Obligatorio 1. Procesamiento de imágenes

26 de abril de 2024

1. Generalidades

La aprobación de este curso se consigue mediante la correcta implementación de dos pequeños proyectos de programación (que llamaremos obligatorios). Éstos son propuestos en dos momentos del curso, aumentando en complejidad y forman parte de un mismo paquete, alimentándose mutuamente. Los programas desarrollados en la primer entrega serán utilizados en la segunda. Cada obligatorio será entregado a través de una página web habilitada para tales fines, con fecha límite de entrega señalada en la misma página. Estas entregas se complementan con un parcial escrito cuyo objetivo es evaluar aspectos más teóricos relacionados con el propio obligatorio.

Es importante recalcar que **tanto la prueba escrita como el proyecto entregado son individuales.** El sistema de recepción de entregas, además de almacenar los archivos enviados por los estudiantes, realiza un control de copias contra las entregas de otros estudiantes así como de programas similares que se encuentran en la web. Ese programa es capaz de detectar copias "maquilladas", es decir donde se cambiaron nombres de variables u otras formas de ocultar una copia. Este asunto debe ser bien entendido. No tenemos ningún inconveniente en que discutan soluciones, miren en la web, etc. pero **el trabajo entregado debe ser realmente el producto de vuestro trabajo y si el programa de control de copia detecta que hubo copia ello implica una sanción que puede significar la pérdida del curso e incluso sanciones mayores, tal como está especificado en el reglamento de la Facultad [1].**

El sistema intentará compilar y ejecutar la entrega de cada estudiante, a fin de dar un mínimo de información respecto de qué tan bien funciona. La evaluación preliminar mencionada anteriormente **no** determina la nota obtenida en la prueba, siendo ésta definida por una evaluación global por parte de los docentes que incluye los obligatorios, el parcial y la participación en clase.

El valor de cada obligatorio o parcial en el puntaje final del curso es el siguiente:

Prueba	valor en puntos sobre 100	puntaje mínimo
Obligatorio 1	30	8
Obligatorio 2	30	8
Parcial	40	10

Cuadro 1: Puntajes de cada prueba en PIE.

Además, la suma de todas las pruebas debe ser mayor o igual a 60 puntos.

1.1. Formato del archivo a entregar

El archivo entregado debe ser un archivo comprimido en formato **zip** (NO se aceptan archivos en formato rar), de nombre

nombres_separados_por_infraguiones_apellidos_separados_por_infraguiones.zip

y que los fuentes estén en la raíz del zip. El contenido del archivo debe incluir los siguientes elementos (que deben estar en la raíz del mismo y no en un directorio interno):

[1] `https://www.fing.edu.uy/es/gestion/normas-y-reglamentos`

- Todos los archivos fuente creados por el estudiante (**.h** y **.c**)
- Un archivo **Makefile** para compilar el o los programas requeridos en el trabajo.

Por ejemplo, supongamos que el obligatorio consiste en la generación de un ejecutable de nombre `obligatorio1`, su nombre es Juan Pablo Perez Fernandez, y usted implementó dicho ejecutable en un archivo **imagen.c** y su correspondiente archivo de encabezado **imagen.h**. Entonces debe subir un archivo de nombre `Juan_Pablo_Perez_Fernandez.zip` con el siguiente contenido:

```
imagen.c
imagen.h
obligatorio1.c
Makefile
```

El **Makefile** en este caso debe ser así:

```
all: libimagen.a obligatorio1

DEPS = imagen.h  iio.h
LDLIBS = -ljpeg -ltiff -lpng

obligatorio1: obligatorio1.o libimagen.a
        cc $(CFLAGS) -o $@ obligatorio1.o -L./ -limagen -liio $(LDLIBS) -lm

%.o: %.c %.h $(LDLIBS)
        cc $(CFLAGS) -c $<

libimagen.a: imagen.o
        ar rcs $@ $<

clean:
        rm -rf *.o
        rm -rf libimagen.a
        rm -rf obligatorio1
```

En el caso de trabajar en Mac recientes (M1 o M2), el **Makefile** en este debe ser:

```
all: libimagen.a obligatorio1

DEPS = imagen.h iio.h

# Compiler flags
CFLAGS = -I<path-libs-estudiantes>/jpeg/include -I<path-libs-estudiantes>/libtiff/
    include -I<path-libs-estudiantes>/libpng/include
LDLIBS = -L<path-libs-estudiantes>/jpeg/lib -L<path-libs-estudiantes>/libtiff/lib
    -L/<path-libs-estudiantes>/libpng/lib -ljpeg -ltiff -lpng

obligatorio1: obligatorio1.o libimagen.a
        clang $(CFLAGS) -o $@ obligatorio1.o -L./ -limagen $(LDLIBS) -lm -liio

%.o: %.c $(DEPS)
        clang $(CFLAGS) -c $<

libimagen.a: imagen.o
        ar rcs $@ imagen.o

clean:
        rm -rf *.o
        rm -rf libimagen.a
        rm -rf obligatorio1
```

