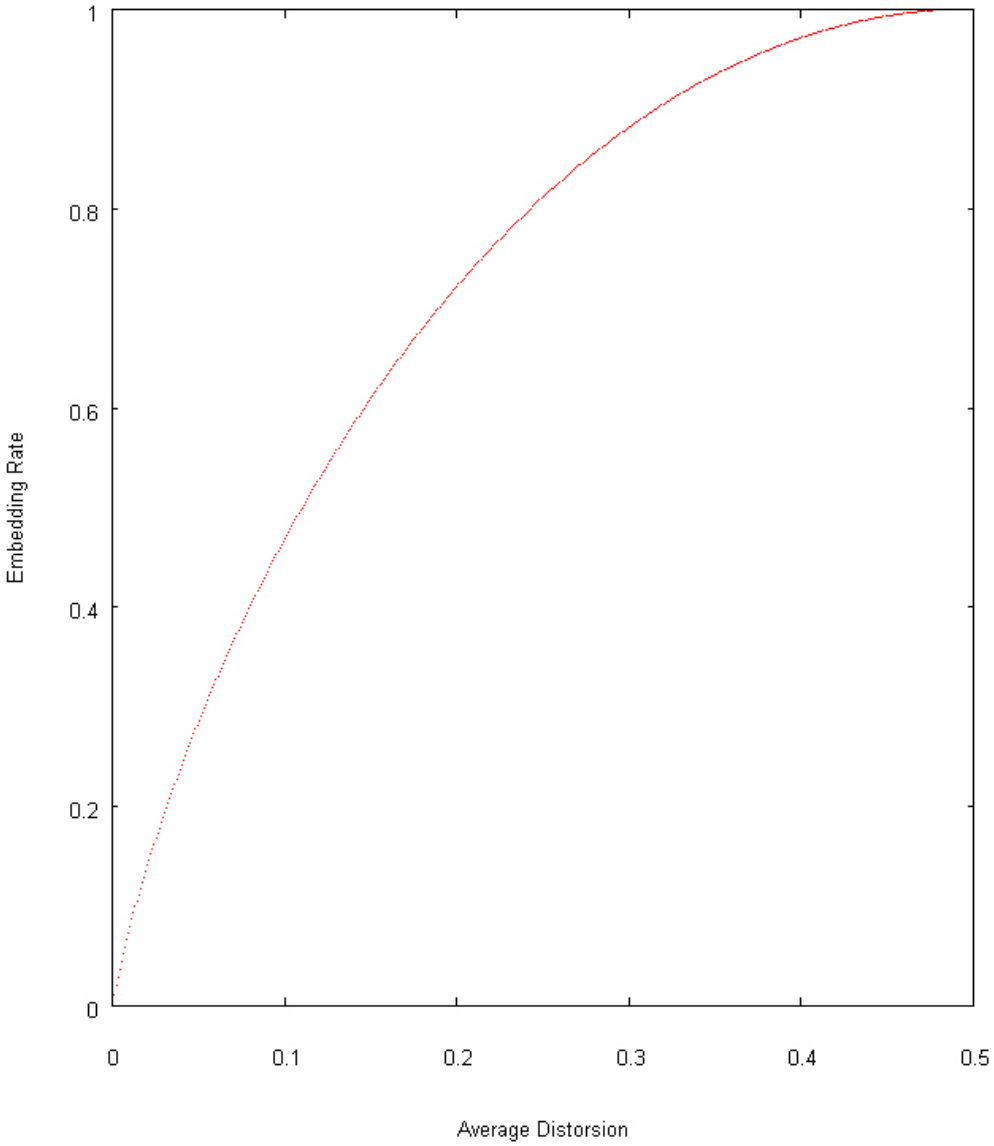

Fig 2: Limite teórico en el caso de Bernoulli

Es importante darse cuenta que el radio de recubrimiento p de un código C es el número de cambios más grande posible y que el propósito de la función de incrustación es minimizar la media de cambios por bits incrustados, R_a. Dado un código de longitud n, dimensión $k = n - t$ y radio de recubrimiento p, siempre podemos componer un protocolo

esteganográfico con un grado de incrustación $\frac{k}{n}$ y una media de distorsión dada por $\frac{p}{n}$. Sin embargo, tal y como se señala en [12], para el mismo grado de incrustación, la media de distorsión menor no siempre se obtiene con el código de menor radio de recubrimiento.

La media de distorsión real viene dada por el algoritmo de codificación usado siendo ese uno de los objetivos de este proyecto: visualizar para unos casos específicos que el algoritmo presentado en [13], que está basado en códigos lineales, tiene una media de distorsión mejor que la obtenida con algoritmos como el F5. Ese algoritmo, esteganografía con el producto de dos códigos perfectos (PPC), es el tercer método de este proyecto.

Por tanto, el principal objetivo de este proyecto es, usando JPEG2000 como portador del mensaje, programar el algoritmo presentado en [13] y comparar el resultado con los otros dos métodos presentados en el proyecto: LSB y F5.

Como había que comparar todos los métodos en un mismo entorno, los tres métodos han sido programados o trasladados a Java y JPEG2000. La elección de este lenguaje de programación vino dada por la implementación de JPEG2000 utilizada, BOI, la cual está programada en Java y por tanto parecía la elección más adecuada.

4.- Implementación

Como se ha mencionado anteriormente, en este proyecto se han implementado tres protocolos esteganográficos diferentes: LSB, F5 y PPC. Pero aún y cuando son unos protocolos muy diferentes entre ellos hay partes de la implementación que son comunes. A continuación se explicarán esas partes.

4.1.- Partes comunes para los tres métodos

Los tres métodos están implementados dentro de BOI. Es por esa razón que hablaremos primero de su estructura.

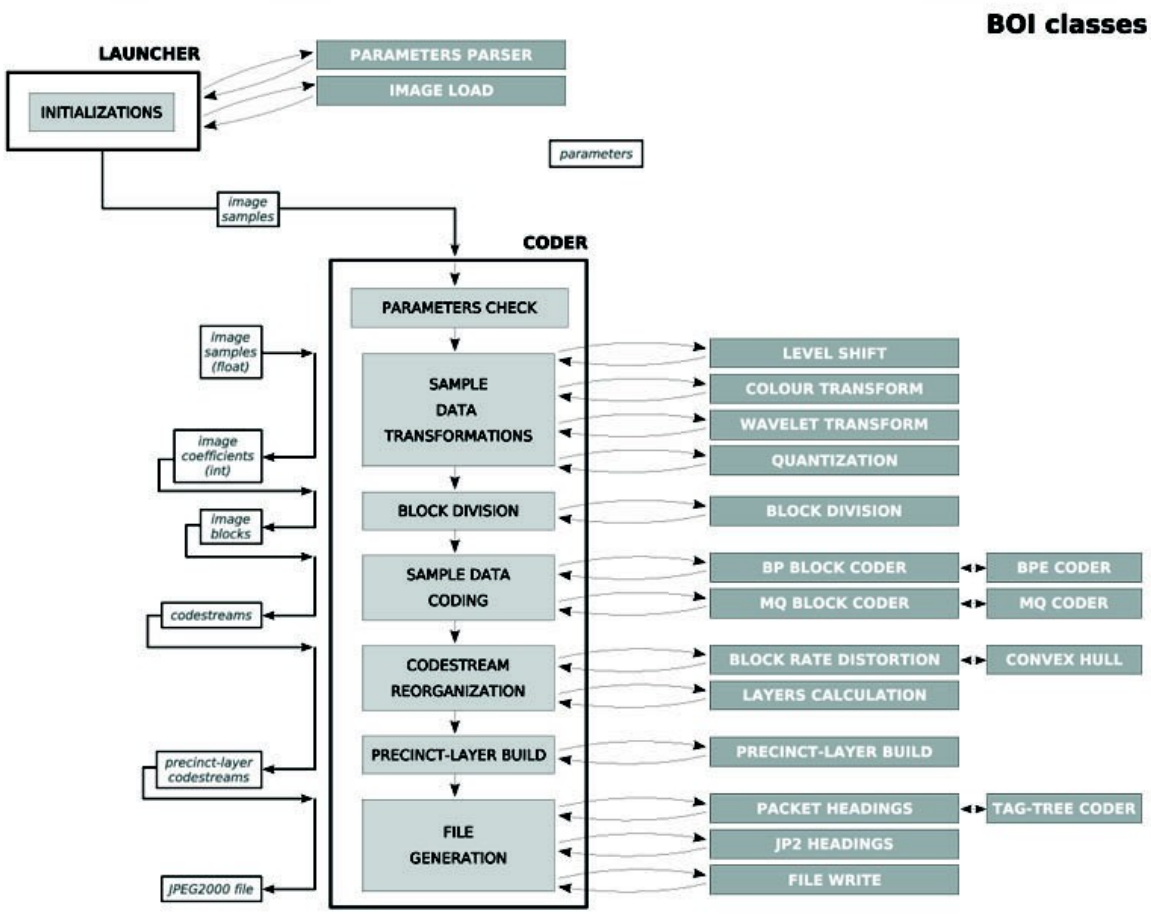

Fig. 3: Clases de BOI

En la imagen anterior podemos ver que la clase Launcher recibe los argumentos, carga la imagen y le pasa estos argumentos a la clase Coder, que es la clase principal de la aplicación. Coder comprueba que todos los parámetros recibidos sean correctos y entonces llama a la primera de las clases que se ocupa de las etapas de compresión: Sample Data Transformations. Esta clase está compuesta de las cinco clases que vemos al lado con el fondo verde y las letras blancas: Level Shift, Colour Transform, Wavelet Transform y Quantization.

