Introducción al Modelado del Continuo

Trabajo Práctico $n°$ 2 : Compresión de Imágenes.

Vamos a implementar una versión apenas simplificada del algoritmo de codificación de los archivos `.jpg`, que se basa en una propiedad central que vimos en la transformada de Fourier: la transformada de señales de la vida real suele estar formada fundamentalmente por frecuencias bajas, con muy poca contribución de las frecuencias altas. Esto induce la siguiente idea: dada una señal podemos considerar su transformada y descartar todas las frecuencias por encima de un cierto umbral. Esto permite almacenar sólo un pequeño número de frecuencias, que forman la señal *comprimida*. Para recuerar la señal lo que debe hacerse es volver a completar el vector transformado con ceros y anti-transformar.

La idea del algoritmo `jpg` es esencialmente esa, pero incluye también algunos otros trucos. En primer lugar se utiliza la transformada del coseno en lugar de la de Fourier. Las ventajas de la transformada del coseno para tareas de compresión son dos: en primer lugar es una transformada que convierte reales en reales, lo que evita el paso por los complejos (que implican almacenar dos flotantes por cada número). En segundo lugar, se observa en la práctica que en la descomposición dada por la transformada del coseno hay aún mayor concentración de información en las frecuencias bajas, lo que permite descartar más valores. La transformada del coseno está implementada en la librería `FFTW`, en el comando `dct` (discrete cosine transform).

El algoritmo

Preparación

Nuestro algoritmo asumirá que las dimensiones de la imagen son divisibles por 16. Para simplificar la implementación, haremos un primer paso que consistirá en rellenar los márgenes de la imagen con negro hasta lograr que ambas dimensiones sean múltiplos de 16. Por ejemplo: si un imagen tiene 37 píxeles de ancho queremos agregarle una banda de 5 píxeles negros a la izquierda y otra de 6 a la derecha, hasta tener 48 píxeles. Este proceso no es óptimo y puede generar artefactos indeseados en la imagen comprimida, pero es la variante más simple.

Implementar una función que realice este proceso.

Primera etapa

En algoritmo trabaja sobre la descomposición `YCbCr`, por lo cual lo primero que hay que hacer es convertir la imagen a este formato. Además, necesitamos separar los tres canales para operar sobre ellos por separado. Esto se logra con el comando `channelview` que convierte una imagen de `n×m` pixels en un arreglo `A` de `3×n×m`. De esta manera `A[1,:,:]` corresponderá al canal `Y`, etc.

Finalmente: dado que el ojo humano es mucho más sensible a la luminosidad que a la intensidad de color, podemos reducir las matrices `Cb` y `Cr`. Para ellos generaremos nuevas matrices de tamaño `n/2×m/2` en las que cada pixel sea el promedio de `4` pixels vecinos en la matriz original. Por último, haremos un corrimiento en los coeficientes de todas las matrices para que éstos queden centrados en 0. Dado que nuestra codificación arrojará valores entre 0 y 255, lo que hacemos es restar 128 en cada casillero.

Implementar una función que reciba una imagen y realice todo este proceso, devolviendo una matriz numérica `Y` de `n×m` y matrices numéricas `Cb`y `Cr` de tamaño la mitad en cada dimensión.

Implementar también el proceso inverso que consiste en, dadas tres matrices como las anteriores, sumarles 128, ampliar `Cb` y `Cr` (generando `4` pixels iguales por cada pixel original), reensamblarlas en una imagen `YCbCr` y finalmente convertirla a `RGB`. Aquí es necesario el comando `colorview` que recibe un tipo de dato (en este caso `YCbCr`) y un arreglo de `3×n×m` y devuelve la imagen.

Transformada por bloques

El siguiente paso consiste en pensar a cada matriz como una agrupación de bloques de `8×8` y en cada uno de estos bloques aplicar la transformada del coseno discreta (`dct`). El resultado de esta etapa son las 3 matrices transformadas por bloques. Para ahorrar memoria pueden utilizarse los comandos `view` (que permite acceder y modificar en el lugar secciones de matrices) y `dct!` (que aplica la transformada modificando su argumento).

Implementar esta función y su inversa, que debe aplicar `idct` (o `idct!`) por bloques.