*Nota: se sugiere transcribir manualmente el Makefile al archivo correspondiente, **no** utilizar Copiar y Pegar para evitar que se introduzcan espacios u otros caracteres adicionales. Notar también que el*

Makefile es sensible a indentación, por lo que es necesario que se respete la tabulación de las líneas 5,7 y 10.

Nota 2: si tienen una Mac reciente deben sustituir $<path-libs-estudiantes>$ por el camino al lugar donde tienen las bibliotecas en su máquina.

En el archivo **Makefile** anterior se puede generar la biblioteca **libimagen.a** y también compilar un pequeño programa de prueba que hemos llamado **obligatorio1.c** que simplemente llama a las funciones implementadas con ciertos valores como parámetros de entrada e imprime el resultado. Esto les debe servir a ustedes para ver si dan los resultados correctos las funciones que están en la biblioteca. Para ello se debe invocar el **Makefile** de la siguiente manera:

```
make libimagen.a
make obligatorio1
```

La primera línea genera la biblioteca **libimagen.a** que se utiliza en el programa **obligatorio1.c**. La segunda línea genera el ejecutable **obligatorio1**.

En la ultimas líneas se incluye clean, que se encarga de borrar todos archivos generados durante la compilación. Lo pueden ejecutar de esta forma:

```
make clean
```

También pueden invocar todo (all):

```
make
```

Nota: Pueden crear un zip desde la máquina Linux/Mac con el comando `zip`; la sintaxis es, desde la carpeta de trabajo:

```
$zip nombre_archivo.zip imagen.c imagen.h obligatorio1.c Makefile
```

en el ejemplo anterior, sería `zip Juan_Pablo.Perez_Fernandez imagen.c imagen.h obligatorio2.c Makefile`

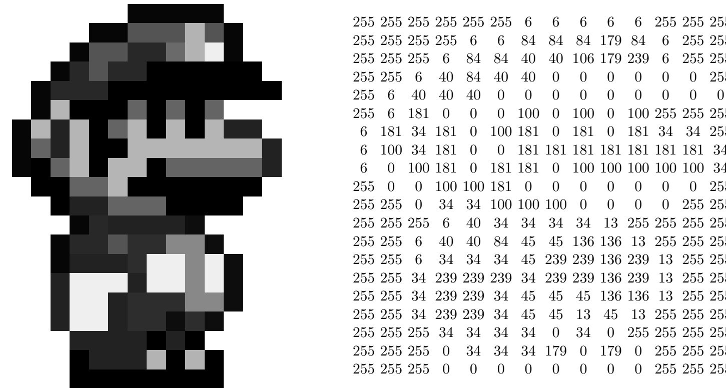
2. Introducción al problema

A lo largo de este curso realizaremos 2 obligatorios que están relacionados entre si. El objetivo final es realizar operaciones en imágenes planas: adicionar y filtrar ruido, transformar la apariencia de la imagen y extraer los bordes que están presentes en la misma. Para realizar este tipo de operaciones deberemos ser capaces de abrir un archivo de imagen, leer la misma, modificar sus características y guardar el resultado en otro archivo. Utilizaremos una pequeña biblioteca que permite leer y salvar imágenes en formatos png, pgm y jpg. En términos generales los obligatorios se ocuparán de los siguientes aspectos:

- Obligatorio 1. Construiremos una pequeña biblioteca de mejora de imágenes *mejora.bib* que permita agregar ruido de dos tipos (gaussiano e impulsivo) a las mismas, calcular su histograma, ecualizar la imagen y aplicar dos filtros para disminuir el ruido de la imagen: filtro promediador y filtro de mediana. El resultado de las operaciones aplicadas sobre una imagen leída de un archivo se guardará en un archivo para poder observarlo con un programa de visualización de imágenes estándar.

- Obligatorio 2. Construiremos una pequeña biblioteca de procesamiento de imágenes *imagenes.bib* que permita detectar los bordes presentes en una imagen y producir una representación vectorial de los mismos.

Figura 1: Imagen en escala de grises.