Es después de esta fase, Quantization, donde se llama a las clases de los protocolos esteganográficos. La elección de este lugar para esconder la información viene dada por el hecho de que justo después de la cuantización las etapas son siempre sin pérdida y por tanto no existe el riesgo de perder la información incrustada. También a partir de este momento las muestras de la imagen pasan a ser números enteros y no números reales, con lo cual su manipulación es más sencilla.

Ya que es la clase Coder la que se ocupa de controlar todas las clases implicadas en el proceso, las llamadas a las clases de los métodos están en dicha clase siguiendo el siguiente patrón:

```
System.out.println ("Select steganography algorithm: ");
System.out.println ("1.- LSB\n2.- F5\n3.- PPC");
str = in.readLine();

if (str.equals("1"))
{
      //LSB STEGANOGRAPHY
      System.out.println ("Begining LSB Steganography");
      LSB lsb = new LSB(imageSamplesInt);
      try
      {
            lsb.setParameters(new FileInputStream("./messagefiles/bin.noise"));
      }catch(Exception e) {e.printStackTrace();}
      imageSamplesInt = lsb.run();
      //Free unused memory
      lsb = null;
      //Show statistics
      showTimeMemory("LSB");
}else if(str.equals("2")) ...
```

Como vemos, al usuario se le da a elegir el algoritmo esteganográfico a utilizar y dependiendo de su elección Coder llamará a una clase o a otra. Cada una de las clases recibe la imagen como una matriz multidimensional de valores enteros (que tendrá una dimensión o tres dependiendo de si es una imagen en escala de grises o a color, respectivamente.) Se inicializan los parámetros, que en el caso de LSB y PPC son el directorio y nombre del fichero. En cambio, en el caso de F5 se ha de pasar también una palabra clave que será usada en el generador de números pseudoaleatorios cuya función será explicada más adelante. Una vez comprobado que todos los parámetros son correctos, se ejecuta el algoritmo en cuestión, se guarda el resultado y se libera la memoria usada.

Otro elemento común en los tres métodos es la palabra de estado, tomada de la implementación de F5 en [8] y adaptada para los otros dos métodos. La idea en esta palabra de estado es poder pasar, incrustados en la imagen, los parámetros necesarios para que la función de extracción pueda extraer el mensaje correctamente. Estos parámetros varían dependiendo del método usado. En el LSB simplemente hay que pasar el tamaño del mensaje incrustado, pero en el F5 tambien se pasa el parametro k y en el PPC los parametros k y k_2.

Primero debemos averiguar el tamaño del mensaje que queremos incrustar y la longitud de los códigos que vamos a usar en F5 y PPC. Y una vez averiguadas creamos una palabra de estado de 32 bits, una potencia de dos lo suficientemente grande como para poder albergar el tamaño del mensaje y los dos parametros.

```
try
{
        byteToEmbed = embeddedData.available();
}catch (Exception e) {e.printStackTrace();}

byteToEmbed |= k<<24;          //F5 and PPC
byteToEmbed |= k2<<28;         //PPC only
```

Cuando tenemos la palabra de estado creada, lo que debemos hacer es incrustarla en los siguientes coeficientes válidos de la imagen, que variarán dependiendo el método que estemos usando. El código siguiente pertenece a LSB, que es el más básico de los tres. En F5

y PPC hay que tener en cuenta otros aspectos, pero la idea básica para crear la palabra de estado es la misma.

```
nextBitToEmbed = byteToEmbed & 1;
byteToEmbed >>= 1;
availableBitsToEmbed=31;

for(i=0; i<coeffCount; i++)
{
        if (imageSamplesArray[i] > 0)
        {
                if ((imageSamplesArray[i]&1) != nextBitToEmbed)
                {
                    imageSamplesArray[i]--;
                }
        } else
        {
                if ((imageSamplesArray[i]&1) != nextBitToEmbed)
                {
                        imageSamplesArray[i]++;
                }
        }
        if (availableBitsToEmbed==0) break;  // statusword embedded.
        nextBitToEmbed = byteToEmbed & 1;
        byteToEmbed >>= 1;
        availableBitsToEmbed--;
}
```

Para extraer la palabra de estado simplemente debemos usar el método de extracción del LSB en los 32 primeros coeficientes válidos de la imagen.

```
for (i=0; availableExtractedBits<32; i++)
{
      extractedBit=imageSamplesArray[i]&1;
      extractedFileLength |= extractedBit << availableExtractedBits++;
}
extractedFileLength &= 0x007fffff;
```

En el caso de F5 y PPC también deberemos extraer las longitudes de los códigos de la palabra de estado.

```
k2 = extractedFileLength >> 28; //PPC only
k = extractedFileLength >> 24;
k = k & 15;
k2 = k2 & 15; //PPC only
```

Se debe mencionar también el Decoder. Es la clase que se ocupa de realizar el proceso inverso del Coder. Aún y así el Decoder es más simple que el Coder. Existen los mismos paquetes y clases, exceptuando el *BOI Codestream Reorganization*. Todas las clases son réplicas de las usadas en el Coder, que realizan los algoritmos a la inversa. Se mantiene el nombre de las variables y tienen la misma funcionalidad. Por ese motivo la inclusión de los algoritmos de extracción de los métodos es equivalente a la forma en que fueron incluidos en el Coder: justo antes de la descuantización.

El cálculo de la media de distorsión y el grado de incrustación también es común para los tres métodos. Estos dos valores se usan para hacer los gráficos y así poder comparar los diferentes métodos.

Para calcular la media de distorsión se ha llevado la cuenta de cada uno de los cambios realizados para incrustar el mensaje en la imagen y ese número se divide por el número total de coeficientes de la imagen utilizados para esconder el mensaje.

Para calcular el grado de incrustación se ha llevado la cuenta de cada uno de los bits del mensaje incrustados en la imagen y ese número se divide por el número total de coeficientes de la imagen utilizados para esconder el mensaje.

Utilizando diferentes archivos de texto de diferente tamaño, de 37 bytes a 34.756 bytes (con este tamaño se supera el tamaño máximo de la imagen usada para las pruebas en el método LSB, que es el método con más capacidad), se calculan diferentes valores para la media de distorsión y el grado de incrustación. Con esos valores escritos en un fichero de texto se dibuja una gráfica para cada método con el grado de incrustación en función de la media de distorsión.

El último elemento común en los tres métodos es la imagen utilizada para hacer las pruebas, la misma imagen que se usó en las pruebas de BOI.

Fig. 4: Lenna, la imagen usada en las pruebas

La imagen está en escala de grises, sus dimensiones son 512x512, su formato es .pgm y su tamaño es de 262.159 bytes.

4.2.- LSB (Least Significant Bit)

Este método es el más usado cuando se habla de esteganografía en imágenes ya que es la aproximación más simple y más directa. La idea es cambiar el bit menos significativo (sumándole o restándole 1) de cada byte de la imagen introduciendo así el mensaje

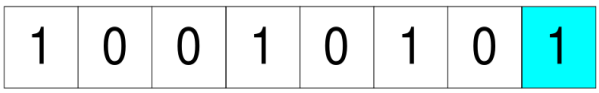

Fig. 5: Bit menos significativo

que queremos enviar. Como vemos en la Fig. 5, el bit menos significativo en un número binario es el que está mas a la derecha. Cuando una imagen en JPEG2000 sale de la cuantización, queda representada por una secuencia de números enteros, los cuales a su vez pueden representarse de forma binaria. Así, si en los coeficientes de la imagen cambiamos el bit menos significativo, el ojo humano no puede apreciar la diferencia, ya que pasaría de 149 a 150 por ejemplo, y eso es lo que se aprovecha este método.