Cuantización

En esta etapa se realiza la compresión más importante. Esencialmente, se trata de descartar las frecuencias altas en cada bloque de `8×8`. Esto se hace a través de una matriz de *cuantización*. la `dct` descompone en frecuencias positivas, por lo cual al transformar una de nuestras matrices la frecuencia `0` se encuentra en el casillero `[1,1]` y la frecuencia más grande en el casillero `[8,8]`. Una matriz de cuantización será una matriz de enteros fija de `8×8`, simétrica, que tendrá típicamente valores chicos en el casillero `[1,1]` y valores cada vez más altos a medida que nos aproximamos al casillero `[8,8]`. La siguiente matriz es un posible ejemplo:

```
quant=[16 11 10 16 24 40 51 61;
       12 12 14 19 26 58 60 55;
       14 13 16 24 40 57 69 56;
       14 17 22 29 51 87 80 62;
       18 22 37 56 68 109 103 77;
       24 35 55 64 81 104 113 92;
       49 64 78 87 103 121 120 101;
       72 92 95 98 112 100 103 99]
```

El proceso de cuantización consiste en tomar cada bloque de cada una de las tres matrices de la imagen y dividirlo casillero a casillero por la matriz de cuantización, redondeando el resultado.

Implementar una función que realice el proceso de cuantización sobre una matriz.

Implementar también el proceso inverso, que consiste en multiplicar por la matriz de cuantización casillero a casillero.

Observar que al aplicar la cuantización y su inversa el resultado es que se convierten en `0` muchos de los valores, pero los otros se preservan aproximadamente iguales (salvo error de redondeo).

Compresión

Finalmente, llegamos al momento de descartar los ceros de los bloques transformados. Para ello aplicaremos el siguiente procedimiento:

1. Leeremos cada bloque en zig-zag, convirtiéndolo en un vector. El orden de lectura es:

   `[1,1],[1,2],[2,1],[3,1],[2,2],[1,3],[1,4],...,[8,6],[7,7],[6,8],[7,8],[8,7],[8,8]`

   Observar que de esta manera tendremos que casi siempre la cola del vector está formada por ceros. Vale la pena tener en cuenta que Julia admite indexar una matriz como si fuera un vector, en cuyo caso la lee por columnas.

2. El vector resultante lo comprimiremos haciendo uso del método *Run Length Encoding* que consiste en indicar la cantidad de veces consecutivas que se repite un número, y el número. De esta manera, la tira `[3,3,3,3,3]` se codificaría con un `3` y un `5` (dos números en lugar de cinco). Felizmente esto está implementado en la librería `StatsBase` en la función `rle`. Probar el código:

   ```
   vector_test = [1,1,1,1,0,0,1,1,0,0,0,0,2,0,0,0,0]
   vals,reps   = rle(vector_test)
   ```

   También existe la inversa:

   ```
   inverse_rle(reps,vals)
   ```

   Por último, haremos un largo vector en el que almacenaremos todos estos números. Observar que tendremos un par de vectores (repeticiones y valores) por cada bloque de cada matriz. Podemos generar un vector de vectores de la forma `[reps`$_1$`,vals`$_1$`,reps`$_2$`,vals`$_2$`,...]`, o directamente poner todos los números en una sola tira. En el siguiente paso grabaremos esto en un archivo, y allí no habrá vectores sino sólo una larga tira de números.

   Puede resultar más simple hacer una función que procese sólo una matriz y genere la tira de datos que le corresponde y luego otra que simplemente concatene las tres tiras.

Implementar la función que genera este vector.

Implementar también su inversa que debe tomar la larga tira de datos y separarla en 3 trozos de tamaño adecuado, y cada trozo en parejas (repeticiones y valores) por bloque, que luego deben reensamblarse (con `inverse_rle`) y acomodarse en la matriz correspondiente.

Guardado

Por último, queremos almacenar nuestra tira de datos en un archivo que representará el formato comprimido.

Para esto resultarán útiles los siguientes comandos:

1. Para abrir un archivo (y crearlo si no existe):

   ```
   io = open("nombre.ext","w")
   ```

   Esto genera una variable `io` de tipo `IOStream` (input-output stream), que contiene todo lo que tiene el archivo. Si se quiere abrir el archivo sólo para leerlo, se omite la `"w"`.
2. Para escribir un número `num` sobre el archivo abierto:

   ```
   write(io,num)
   ```
3. Para cerrar el archivo (esto es muy importante para que el archivo se guarde):

   ```
   close(io)
   ```
4. Para leer hay varias formas. El comando `read` aplicado a `io` devolverá un vector con todos los bytes en formato hexadecimal. Más práctico es utilizar `read` indicando el tipo de dato que se espera obtener. Esto leerá la cantidad necesaria de bytes y devolverá el tipo de dato deseado. Además, el *cabezal de lectura* queda ubicado en la última posición por lo cual la siguiente aplicación de `read` leerá los siguientes bytes. Por ejemplo:

   ```
   a = read(io,Int64)
   ```

   lee 8 bytes y los devuelve como número entero, que se guarda en `a`. Si luego se ejecuta:

   ```
   b = read(io,Int8)
   ```

   se leerá el noveno byte del archivo como un entero y se lo guardará en `b`.