255	255	255	255	255	255	6	6	6	6	6	255	255	255
255	255	255	255	6	6	84	84	84	179	84	6	255	255
255	255	255	6	84	84	40	40	106	179	239	6	255	255
255	255	6	40	84	40	40	0	0	0	0	0	0	255
255	6	40	40	40	0	0	0	0	0	0	0	0	0
255	6	181	0	0	0	100	0	100	0	100	255	255	255
6	181	34	181	0	100	181	0	181	0	181	34	34	255
6	100	34	181	0	0	181	181	181	181	181	181	181	34
6	0	100	181	0	181	181	0	100	100	100	100	100	34
255	0	0	100	100	181	0	0	0	0	0	0	0	255
255	255	0	34	34	100	100	100	0	0	0	0	255	255
255	255	255	6	40	34	34	34	34	13	255	255	255	255
255	255	6	40	40	84	45	45	136	136	13	255	255	255
255	255	6	34	34	34	45	239	239	136	239	13	255	255
255	255	34	239	239	239	34	239	239	136	239	13	255	255
255	255	34	239	239	34	45	45	45	136	136	13	255	255
255	255	34	239	239	34	45	45	13	45	13	255	255	255
255	255	255	34	34	34	34	0	34	0	255	255	255	255
255	255	255	0	34	34	34	179	0	179	0	255	255	255
255	255	255	0	0	0	0	0	0	0	0	255	255	255

Figura 2: Izq.:Imagen en escala de grises de 20×20 píxeles. Der.: píxeles de la imagen

El problema que se plantea en este obligatorio, el procesamiento de imágenes, es una de las subáreas más populares e importantes del procesamiento de señales. Si bien desde el punto de vista teórico y formal, las herramientas para trabajar con este tipo de problemas se ven recién en los últimos dos años de la carrera de Ingeniería Eléctrica, es posible trabajar con, y comprender informalmente, muchos algoritmos importantes de procesamiento de imágenes del estilo de los que se ven en aplicaciones populares de retoque de fotografías. En este caso vamos a trabajar con aspectos muy elementales de ese mundo pero esperamos que ello ya les permita tener una idea de cómo modificar imágenes y quizás los anime a experimentar más.

2.1. Representación digital de imágenes

Lo primero que tenemos que definir es cómo se representa una imagen digitalmente, de modo de poder almacenarla en memoria y trabajar con ella. Consideremos en primer lugar las imágenes en escala de grises, como la que se ve en la figura 1. La representación usual de este tipo de imágenes es la de una matriz de tamaño $w \times h$, donde w es el ancho y h el alto, en píxeles, de la imagen. Si denominamos como I a tal matriz, y medimos la posición de los píxeles de la imagen a partir de su esquina superior izquierda $(0, 0)$, el valor de la posición $I(i, j)$ de la matriz correspondiente a esa imagen nos indicará la intensidad del píxel ubicado en la coordenada (i, j); mientras más alto el valor de $I(i, j)$, más brillante será el píxel de la coordenada (i, j) en la imagen. Para imágenes en niveles de gris nos limitaremos a 8 bits por píxel, que se traduce en que $I(i, j)$ puede tomar valores entre 0 y 255, siendo 0 el color *negro* y 255 el *blanco*. La figura 2 da un ejemplo de lo anterior.

255, 0, 0	255, 255, 0	0, 255, 0	0, 255, 255
255, 0, 0	255, 255, 0	0, 255, 0	0, 255, 255
0, 0, 255	0, 0, 0	255, 255, 255	255, 0, 255
0, 0, 255	0, 0, 0	255, 255, 255	255, 0, 255

Figura 3: Imagen RGB de tamaño 4×4. Izq.: imagen, Der.: píxeles de la imagen. Cada terna representa un píxel con sus valores de rojo (R), verde (G) y azul (B). Noten que 255 quiere decir que ese canal tiene el valor máximo posible pues cada canal está codificado en 8 bits, o sea un *unsigned char*.

2.2. Representación de imágenes a color

Existen muchas formas de representar una imagen a color bidimensional. En este caso el valor de cada píxel, en lugar de representar una intensidad de gris, representa un color. En general, esto se logra representando a cada color como un vector de *canales* que definen al color. Ejemplos de estos son la descomposición RGB (Red, Green, Blue) o la HSV (Hue, Saturation, Value). La primera es más sencilla, mientras que la segunda está más alineada con la forma en que el ojo humano percibe el color.