La idea, por tanto, es hacer que el conjunto de bits menos significativos de los coeficientes de la imagen coincida con la representación en binario del mensaje para así, a la hora de la extracción, leer cada uno de esos bits y recuperar el mensaje. Además podríamos interpretar que con esta técnica se esconden dos bits por cambio ya que, en un caso aleatorio, el 50% de los bits no necesitan ser cambiados porque coinciden con los bits que queremos introducir en la imagen.

Por ejemplo, para esconder la palabra LSB (cuya representación en binario es: 01001100 01010011 01000010) necesitaríamos veinticuatro coeficientes de la imagen. Supongamos que la representación en binario de ocho primeros coeficientes de la imagen es:

01010100	00101100
11100100	10110111
10001111	10111001
10101010	01111100

Después de de incrustar la L, los coeficientes de la imagen quedarían de la siguiente manera, con los cambios en rojo y los que quedan igual en azul.

0101010**0**	0001011**1**
1110010**1**	1011011**1**
1000111**0**	1011100**0**
1010101**0**	0111110**0**

Se continuaría con la S y la B de manera análoga, y para extraer el mensaje bastaría con leer el lsb de cada uno de los veinticuatro coeficientes.

En la implementación del método en este proyecto la fase de incrustado transcurre de la siguiente forma: Tras recolectar los coeficientes de la imagen en un array unidimensional e insertar la palabra de estado se procede a incrustar el mensaje tal y como vemos en el siguiente fragmento de código.

```
for(; i < coeffCount; i++)
{
      if (imageSamplesArray[i] >= 0)
      {
              if ((imageSamplesArray[i]&1) != nextBitToEmbed)
                      imageSamplesArray[i]--;
      }else
      {
         if ((imageSamplesArray[i]&1) != nextBitToEmbed)
             imageSamplesArray[i]++;
      }
      if (availableBitsToEmbed==0)
      {
              try
              {
                      if (embeddedData.available() == 0) break;
                      byteToEmbed = embeddedData.read();
              } catch (Exception e) {e.printStackTrace(); break;}
              availableBitsToEmbed = 8;
      }
      nextBitToEmbed = byteToEmbed & 1;
      byteToEmbed >>= 1;
      availableBitsToEmbed--;
}
```

Para cada coeficiente del array que contiene los coeficientes de la imagen comprobamos si el siguiente bit para incrustar coincide con el lsb del coeficiente actual y si no coincide sumamos o restamos 1. Después comprobamos si tenemos más bits para incrustar y si no es así, leemos el byte siguiente del mensaje y repetimos el proceso. Después de eso ponemos los coeficientes modificados en la imagen y continuamos con la siguiente fase de JPEG2000.

```
  try
  {
        fos = new FileOutputStream("output.txt");
        for (; i < coeffCount; i++)
        {
                extractedBit = imageSamplesArray[i]&1;

                extractedByte |= extractedBit << availableExtractedBits++;

                if (availableExtractedBits == 8)
                {
                        fos.write((byte) extractedByte);
                        extractedByte = 0;
                        availableExtractedBits = 0;
                        nBytesExtracted++;
                        if (nBytesExtracted == extractedFileLength) break;
                }
        }

        if (nBytesExtracted<extractedFileLength)
        {
                System.out.println("Incomplete file: only "+nBytesExtracted+"
                of"+extractedFileLength+" bytes extracted");
        }
} catch (Exception e) {e.printStackTrace();}
```

El la fase de extracción sólo debemos extraer la palabra de estado para conocer el momento en que debemos detener la recolección del mensaje y hasta ese momento, iremos agrupando los bits menos significativos de los coeficientes de la imagen en grupos de ocho para escribirlos en el archivo de texto de salida.

La gráfica resultante de aplicar el método a los diversos archivos de texto mencionados anteriormente podemos verla en la página siguiente.

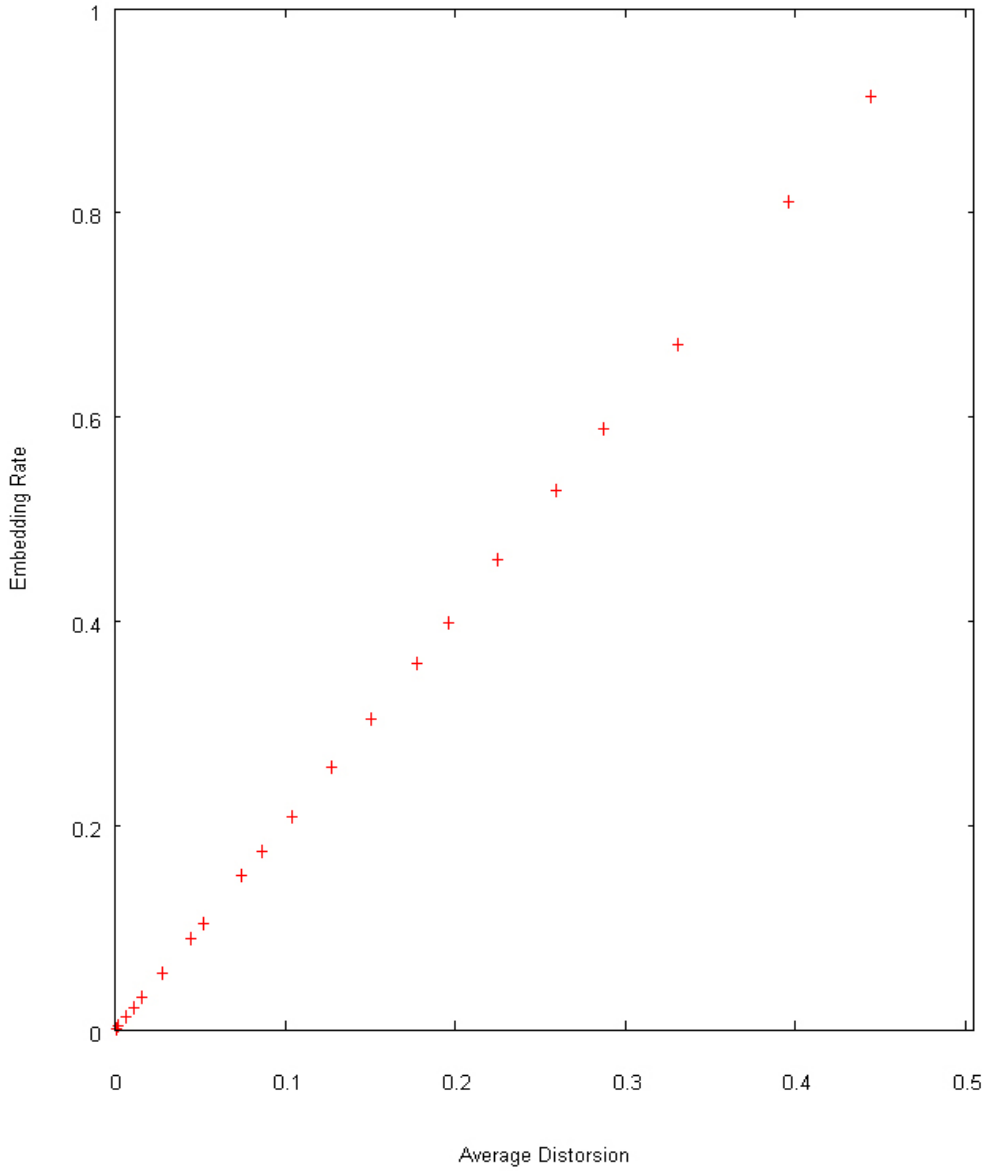

Fig. 6: Rendimiento del LSB

4.3.- F5

El segundo método del proyecto es el F5, creado por Andreas Westfeld y presentado en [8]. En ese artículo A. Westfeld habla de varios algoritmos esteganográficos y los va desechando uno por uno explicando sus carencias. Lo hace como método para explicar el nuevo método que ha desarrollado que, además de ofrecer resistencia ante ataques visuales y estadísticos, presenta una capacidad esteganográfica mayor.

Uno de los problemas que menciona es que esos algoritmos incrustan los datos de manera continua. Cuando incrustamos datos en imágenes tenemos una capacidad limitada para esconder el mensaje. En muchas ocasiones, el mensaje que queremos incrustar no requiere usar toda esa capacidad y parte de la imagen queda sin utilizar. Como queda representado en la Fig. 7, eso provoca que los cambios (representados por x) se concentren en el principio de la imagen y que el resto de la imagen permanezca inalterada. Esto provoca que la imagen sea vulnerable a un ataque estadístico. Para prevenir ataques, la función de incrustación debería usar la imagen una forma regular.