Nota sobre tipos:
Una imagen en formato `RGB` típicamente se codifica en enteros entre 0 y 255, cuyo formato es `UInt8` (unsigned integer, 8 bits). Julia utiliza el formato (equivalente) `N0f8` que representa flotantes entre 0 y 1 de la forma $\frac{k}{255}$ con $k = 0, \ldots, 255$. Esto es importante porque si se utilizaran flotantes de tipo `Float64` o enteros `Int64` cada número ocuparía 8 bytes en lugar de 1, y el tamaño total del archivo sería 8 veces el necesario.

Para generar nuestro pseudo-jpg debemos tener esto en cuenta. Los datos del `RGB` vendrán en `N0f8`, que se convertirán en flotantes entre 0 y 255 al pasar a `YCbCr`. Al restar 128, pasaremos al rango entre -128 y 127, pero seguiremos teniendo flotantes. Podríamos redondearlos y pasarlos a `Int8`. Sin embargo, al aplicar `dct` obtendremos flotantes en un rango mayor. Este proceso típicamente se revierte al aplicar la cuantización. Los valores redondeados de la matriz cuantizada pueden almacenarse como `Int8`, ocupando sólo 1 byte cada uno. El objetivo es guardarlos de esta manera en el archivo.

Implementar una función que guarde en un archivo la siguiente información:

1. las dimensiones de la imagen: dos números, `n` y `m`, en formato `UInt16` (las necesitaremos para reconstruir la imagen).
2. La matriz de cuantización utilizada: 64 números en formato `Uint8`.
3. La tira de datos que codifica con *Run Length Encoding* la información para reconstruir las matrices: muchos números, en formato `Int8`.

Implementar también la función inversa, que debe leer el archivo y devolver las dimensiones, la matriz de cuantización y la tira de datos que permite reconstruir las matrices.

Los distintos formatos de archivo utilizan marcadores para identificar el inicio o fin de ciertos bloques de información. Por ejemplo: se reserva un código hexadecimal especial para indicar el inicio de la matriz de cuantización y otro para el inicio de los datos que codifican las matrices, etc. Esto permite introducir ciertas variaciones en el formato manteniéndolo compatible con cualquier lector. El lector no esperará que la matriz de cuantización comience en el tercer *casillero* del archivo, sino que la buscará a continuación del marcador estándar. Nosotros hacemos algo más casero: nuestro formato quedará determinado secuencialmente y sólo es posible leerlo conociendo a priori los distintos tipos de datos que se utilizaron.

Nota:
El algoritmo `jpg` tiene un paso más antes del guardado que consiste en codificar los vectores que nosotros generamos mediante un código de Huffman. Este código transcribe la información en formato binario, optimizando la longitud de los "símbolos" que la componen. Nosotros nos estamos salteando ese paso, que permitiría mejorar un poco más la compresión. Una consecuencia de esto es que nuestros archivos no serán verdaderos `.jpg`, sino un nuevo formato que podremos interpretar gracias a las funciones inversas que permiten revertir el proceso.

Todo junto
Ya tenemos todos los elementos. Sólo resta implementar dos funciones que junten todo lo anterior. La primera debe recibir el nombre de archivo de imagen, cargarlo, hacer la compresión y guardarla en un nuevo archivo con el mismo nombre y alguna extensión ficticia. La segunda debe recibir un archivo comprimido y realizar el proceso inverso al de compresión y devolver la imagen (de modo de poder verla dentro de `Julia`).

Pruebas

Probar la compresión con un par de imágenes a elección. Si es posible, busquen imágenes en formato `.bmp` (sin comprimir). Si utilizan imágenes `.jpg` seguramente no obtendrán archivos más chicos. En tal caso es mejor partir de imágenes de alta calidad (que dejen margen para comprimir un poco más). En importante tener en cuenta que distintas matrices de cuantización darán lugar a distintos grados de compresión. Probar al menos dos matrices de cuantización.

Entrega

La entrega debe consistir en un archivo de texto `.jl` que contenga todas las funciones necesarias, y los comentarios que se consideren adecuados para clarificar su funcionamiento. Deben estar claramente identificadas las funciones principales, que realizan la codificación y decodificación de la imagen.

Se sugiere también incluir las imágenes que hayan utilizado para testear el algoritmo y algunas líneas comentadas mostrando cómo correr la compresión y descompresión, luego de incluir el archivo en una sesión de la consola de `Julia`.