En nuestro caso utilizaremos la representación RGB, que es una representación *aditiva* del color, en el sentido de que el color resulta de sumar la intensidad lumínica de los tres canales; ésta es la representación de colores utilizada en todos los monitores y televisores. En esta representación, cada píxel es representado por una terna de valores enteros (r, g, b), cada uno en el rango $[0, M]$ que, de la misma manera que en las imágenes de tono de gris, indican la intensidad del canal correspondiente. El valor $(0, 0, 0)$ se corresponde con el negro, y el (M, M, M) con el blanco. En nuestro caso, trabajaremos con 8 bits por canal, por lo que M sera a lo sumo 255 y cada canal requiere solo un byte para representar su valor. La figura 3 muestra un ejemplo muy sencillo de imagen RGB con $M = 255$.

Podemos representar una imagen color como un arreglo de enteros de tamaño $w \times h$, donde w es la cantidad de columnas (ancho) y h es la cantidad de filas (alto) de la imagen. Un entero de 32 bits contiene 4 `bytes` y por tanto la posibilidad de representar la terna correspondiente al color y sobra un `byte` que a veces se utiliza con otros fines.

Acorde con lo arriba descrito, diremos que una imagen color I es una matriz de tamaño $m \times n$, donde cada elemento es una terna (r, g, b), de modo que

$$I(i, j) = (r_{ij}, g_{ij}, b_{ij}), 0 \leq i < w, 0 \leq j < h.$$

Asimismo, definimos los canales de una imagen color RGB como las tres imágenes de escalas de grises I_R, I_G e I_B cuyos valores son las intensidades de cada uno de los canales por separado como se muestra en la figura 4. Es decir

$$I_R(i, j) = r_{ij}, \quad I_G(i, j) = g_{ij}, \quad I_B(i, j) = b_{ij}, \quad 0 \leq i < w, 0 \leq j < h$$

Es posible también extraer los canales de una imagen así representada y construir 3 arreglos de $w \times h$ `bytes`.

2.3. Representación en memoria

Hemos visto que una imagen se representa como una matriz de píxeles. Resta entonces definir cómo representar esa matriz en memoria. Hay básicamente dos maneras: mediante arreglos multidimensionales, o mediante arreglos unidimensionales de algún tipo numérico que represente el valor de un píxel (por ejemplo `int` o `unsigned char`). La primera es posiblemente la más natural, pero es posible también trabajar con la segunda forma, y es la que utilizaremos en este obligatorio.

En el caso unidimensional, es necesario definir cómo se traduce cada coordenada bidimensional (i, j) de la imagen a un índice unidimensional dentro del arreglo y viceversa. Usamos i para indexar las filas y j las columnas. El método más usado, llamado "barrido por filas", almacena la fila superior

Figura 4: Imagen a color I y sus tres componentes I_R, I_G y I_B. Notar que el blanco se forma con el máximo de los tres canales, por lo que todos aparecen también con la máxima intensidad en el fondo. Sin embargo, el canal azul (más a la derecha) aparece más oscuro, debido a la poca presencia del azul en la imagen mostrada.

en los primeros n elementos del arreglo (recuérdese que en C los índices comienzan en 0, por lo cual el píxel $(0,0)$ irá en la posición 0 del arreglo y el píxel $(0,1)$, primero de la segunda fila, irá en la posición $w-1$). La segunda fila ocupa los siguientes w elementos (de w a $2w-1$), y así por delante hasta completar los $w\times h$ elementos del arreglo.

En este obligatorio utilizaremos arreglos unidimensionales de píxeles para almacenar las imágenes. Una imagen es entonces un arreglo de $m\times n$ píxeles. Dependiendo de si la imagen es en grises o en colores ese arreglo deberá ser de uno o varios `unsigned char`.

Habiendo definido cómo se representan las imágenes en memoria, vayamos a lo que es el objetivo de este proyecto, es decir, poder manipular de manera muy elemental las imágenes. En general tomaremos una imagen de entrada, operaremos sobre ella y produciremos una imagen de salida, de manera que esta última es una versión alterada de la primera. Vamos a leer y escribir archivos en formato *.pgm*, *.png* y *.jpg* y convertir imágenes a color a imágenes en escala grises. Luego las imágenes de grises van a sufrir ciertas transformaciones y el resultado será guardado en formato *.pgm*.

Utilizaremos cualquier utilitario estándar para visualizar las imágenes, tanto las que tomemos como entrada como aquellas que vamos a producir con nuestro programa. Sugerimos usar IJ-ImageJ (url: https://imagej.net/ij/).