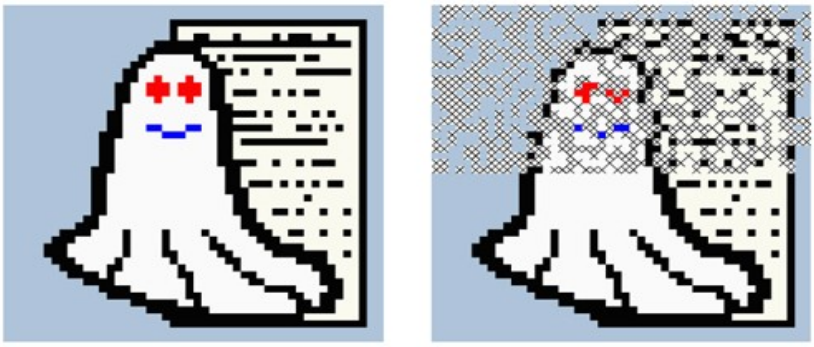

Fig. 7: Efecto de incrustar de forma continua

Lo que hace F5 para combatir ese problema es usar una permutación para mezclar los coeficientes de la imagen y es entonces cuando incrusta el mensaje. Esta permutación depende de una clave generada por una contraseña que el receptor debe conocer para poder generar de nuevo la misma permutación y así extraer el mensaje correctamente. La Fig. 8 muestra el resultado de dicha permutación.

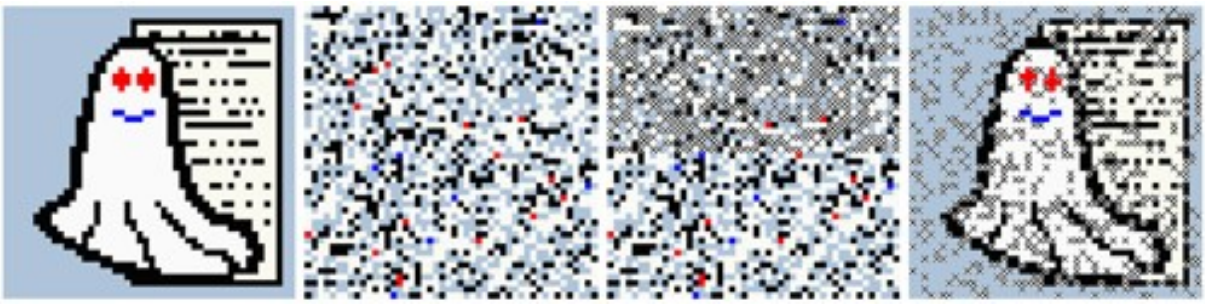

Fig. 8: Efecto de incrustar con una permutación

Como ya se ha mencionado anteriormente, F5 está basado en la codificación de matrices, presentado por R Candrall en [7]. La codificación de matrices lo que consigue es que si la mayor parte de la capacidad del portador queda sin usar, el número de cambios a realizar decrece.

Supongamos que queremos incrustar dos bits x_1 y x_2 , en tres bits a_1, a_2 y a_3, cambiando únicamente un bit como mucho. Nos podemos encontrar cuatro casos:

- $x_1 = a_1 \oplus a_3, x_2 = a_2 \oplus a_3 \rightarrow$ No cambiar nada
- $x_1 \neq a_1 \oplus a_3, x_2 = a_2 \oplus a_3 \rightarrow$ Cambiar a_1
- $x_1 = a_1 \oplus a_3, x_2 \neq a_2 \oplus a_3 \rightarrow$ Cambiar a_2
- $x_1 \neq a_1 \oplus a_3, x_2 \neq a_2 \oplus a_3 \rightarrow$ Cambiar a_3

En los cuatro casos sólo cambiamos un bit. En general tenemos una palabra código *a* de *n* bits que se pueden modificar por *k* bits del mensaje secreto *x*. Sea *f* una función hash que extrae *k* bits de esa palabra código *a*. La codificación de matrices nos permite encontrar una palabra código *a'* para cada palabra código *a* y con $x = f(a')$ tal que la distancia de Hamming es $d(a,a') < d_{max}$.

Este código lo podemos representar con una tupla con los siguientes elementos (d_{max},n,k): en una palabra código de *n* bits se cambiarán como máximo d_{max} posiciones para incrustar *k* bits. Como ya hemos visto anteriormente, F5 es el caso particular en que $d_{max} = 1$. Para $(1,n,k)$ las palabras código tienen longitud $n = 2^k - 1$.

Para implementar F5 hay que seguir los siguientes pasos:

(1) Comenzar la compresión en JPEG2000 y detenerse tras la cuantización.
(2) Inicializar un generador de números aleatorios con la clave derivada de la contraseña.
(3) Inicializar la permutación con dos parámetros: el generador de números aleatorios y el número de coeficientes.

```
F5Random random = new F5Random(password.getBytes());
Permutation permutation = new Permutation(coeffCount, random);
```

(4) Copiar los coeficientes de la imagen en un array y calcular el parámetro *k* usando la capacidad de la imagen y la longitud del mensaje secreto.
(5) Calcular la palabra código de longitud $n = 2^k - 1$.

```
for (i=0; i<imageSamplesArray.length; i++)
 {
       if (imageSamplesArray[i]==2) _two++;
       if (imageSamplesArray[i]==-2) _two++;
       if (imageSamplesArray[i]==1) _one++;
       if (imageSamplesArray[i]==-1) _one++;
       if (imageSamplesArray[i]==0) _zero++;
 }
 _large = coeffCount - _zero - _one - _two;
 _expected = _large+ _one/3;

 for (i = 1; i < 15; i++)
 {
       int usable, changed;
       n = (1 << i) - 1;
       usable = ((_expected * i) / n) - ((_expected * i) / n) % n;
       usable /= 8;
       if (usable == 0) break;
       if (usable < byteToEmbed + 4) break;
 }
 k = i - 1;
 n = (1 << k) - 1;
```

(6) Incrustar el mensaje secreto con el código (1,n,k) usando codificación de matrices:

a) Coger el primer byte del mensaje secreto y prepararlo para incrustarlo.

```
embeddingLoop:
 do
 {
       kBitsToEmbed = 0;
       // get k bits to embed
       for (i=0; i<k; i++)
       {
              if (availableBitsToEmbed==0)
              {
                     // If the byte of embedded text is
                     // empty, we will get a new one.
[...]/[...]
```

```
[...]/[...]
                        try
                        {
                                if (embeddedData.available()==0)
                                {
                                        isLastByte = true;
                                        break;
                                }
                                byteToEmbed = embeddedData.read();
                                byteToEmbed ^= random.getNextByte();
                        } catch (Exception e)
                                {e.printStackTrace(); break;}

                        availableBitsToEmbed=8;
                }
                nextBitToEmbed = byteToEmbed & 1;
                byteToEmbed >>= 1;
                availableBitsToEmbed--;
                kBitsToEmbed |= nextBitToEmbed << i;
        }
```

b) Llenar un buffer con *n* coeficientes de la imagen diferentes de 0. Comprobar también que no se usa el segundo 1 o -1.

```
        do
        {
                j = startOfN;
                one = startOne;
                for (i = 0; i < n; j++)
                {
                        if (j >= coeffCount)
                        {
                                // in rare cases the estimated capacity is too
                                small
                                System.out.println("Capacity exhausted.");
                                break embeddingLoop;
                        }
                        shuffledIndex = permutation.getShuffled(j);
[...]/[...]
```

```
[...]/[...]
                        // skip zeroes
                        if (imageSamplesArray[shuffledIndex] == 0) continue;
                        // rev. 12: skip every second 1 or -1
                        if (Math.abs(imageSamplesArray[shuffledIndex]) == 1)
                        {
                                if ((++one &   1) == 0) continue;
                        }
                        codeWord[i++]=shuffledIndex;
                  }
```

c) Generar un valor hash de *k* bits. Añadir los siguientes *k* bits del mensaje al hash con una xor.

```
                endOfN = j;
                hash = 0;
                for (i = 0; i < n; i++)
                {
                        if (imageSamplesArray[codeWord[i]] > 0)
                        {
                                extractedBit =
                                        imageSamplesArray[codeWord[i]] & 1;
                        }else
                        {
                                extractedBit =
                                        1 - (imageSamplesArray[codeWord[i]] & 1);
                        }
                        if (extractedBit == 1)
                        {
                                hash ^= i + 1;
                        }
                  }
```

d) Si la suma es cero, el buffer permanece inalterado. Si no, sumamos o restamos uno en esa posición de la imagen dependiendo si el coeficiente es positivo o negativo, para así decrementar el valor absoluto.

```
                i = hash ^ kBitsToEmbed;
                if (i == 0) break; // embedded without change
                i--;
                if (imageSamplesArray[codeWord[i]] > 0)
                {
                        imageSamplesArray[codeWord[i]]--;
                } else
                {
                        imageSamplesArray[codeWord[i]]++;
                }
```

e) Comprobar durante el proceso que no hemos producido un cero. Si lo hemos hecho, ajustar el buffer eliminando el cero, es decir, repetir el paso (6) a) desde el mismo coeficiente. Si no hemos producido ningún cero continuar con los siguientes coeficientes del buffer. Si todavía queda mensaje para incrustar continuamos desde el paso (6) a).

```
                if (imageSamplesArray[codeWord[i]]==0)
                {
                        _thrown++;
                }
        } while (true);
        startOfN = endOfN;
        startOne = one;
} while (!isLastByte);
```

(7) Continuar con la compresión JPEG2000 una vez los coeficientes modificados han sido incorporados a la imagen.

Este es el proceso a seguir cuando $n > 1$. Cuando $n = 1$, para incrustar el mensaje se utilizará un LSB modificado, es decir, que incrusta los bits como si fuera LSB, pero tiene en cuenta la permutación utilizada en F5 y comprueba que los coeficientes sean diferentes de 0 y comprueba también que no se utilice el segundo 1 o -1.

La extracción del mensaje secreto se hace de forma equivalente al método para incrustar los datos. Extraemos la longitud del mensaje y *k* de la palabra de estado. Con esa información y el array de coeficientes de la imagen leemos *n* posiciones del array, que vendrán dadas por la permutación de F5, para así poder extraer los *k* bits que fueron incrustados en esas posiciones.

```
extractingLoop:
      do
      {
            // 1. read n places, and calculate k bits
            hash = 0;
            int code = 1;
            for (i = 0; code <= n; i++)
            {
                  // check for pending end of imageSamplesArray
                  if (startOfN + i >= coeffCount) break extractingLoop;
                  shuffledIndex = permutation.getShuffled(startOfN + i);
                  // skip zeroes
                  if (imageSamplesArray[shuffledIndex] == 0) continue;
                  // rev. 12: skip every second 1 or -1
                  if (Math.abs(imageSamplesArray[shuffledIndex]) == 1)
                        if ((++one & 1) == 0) continue;
                  if (imageSamplesArray[shuffledIndex] > 0)
                      extractedBit = imageSamplesArray[shuffledIndex] & 1;
                  else
                      extractedBit=1-(imageSamplesArray[shuffledIndex] &1);
                  if (extractedBit == 1)
                  {
                        hash ^= code;
                  }
                  code++;
            }
            startOfN += i;
```

Una vez tenemos los *k* bits que fueron incrustados sólo nos falta agruparlos en bytes y escribirlos en el fichero de texto.

```
        // 2. write k bits bytewise
        for (i = 0; i < k; i++)
        {
                extractedByte |= ((hash >> i) & 1) << availableExtractedBits++;
                if (availableExtractedBits == 8)
                {
                        // remove pseudo random pad
                        extractedByte ^= random.getNextByte();
                        fos.write((byte) extractedByte);
                        extractedByte = 0;
                        availableExtractedBits = 0;
                        nBytesExtracted++;
                        // check for pending end of embedded data
                        if (nBytesExtracted == extractedFileLength)
                                break extractingLoop;
                }
        }
} while (true);
```

La gráfica resultante de aplicar el método a diferentes textos de tal forma que hubiera una muestra de cada código desde (1,1,1) hasta (1,1023,10) la podemos ver en la siguiente página.

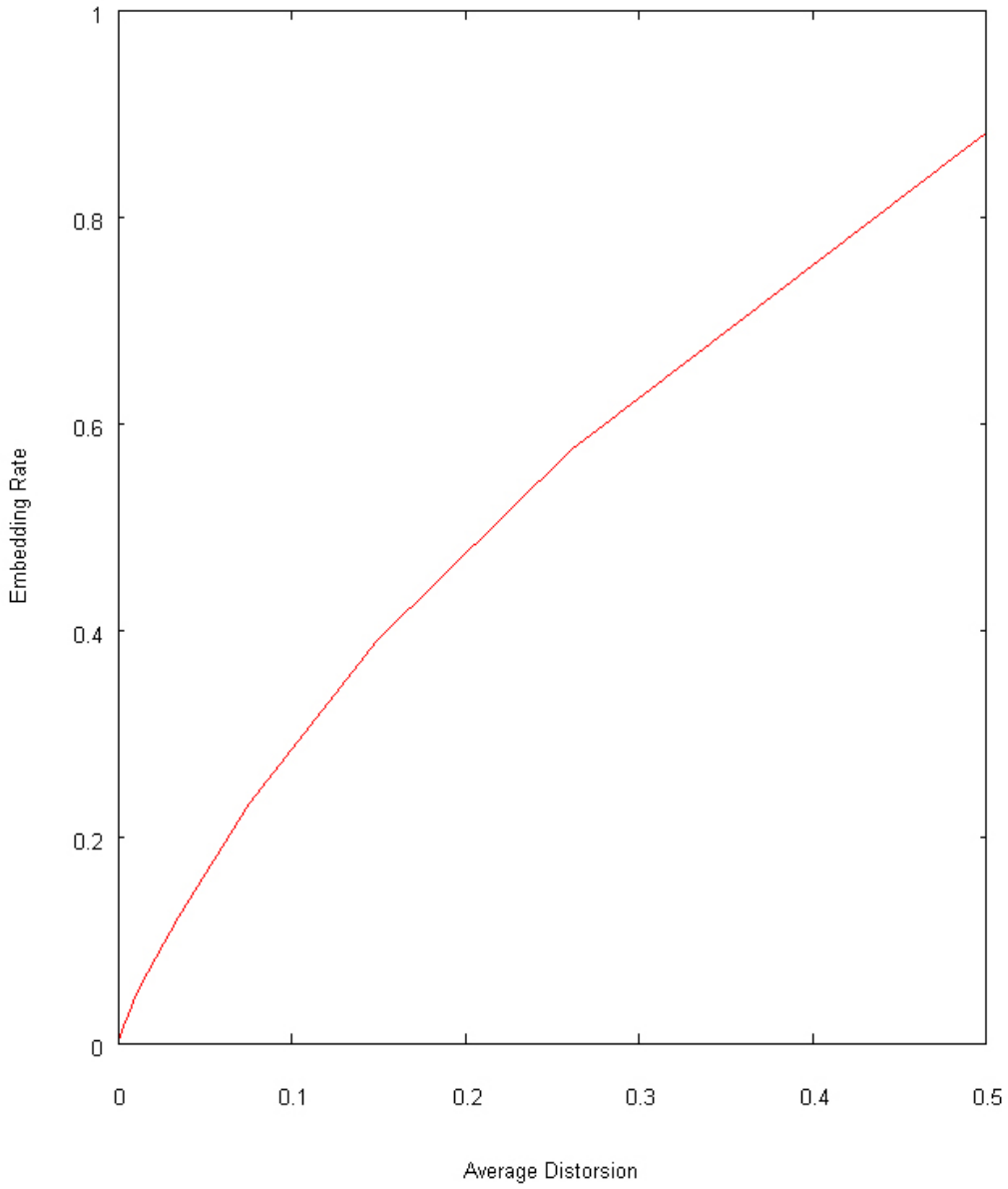

Fig. 9: Rendimiento del F5

4.4.- PPC (Product Perfect Codes)

El tercer método del proyecto es el presentado en [13], un protocolo esteganográfico eficaz basado en códigos de Hamming cuyos algoritmos de incrustación y extracción tienen el mismo coste computacional que el F5. El método propuesto usa el producto de códigos perfectos lineales, pero no usa el Algoritmo de Clases en todo el código producto.

Como se muestra en [8], los protocolos esteganográficos basados en códigos de Hamming son muy eficaces, pero los parámetros para la media de distorsión $D = \frac{R_a}{n}$, deben de tomarse en un rango estrecho. Para códigos perfectos es fácil de calcular la media de distorsión teniendo en cuenta que $R_a = \frac{1}{q^t}\sum_{i=1}^{p} a_i \cdot i$, donde $a_i = \binom{n}{i}$. Para encontrar un protocolo con una media de distorsión $D = \frac{R}{n}$ diferente a la dada por un código de Hamming podemos encontrar dos códigos de Hamming con distorsiones medias $\frac{R_a}{2^t - 1}$ y $\frac{R'_a}{2^{t+1} - 1}$ tal que $\frac{R}{n} = \gamma \cdot \frac{Ra}{2^t - 1} + (1 - \gamma) \cdot \frac{R'_a}{2^{t+1} - 1}$ donde $0 \le \gamma \le 1$. Podemos usar ambos códigos de Hamming, en el rango dado por γ, para esconder información.