Lo primero que tenemos que hacer es ser capaces de leer y escribir en ciertos formatos de archivo de imágenes, para ello utilizaremos una pequeña biblioteca `libiio.a`[2] y su archivo de encabezado asociado `iio.h` que les proporcionamos. En particular utilizaremos las siguientes funciones:

- `uint8_t *iio_read_image_uint8_vec(const char *fname, int *w, int *h, int *pd)` que lee el archivo *fname* y devuelve un arreglo unidimensional de `uint8_t` conteniendo los píxeles de la imagen, de dimensión $w \times h$ si es una imagen de gris y 3 veces esa dimensión si es color, conteniendo $w \times h$ valores de los píxeles como un `uint8_t` cuando es en nivel de gris o 3 `uint8_t` cada uno con la componente rojo, verde y azul de ese pixel (RGB) cuando es color. En w devuelve el número de columnas, en h el número de filas y en *pd* la dimensión del píxel: 1 si es una imagen de grises y 3 si es color. Noten que se trata de flotantes pero el rango de los valores utilizados solo van de 0 a 255. La memoria para guardar los datos es reservada dentro de la función, es responsabilidad del usuario liberarla una vez que no se use más. El nombre del archivo a leer indica en qué formato está (jpg, png o pgm).

  Noten que la función genera un arreglo de `uint8_t`, que es un tipo definido en la biblioteca `libiio.a` y que refiere a los `unsigned char`. Noten también que dependiendo de si la imagen es monocromática o color, cada píxel esta formado por 1 o 3 caracteres, respectivamente.

- `void iio_write_image_uint8_vec(const char *fname, uint8_t* data, int w, int h, int pd)` que guarda en el archivo llamado *fname* los datos apuntados por *data*, organizados en h filas y w columnas. En este caso siempre guardaremos imágenes monocromas (niveles de gris), por lo

[2] https://github.com/mnhrdt/iio

que el tamaño del píxel es $pd = 1$. El nombre del archivo indica en qué formato se guarda (jpg, png o gpm).

La forma de utilizar la biblioteca es incluyéndola en el linkeditado, tal como se indica en el `Makefile` explicitado antes. Para ello recuerden que la opción correspondiente de `cc` define por un lado dónde buscar las bibliotecas locales (`-L./`) y por otro lado a cada una le elimina las letras **ib** de **lib** y elimina también el **.a** al final, de modo que para invocar la biblioteca **libiio.a** se debe poner **-liio**, como aparece en el Makefile, y el archivo debe llamarse en ese caso **libiio.a**. además, en aquellos archivos fuente en que se utilice la biblioteca se debe incluir el archivo `iio.h`.

Cuando leamos una imagen, verificaremos si está en color o es monocromática. En el primer caso llenaremos los campos RGB `pixelesR`, `pixelesG` y `pixelesB` en la estructura `Imagen` y procederemos a transformar la imagen a grises y llenar los valores correspondientes en el campo `pixeles` de dicha estructura. En el segundo caso sólo llenaremos directamente los valores del campo `pixeles` en dicha estructura. Todas las funciones las aplicaremos sobre las imágenes de grises (monocromáticas), es decir sobre los valores guardados en `pixeles`.

Existe una enorme cantidad de filtros y efectos de procesamiento de imágenes; algunos tienen como objetivo mejorar las imágenes (como por ejemplo eliminar ruido o aumentar el contraste y la nitidez), distorsionar o aplicar efectos artísticos, resaltar partes (por ejemplo bordes y esquinas) o transformar (rotar, escalar). El objetivo de este obligatorio es el de implementar las siguientes funciones básicas:

1. Conversión de imágenes de color a grises.
2. Agregar ruido Gaussiano a la imagen.
3. Agregar ruido de sal y pimienta a la imagen.
4. Filtro promediador.
5. Filtro de mediana.
6. Calcular el histograma de la imagen.
7. Calcular el histograma acumulado de la imagen.
8. Ecualizar la imagen.

2.4. De color a blanco y negro

En ocasiones es necesario obtener una versión en grises de una imagen a color. Idealmente, la versión en blanco y negro de una imagen I, a la que denominaremos I_M, corresponde a la intensidad de luz percibida por el ojo humano para cada uno de sus píxeles. Existen varias fórmulas para estimar esta intensidad a partir de los colores de un píxel. Nosotros utilizaremos la siguiente. Si (r, g, b) son los tres canales de un píxel color, su versión blanco y negro de acuerdo a esta fórmula será:

$$w = 0{,}2 \times r + 0{,}7 \times g + 0{,}1 \times b$$

2.5. Agregar ruido Gaussiano a la imagen

Se trata de agregar a cada píxel de la imagen un valor producido por un proceso Gaussiano de parámetros media $\mu = 0$ y desviación estándar $\sigma = \mathit{potencia} \times 255$, donde *potencia* es un parámetro entre 0 y 1, de forma tal que $0 \leq \sigma \geq 255$. se debe truncar a 0 si el valor calculado es menor a 0 y a 255 si es mayor a 255, para garantizar que el valor del píxel con ruido siga siendo un valor valido para un píxel de 8 bits.

2.6. Agregar ruido de sal y pimienta a la imagen

Se trata de escoger aleatoriamente una coordenada de la imagen (En esta ocasión vamos a escoger el píxel $I(i,j)$ definiendo primero aleatoriamente un valor de la coordenada i y luego, de nuevo aleatoriamente, un valor de la coordenada j) y, finalmente, nuevamente aleatoriamente darles el valor 0 o 255. La potencia del ruido es un parámetro entre 0 y 1 y define el porcentaje de píxeles de la imagen que son cambiados.

2.7. Filtro promediador

Definimos el filtrado de una imagen I como la convolución de la misma por un kernel K:

$$I_f = K * I$$

En la práctica, se recorre I de izquierda a derecha y de arriba a abajo, visitando todos los píxeles de la imagen de entrada y calculando en cada caso el valor del píxel homólogo de la imagen de salida:

$$I_f(x,y) = 1/N \times \sum_{u=-orden}^{u=orden} \sum_{v=-orden}^{v=orden} (I(x+u, y+v) \times K(u,v))$$

donde el kernel K es un arreglo cuadrado de lado $2 \times orden + 1$ y esta centrado, es decir que en la ecuación anterior el elemento central del kernel está posicionado sobre el píxel en que se está calculando el valor. Los valores de K definen los parámetros del filtro. El factor de normalización es:

$$N = \sum_{u=-orden}^{u=orden} K(u,v)$$

En el caso del filtro promediador el kernel de orden 1 es un cuadrado de dimensión 3×3:

$$K_{prom} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

De manera que el filtro efectivamente promedia los valores de los píxeles vecinos y da ese valor al píxel homólogo en la imagen de salida.

Estrategia de borde Para calcular el valor del píxel filtrado cuando está ubicado muy cerca del borde de la imagen nos encontramos con un problema, pues el kernel tiene valores que caen fuera de la imagen. En esos casos debemos utilizar una estrategia de borde. En nuestro caso optaremos por crear una imagen de salida I_f idéntica a la original pero *orden* filas y columnas al principio y al final deberán tener un valor cero, dado que en esos lugares no podemos calcular el resultado del filtro porque parte de los valores están indeterminados, pues el núcleo del filtro cae parcialmente fuera de la imagen.

2.8. Filtro de mediana

El filtro de mediana usa también una ventana cuadrada de dimensión $2 \times orden + 1$, pero en vez de hacer una convolución de la imagen con dicho kernel, lo que hace es ordenar de menor a mayor los píxeles de la imagen I en ese vecinaje y asignar al píxel homólogo en la imagen filtrada el valor del medio de dicho ordenamiento. Recorremos la imagen de la misma manera y usamos la misma estrategia de borde que en el filtrado promediador.

1	2	3	4	5

Cuadro 2: LUT identidad.

1	1.4142	1.7320	2	2.2360

Cuadro 3: LUT raíz cuadrada.

2.9. Histograma

El histograma de una imagen $H(I)$ es una tabla con el conteo del número de píxeles de la imagen que tienen un dado valor. Para una imagen de grises los píxeles toman 256 valores posibles, por tanto el histograma es una tabla de 256 casillas. La casilla 0 contiene el número de píxeles de la imagen cuyo valor es 0, la casilla 1 contiene el número de píxeles de la imagen cuyo valor es 1, y así sucesivamente. Esta función representa la distribución de los valores de los píxeles en el imagen.

2.10. Histograma Acumulado

El histograma acumulado de una imagen $H(I)_a$ es una tabla con el conteo del número de píxeles de la imagen que tienen un valor menor o igual que el correspondiente al índice de la tabla. Para una imagen de grises los píxeles toman 256 valores posibles, por tanto el histograma es una tabla de 256 casillas. La casilla 0 contiene el número de píxeles de la imagen cuyo valor es 0, la casilla 1 contiene el número de píxeles de la imagen cuyo valor es 0 o 1, y así sucesivamente. Por construcción, se trata de una función monótona creciente.