La idea presentada en [13] para mejorar el protocolo recién presentado es usar un código producto de dos códigos de Hamming. Dados dos códigos de Hamming, C_1 de longitud $n_1 = 2^x - 1$ y C_2 de longitud $n_2 = 2^y - 1$, el código producto $C_1 \times C_2$ es el código de longitud $n = (2^x - 1)(2^y - 1)$, dimensión $k = n - t = (n_1 - x)(n_2 - y)$ y con la peculiaridad de que sus palabras código pueden verse como matrices $(2^y - 1) \times (2^x - 1)$, donde las columnas son palabras código de C_1 y las columnas palabras código de C_2.

El protocolo presentado se basa en tres propiedades:

Lema 1: Sea C un código Hamming de longitud $2^t - 1$. Sea $i,j \in \{1, ..., 2^t - 1\}$. Dados estos elementos, existe una coordenada única $g_1(i,j) \in \{1, ..., 2^t - 1\}$ tal que el vector de soporte $\{i,j,g_1(i,j)\}$ pertenece a C.

Demostración: Es directa ya que C es un código perfecto.

La coordenada g_1 la calculamos con la siguiente función:

```
private int g1(int a, int b, int n)
{
      int c = -1;
      for (int i = 1; i <= n; i++)
      {
            if (((i^a^b) == 0))
            {
                  c = i;
                  break;
            }
      }
      return c;
}
```

Lema 2: Sea C un código Hamming de longitud $2^t - 1$. Siempre es posible tener una matriz de control tal que para cualquier coordenada $1 \leq i \leq 2^{t-1} - 1$ existen dos coordenadas, especificamente la $(2^{t-1} - 1 + i)$ y la $(2^t - 1)$ tal que el vector v con soporte $\{i, g_2(i), 2^t - 1\}$ pertenece al código C, donde $g_2(i) = 2^{t-1} - 1 + i$.

Demostración: Tomamos H_t, la matriz de control $(t \times 2^t - 1)$ recursivamente para el código de longitud $2^t - 1$:

$$H_t = \begin{pmatrix} 0 \cdots 0 & 1 \cdots 1 & 1 \\ H_{t-1} & H_{t-1} & 0 \end{pmatrix}$$

donde podemos empezar la secuencia con la matriz de control para el código de longitud 2^2-1:

$$H_2 = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

Lema 3: Sea C un código Hamming de longitud $n = 2^t - 1$ con una matriz de control H del tipo del Lema 2. Sea $i,j \in \{1, ..., 2^t - 1\}$ con $i < j < r$ tal que el vector x con soporte $\{i, j, r\}$ pertenece al código C. Entonces, el vector y con soporte $\{j, g_2(i), g_2(r)\}$ pertenece al código C y $g_2(i), g_2(r) \in \{2^{t-1}, ..., 2^{t-2}\}$.

Demostración: Por el Lema 2, los vectores x_1, x_2 con soporte $\{i, g_2(i), 2^t - 1\}$ y $\{r, g_2(r), 2^t - 1\}$, respectivamente, pertenecen a C y $g_2(i), g_2(r) \in \{2^{t-1}, ..., 2^{t-2}\}$.

Por tanto, el vector $y = x + x_1 + x_2$ pertenece a C y tiene el soporte deseado.

Dados dos códigos de Hamming C_1 y C_2 de longitudes $n_1 = 2^x - 1$ y $n_2 = 2^y - 1$, respectivamente, el protocolo esteganográfico se define de la siguiente manera:

(1) Usamos el F5 para procesar las filas. Este F5 ha sido modificado para el correcto funcionamiento del PPC. No se usa la permutación de los coeficientes de la imagen, simplemente se van tomando de manera secuencial. Según vamos incrustando los bits del mensaje en la imagen, los guardamos en una matriz de tamaño $n_1 \times n_2$ para así poder tratar las columnas después. También guardamos la posición que hemos cambiado en un array ya que la necesitaremos más adelante.

```
for (l = 0; l < codeWord.length; l++)
{
      ramCodeMatrix[l][m] = imageSamplesArray[codeWord[l]];
      positionCheck[l][m] = codeWord[l];
}
```

(2) Una vez hemos llenado la matriz con coeficientes de la imagen llamamos a la función que se encarga de procesar las columnas. A esta función le pasamos la matriz con los coeficientes de la imagen modificados, el número de columna que procesaremos, la dimensión del código de longitud n_2 (la variable k2), la información para incrustar en las columnas y el array con las posiciones cambiadas en las filas.

```
RamCodeMatrix = columnEmbedding(ramCodeMatrix,c,k2,kBitsToEmbed,positionChanged);
```

(3) Procesamos las columnas recursivamente empezando por la primera columna hasta la columna $2^{x-1} - 1$-ésima. Aplicamos el mismo algoritmo que hemos usado en las filas en la columna i-ésima para encontrar una coordenada c_i para cambiar y así incrustar los bits del mensaje secreto. Pueden darse 6 casos:

a) La coordenada c_i no existe, porque la información de la columna coincide con los bits que queremos incrustar, y no hacemos nada.

b) La coordenada c_i coincide con r_j (coordenada que obtuvimos cuando procesamos la fila *j*-ésima). En este caso debemos cambiar la coordenada $c_i = r_j$ y para evitar efectos no deseados en la fila *j*-ésima, también cambiamos las coordenadas $2^x\text{-}1$ y $g_1(c_i, 2^x\text{-}1)$. Este caso aumenta en uno la cantidad de bits incrustados.

```
if (positionChanged[i] == c)
{
      ramCodeMatrix[c][i] = embed(ramCodeMatrix[c][i]);
      ramCodeMatrix[g1b-1][i] = embed(ramCodeMatrix[g1b-1][i]);
      g1c = g1(c+1,g1b,n1);
      ramCodeMatrix[g1c-1][i] = embed(ramCodeMatrix[g1c-1][i]);
}
```

c) La coordenada c_i se encuentra en la fila *j*-ésima y en esta fila no hay ningún elemento r_j procesado anteriormente. En este caso cambiamos la coordenada c_i y para evitar efectos no deseados en la fila *j*-ésima, también cambiamos las coordenadas $2^x\text{-}1$ y $g_1(c_i, 2^x\text{-}1)$. Este caso aumenta en tres la cantidad de bits incrustados.

```
if (positionChanged[i] == -1)
{
      ramCodeMatrix[c][i] = embed(ramCodeMatrix[c][i]);
      ramCodeMatrix[g1b-1][i] = embed(ramCodeMatrix[g1b-1][i]);
      g1c = g1(c+1,g1b,n1);
      ramCodeMatrix[g1c-1][i] = embed(ramCodeMatrix[g1c-1][i]);
}
```

d) La coordenada c_i se encuentra en la fila *j*-ésima, en esta fila hay un elemento r_j procesado anteriormente y $r_j < c_i$. En este caso cambiamos la coordenada c_i y para evitar efectos no deseados en la fila *j*-ésima, también cambiamos las coordenadas $2^x\text{-}1$ y $g_1(c_i, 2^x\text{-}1)$. Este caso aumenta en tres la cantidad de bits incrustados.

```
if (positionChanged[i] < c)
{
      ramCodeMatrix[c][i] = embed(ramCodeMatrix[c][i]);
      ramCodeMatrix[g1b-1][i] = embed(ramCodeMatrix[g1b-1][i]);
      g1c = g1(c+1,g1b,n1);
      ramCodeMatrix[g1c-1][i] = embed(ramCodeMatrix[g1c-1][i]);
}
```

e) La coordenada c_i se encuentra en la fila j-ésima, en esta fila hay un elemento r_j procesado anteriormente, $r_j > c_i$ con $g_1(r_j, c_i) > c_i$. En este caso cambiamos la coordenada c_i y para evitar efectos no deseados en la fila j-ésima, también cambiamos las coordenadas r_j y $g_1(r_j, c_i)$. Este caso aumenta en uno la cantidad de bits incrustados. Calcularemos primero la coordenada $g_1(r_j, c_i)$ ya que la necesitamos para hacer la comparación y determinar que estamos en este caso.

```
 if (positionChanged[i] != -1)
 {
       g1sit4 = g1(positionChanged[i]+1,c+1,n1);
       g1sit4--;
 }
[...]/[...]
 if( (positionChanged[i] > c) && (g1sit4 > c) )
 {
       ramCodeMatrix[c][i] = embed(ramCodeMatrix[c][i]);
       ramCodeMatrix[positionChanged[i]][i] =
                              embed(ramCodeMatrix[positionChanged[i]][i]);
       ramCodeMatrix[g1sit4][i] = embed(ramCodeMatrix[g1sit4][i]);
 }
```

f) La coordenada c_i se encuentra en la fila j-ésima, en esta fila hay un elemento r_j procesado anteriormente, $r_j > c_i$ con $g_1(r_j, c_i) < c_i$. En este caso cambiamos la coordenada c_i y para evitar efectos no deseados en la fila j-ésima, también cambiamos las coordenadas r_j, $g_2(c_i)$ y $g_2(r_j)$. Este caso aumenta en dos la cantidad de bits incrustados.

```
 if( (positionChanged[i] > c) && (g1sit4 < c) )
 {
       ramCodeMatrix[c][i] = embed(ramCodeMatrix[c][i]);
       ramCodeMatrix[positionChanged[i]][i]
                              =embed(ramCodeMatrix[positionChanged[i]][i]);
       int g2g1 = g1(c+1,g1b,n1);
       ramCodeMatrix[g2g1-1][i] = embed(ramCodeMatrix[g2g1-1][i]);
       int g2rj = g1(positionChanged[i]+1,g1b,n1);
       ramCodeMatrix[g2rj-1][i] = embed(ramCodeMatrix[g2rj-1][i]);
 }
```

(4) Cuando se acaba con la primera columna, aumentamos el número de columna, guardamos los valores de la matriz temporal en el array de coeficientes de la imagen, comprobando que no hemos producido ceros y volvemos al principio para coger más bits del mensaje secreto, en el caso de que sea necesario.

```
c++;


for(int j1 = 0; j1 < n2; j1++)
{
      for (int i1 = 0; i1 < n1; i1++)
      {
            imageSamplesArray[positionCheck[i1][j1]] =
                                                ramCodeMatrix[i1][j1];
            if (imageSamplesArray[positionCheck[i1][j1]] == 0
                                                      && (even == 0))
            {
                  imageSamplesArray[positionCheck[i1][j1]] =
                        imageSamplesArray[positionCheck[i1][j1]] + 2;
                  even = 1;
            }else if (imageSamplesArray[positionCheck[i1][j1]] == 0
                                                      && (even == 1))
            {
                  imageSamplesArray[positionCheck[i1][j1]] =
                        imageSamplesArray[positionCheck[i1][j1]] - 2;
                  even = 0;
            }
      }
}
```

(5) Comprobamos que el número de columna sea igual a $2^{x-1} - 1$ y si es así reiniciamos la matriz que usamos para guardar los coeficientes y repetimos el proceso desde el punto (1).

```
if (c == (cEnd))
{
      _columns = 0; m = 0; c = 0;
      for(int j1 = 0; j1 < n2; j1++)
            for (int i1 = 0; i1 < n1; i1++)
            {
                  ramCodeMatrix[i1][j1] = 0; positionCheck[i1][j1] = 0;
            }
}
```

(6) Cuando todo el mensaje ha sido incrustado, ponemos los coeficientes del array en la imagen y seguimos con el proceso de JPEG2000.

El valor de n_1 se calcula automáticamente como en el F5. Si $n_1 = 1$ no se utiliza el algoritmo de PPC sino el mismo que se utilizaba en el F5 cuando $n = 1$, pero sin utilizar la permutación.

El valor de n_2 se calcula a partir de k_2, que se pide al usuario antes de empezar el proceso.

La extracción del mensaje se hace de forma equivalente a la utilizada en el F5, pero teniendo en cuenta de que hay que extraer información que fue escondida en las columnas. Por tanto deberemos crear la matriz con los mismos coeficientes y una vez extraída la información de las filas, pasaremos a extraerla de las columnas.

```
 hash = 0;
 int code = 1;
 if (_columns == 0)
 {
      [...]/[...] //F5 wise extraction
 } else
 {
      code = 1;
      for (i=0; i < n2; i++)
            columnWord[i] = ramCodeMatrix[c][i];
      for (i=0; code <= n2; i++)
      {
            extractedBit = columnWord[i]&1;
            if (extractedBit == 1)
                  hash ^= code;
            code++;
      }
      c++;
 }
[...]/[...]
 if (m == n2) _columns = 1;
 if (c == cEnd)
 {
      _columns = 0; m = 0; c = 0;
 }
```

Por tanto, se van extrayendo bits, ya sea de fila o columnas, hasta que se pueden agrupar en bytes y es entonces cuando se escriben en el fichero de texto.

La gráfica resultante de aplicar el método a diferentes textos a las diferentes parejas de códigos usadas es la siguiente.

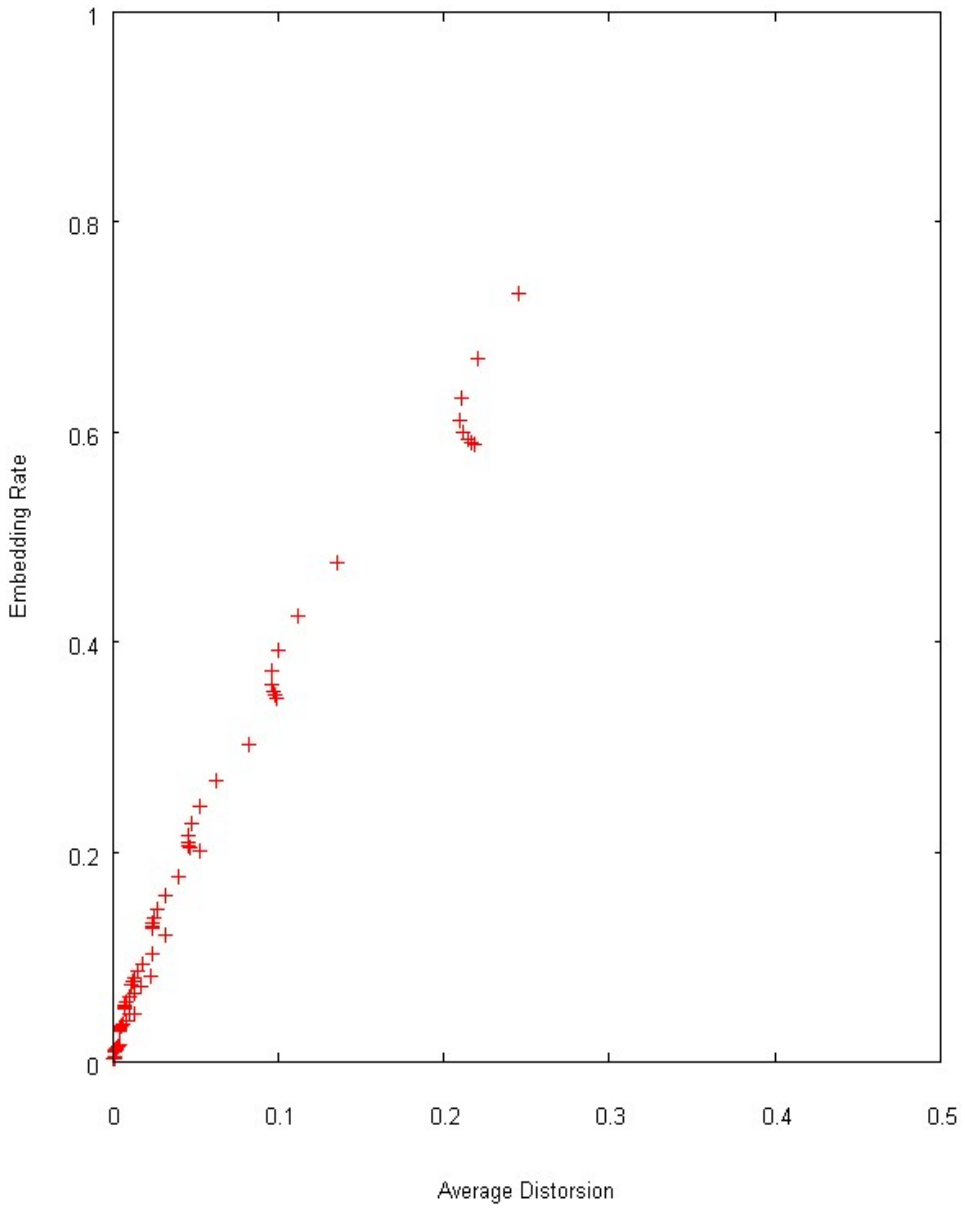

Fig. 10: Rendimiento del PPC

5.- Conclusiones y Ampliaciones

Como ya ha sido mencionado anteriormente, el objetivo de este proyecto es comparar de forma práctica el rendimiento del protocolo esteganográfico presentado en [13]. Para eso usaremos las gráficas del rendimiento de los protocolos y las sobrepondremos en una sola gráfica:

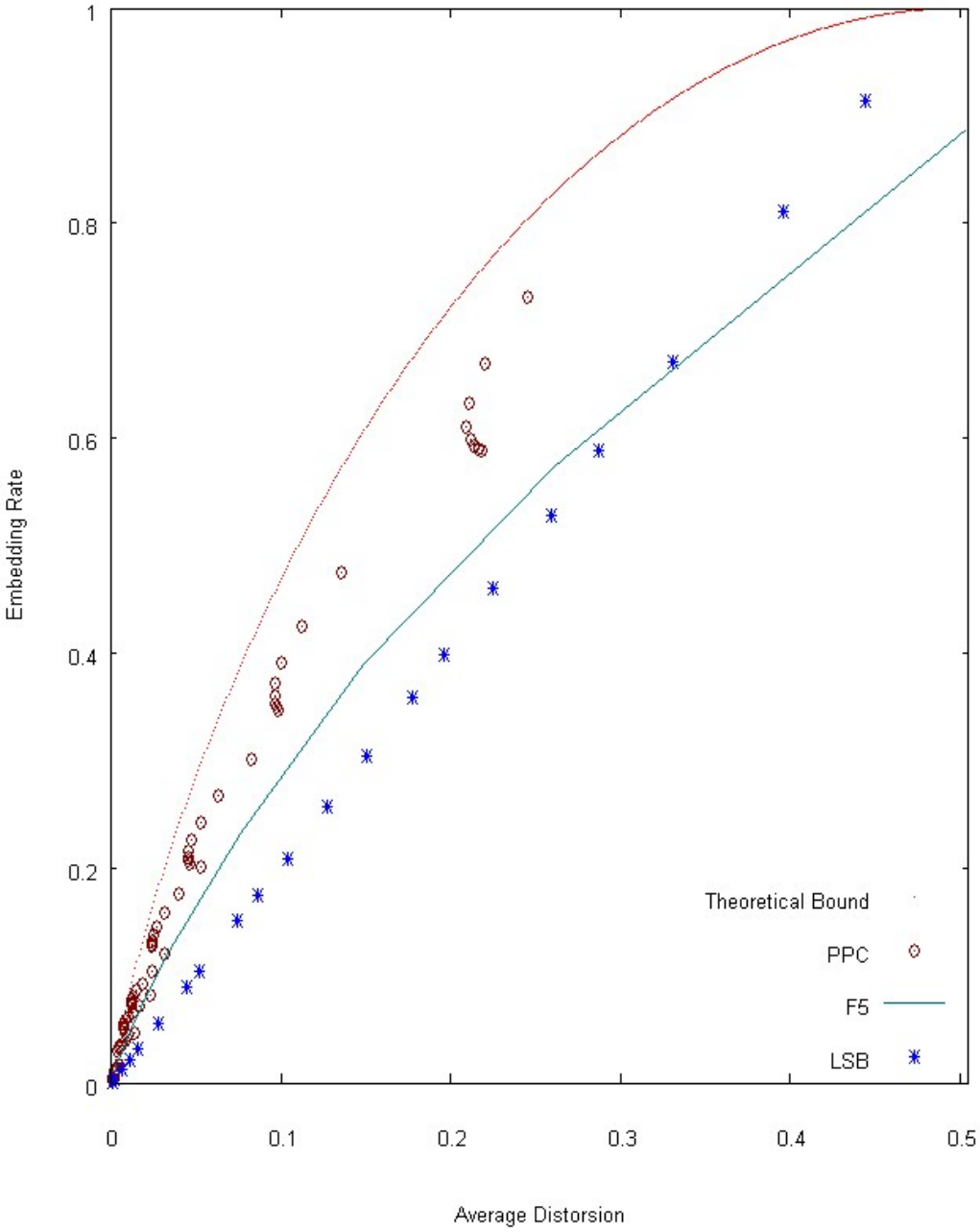

Fig. 11: Rendimiento de los diferentes protocolos esteganográficos

Como podemos ver en la fig. 11 para un valor fijo de *x* e *y* el rendimiento del PPC es mayor, en la mayoría de los casos, que el rendimiento del F5 y el LSB.

También se puede observar lo que ya explicaba A. Wesntfeld en [8]: que el rendimiento de F5 es mayor que el de LSB cuando la cantidad de información a esconder es menor ya que permite realizar menos cambios. Pero debido a que en el LSB implementado no se descarta ningún coeficiente, la capacidad es mayor y por eso en los valores más altos de la gráfica LSB supera a F5.

Pero como en todos los proyectos, siempre queda espacio para mejorar. Estas son algunas de las ampliaciones que podrían implementarse:

- Comprobar el número de canales de la imagen utilizada. Si la imagen es en escala de grises, sólo tiene un canal. Pero si la imagen es a color tiene tres canales que podrían ser aprovechados para esconder más información, ya que tal y como esta implementado sólo utiliza uno de esos canales.
- Para combatir ataques estadísticos se podría implementar también la reorganización de los coeficientes de la imagen con la permutación usada en F5.
- También se podría cifrar el mensaje secreto justo antes de incrustarlo en la imagen, ya fuera con un cifrado de clave pública o de clave privada, para que así fuera más difícil recuperar el mensaje en el caso de que fuera interceptado.

6.- Bibliografía

[1] GJ Simmons: “The Prisoners' Problem and the Subliminal Channel”, LNCS 169, Springer-Verlag, 1984, pp. 51-67.

[2] C. Cachin: “An information-theoretic model for steganography”, in D. Aucsmith (ed.): Information Hiding. 2nd International Workshop, LNCS vol. 1525, Springer-Verlag Berlin Heidelberg (1998), 306-318.

[3] D. Taubman, “High performance scalable image compression with EBCOT”, presented at the Proc. IEEE International Conference on Image Processing, vol. 3, Oct. 1999, pp. 344–348.

[4] Francesc Aulí: “Model-based JPEG2000 rate control methods”, Phdthesis, Universitat Autònoma de Barcelona, Escola Tècnica Superior d'Enginyeria, 2006.

[5] GICI group: “BOI User manual”, Department of Information and Communications Engineering, Universitat Autònoma Barcelona, September 2005. Available online at: http://www.gici.uab.es/BOI/

[6] P. Moulin, Y. Wang: "New results on steganographic capacity," Proceeding of CISS 2004. University of Princeton, Princeton, New Jersey, 2004

[7] R. Crandall: “Some notes on steganography”, posted on Steganography Mailing List, http://os.inf.tu-dresden.de/~westfeld/crandall.pdf, 1998.

[8] A. Westfeld: "High Capacity Despite Better Steganalysis (F5 - A Steganographic Algorithm)", In: LNCS, vol. 2137, Springer-Verlag, New York, pp. 2001, 289-302

[9] C. Munuera: “Steganography and error-correcting codes”, Signal Processing 87 (2007) pp.1528-1533.

[10] W. Zhang, S. Li: “A Coding Problem in Steganography”, Designs, Codes and Cryptography, Springer Netherlands, Vol 46, Number 1, pp. 67-81, Jan. 2008.

[11] J. Bierbrauer, J. Fridrich: “Constructing good covering codes for applications in steganography”, available, http://www.math.mtu.edu/jbierbra/, 2006.

[12] J. Fridrich, P.Lisoněk, D. Soukal: “On steganographic embedding eficiency”, Springer-Verlag, LNCS 4437, pp. 282-296, 2007

[13] H. Rifà-Pous, J. Rifà: “Product Perfect Codes and Steganography”, Nov. 2007. To appear in Digital Signal Processing, Elsevier. 2008.

RESUMEN

En este proyecto se implementan tres algoritmos esteganográficos diferentes usando JPEG2000 como portador del mensaje, se calcula el rendimiento de cada uno de ellos y se comparan usando una gráfica. El objetivo es visualizar para unos casos específicos que el algoritmo basado en el producto de dos códigos lineales perfectos tiene mejor rendimiento que el obtenido con algoritmos como el F5 y el LSB.

RESUM

En aquest projecte s'implementen tres algorismes esteganogràfics diferents utilitzant JPEG2000 com a portador del missatge, es calcula el rendiment de cada un d'ells i es comparan mitjançant una gràfica. L'objectiu és visualitzar per a uns casos específics que l'algorisme basat en el producte de dos codis lineals perfectes té mijor rendiment que l'obtingut amb algorismes com l'F5 i l'LSB.

ABSTRACT

In this project three steganographic algorithms are implemented using JPEG2000 as cover medium for the secret message, performance of each of them are calculated and compared using a graphic. The aim of the project is to visualize that when some specific conditions are met, the algorithm based on product perfect codes has better performance than the one achieved with algorithms like F5 and LSB.