2.11. LUT

Una LUT, o Look Up Table, es una tabla de una dimensión que se usa para aplicar una transformación a una imagen que solo depende del valor del píxel tratado. Para codificar una función $f(x)$, en los valores de la tabla se pone el valor discreto de $f(x)$ y el índice de la tabla representa el valor discreto de x. Por ejemplo, el cuadro 2 codifica la función identidad para 5 valores. En ese caso $LUT(4) = 4$. Si codificáramos la función raíz cuadrada, la LUT sería como se representa en el cuadro 3 y $LUT(2) = 1{,}4142$

2.12. Ecualización

Ecualizar una imagen de gris es redistribuir sus niveles de gris de tal manera que el histograma $H(I_e)$ de la imagen ecualizada I_e se aproxime a una distribución uniforme. Para lograrlo utilizamos una transformación que modifica el valor de cada píxel asignándole el valor correspondiente en el histograma acumulado de la imagen de entrada:

$$I_e(x, y) = H_e(I(x, y))$$

Para esto utilizamos el histograma acumulado como una LUT, lo que implica que sus valores deben ser normalizados entre 0 y 255, que es el rango admisible de valores de gris en una imagen monocromática.

3. Descripción de la tarea

La implementación de esta tarea consiste en crear una biblioteca de tratamiento de imágenes y que sea capaz de hacer algunas operaciones sencillas sobre imágenes.

La biblioteca consistirá en dos archivos: un encabezado `imagen.h` y una implementación `imagen.c`, los cuales serán utilizados dentro del ejecutable a entregar. En `imagen.h` se deberán declarar una serie

de constantes, tipos de datos, y funciones, de acuerdo a requerimientos a ser especificados más adelante en este documento, y luego implementar las funciones en `imagen.c`.

La biblioteca será utilizada como parte de la implementación del ejecutable llamado `obligatorio1`, pero también deberá poder ser invocada por un ejecutable no especificado. En particular en el `obligatorio2` se hará uso de esta biblioteca.

4. Especificación de requerimientos

4.1. Biblioteca de lectura/escritura de imágenes

La biblioteca debe declarar los siguientes tipos en `imagen.h`:

- Tipo `TipoImagen`, en base a una enumeración de nombre `tipo_imagen` con los valores posibles `GRISES=0` y `COLOR=1`
- Tipo `Imagen`, en base a una estructura de nombre `imagen` con los siguientes campos:
    - `tipo`: de tipo `TipoImagen`
    - `columnas`: de tipo `int`
    - `filas`: de tipo `int`
    - `pixeles`: de tipo puntero a `uint8_t`, que apuntará a una zona de memoria suficiente para almacenar los píxeles de la imagen en niveles de gris.
    - `pixelesR`: de tipo puntero a `uint8_t`, que apuntará a una zona de memoria con la componente rojo de la imagen, cuando la imagen es en colores.
    - `pixelesG`: de tipo puntero a `uint8_t`, que apuntará a una zona de memoria con la componente verde de la imagen, cuando la imagen es en colores.
    - `pixelesB`: de tipo puntero a `uint8_t`, que apuntará a una zona de memoria con la componente azul de la imagen, cuando la imagen es en colores.

  Toda imagen tendrá esta estructura. Si es de tipo GRISES, se utilizará sólo el campo `pixeles` (a llenar al leer el archivo) y si es de tipo COLOR, se utilizarán los campos `pixelesR`, `pixelesG` y `pixelesB` para guardar la imagen color que se lee y el campo `pixeles` para guardar su conversión a grises.
- Tipo `CodigoError`, basado en una enumeración `codigo_error` con los siguientes valores posibles:
    - `OK=0`: se devuelve si todo salió bien
    - `ARCHIVO_INEXISTENTE=1`: si el archivo no eiste.
    - `ERROR_LECTURA=2`: si ocurre un error al leer datos del archivo (por ejemplo con `fread`, o `fscanf`)
    - `ERROR_ESCRITURA=3`: si ocurre un error al escribir datos del archivo (por ejemplo con `fwrite`, o `fprintf`)
    - `ERROR_MEMORIA=4`: se devuelve ante cualquier error que ocurra durante la reserva de memoria.
    - `ERROR_NO_GRISES=5`: el campo `pixeles` no está lleno, porque no se hizo conversión a grises de una imagen color.
    - `ERROR=6`: otro error no listado anteriormente.

Asimismo, debe declarar las siguientes funciones en `imagen.h`, y definirlas en `imagen.c`:

- `CodigoError inicializarImagen(Imagen* pnew, int cols, int fils, TipoImagen tipo)` inicializa los datos de la imagen apuntada por `pnew` con los parámetros especificados. Los píxeles deben inicializarse a 0. Devuelve un valor de tipo `CodigoError` según el caso. Noten que deben haber creado previamente la imagen `pnew` y se debe pasar su dirección.

- `CodigoError destruirImagen(Imagen* pimg)` libera la memoria asociada a los píxeles de la imagen apuntada por `pimg` si la misma no es NULL. Si es de tipo GRISES debe hacer esto sobre el campo `pixeles`, si es de tipo COLOR, debe hacer esto sobre los campos `pixelesR`, `pixelesG`, `pixelesB` y `pixeles`. Pone los siguientes atributos en `0`: `columnas`, `filas` y `tipo`. Devuelve un valor de tipo `CodigoError` según el caso.

- `CodigoError duplicarImagen(const Imagen* pin, Imagen* pout);` copia los atributos de una imagen de entrada apuntada por `pin` (inicializada previamente) en la imagen apuntada por `pout` (no inicializada aún), reservando espacio para los nuevos píxeles. Los valores de píxeles de la nueva imagen serán copiados de la de entrada. Devuelve un valor de tipo `CodigoError` según el caso.

- `CodigoError leerImagen(const char* ruta, Imagen* pimg);` usa la función correspondiente de `libiio.a` para leer el archivo ubicado en `ruta` en el campo `pixeles` de la imagen apuntada por `pimg`, si es una imagen monocromática o en los campos `pixelesR`, `pixelesG` y `pixelesB` de la imagen apuntada por `pimg` si es en colores. Devuelve un valor de tipo `CodigoError` según el caso.

- `CodigoError escribirImagen(const Imagen* pimg, char* ruta);` guarda el contenido de la imagen `pimg` en el archivo especificado por `ruta`. Deben utilizar la función correspondiente de `libiio.a`. Al igual que en el caso anterior, debe devolver un valor de tipo `CodigoError` según el caso.

- `CodigoError convertirImagenColorAGris(const Imagen* pin, Imagen* pout)` copia los atributos de una imagen de tipo COLOR de entrada apuntada por `pin` en la imagen de tipo GRISES apuntada por `pout`. La reserva de memoria para los píxeles de `pout` se debe realizar mediante invocación a la función `duplicarImagen`. Se calcula el valor de gris de cada píxel y se llena el campo `pixeles` correspondiente. Se devuelve un valor de tipo `CodigoError` según el caso.

- `long int * Histo(const Imagen* pin)`. Esta función calcula el histograma de la imagen apuntada por `pin` y la devuelve como un arreglo de `long int`.

- `long int * HistoAcum(const long int* histo, int size)`. Esta función calcula el histograma acumulado de la imagen apuntada por `pin` y la devuelve como un arreglo de `long int`. Si el parámetro $size = 0$, en el arreglo de salida los valores varían entre 0 y el número total de píxeles de la imagen. Si $size \neq 0$, se entiende que el valor allí presente es el número total de píxeles de la imagen y se utiliza para cuantificar la LUT de tal manera que los valores de la tabla varíen entre 0 y 255.

- `CodigoError LUT(const Imagen* pin, Imagen* pout, long int * LUT)`. Esta función aplica la LUT apuntada por el parámetro `LUT` a la imagen apuntada por `pin` y el resultado lo pone en la imagen apuntada por `pout`, que debe ser creada dentro de la función mediante una duplicación de `pin`. Se devuelve un valor de tipo `CodigoError` según el caso. Recuerden que esta función se aplica sobre imágenes monocromas.

- `double SampleGaussian(double mu, double sigma, int seed)`. Esta función es un generador de muestras con distribución Gaussiana que debe utilizar el método de Box-Muller Transform [3]. Cada vez que es invocada devuelve un doble que es una muestra de una distribución Gaussiana

[3] Pueden usar el código que se explica en https://phoxis.org/2013/05/04/generating-random-numbers-from-normal-distribution-in-c/

de parámetros μ y σ. Antes de invocar esta función se deberá invocar la función `srand(seed)` para poder usar el generador de números aleatorios `rand()`.

- `CodigoError AgregarRudioSalYPimienta(const Imagen* pin, Imagen* pout, float pot, int seed)` que produce una imagen apuntada por `pout` que es copia de la imagen apuntada por `pin` a la que se le ha agregado ruido de sal y pimienta de potencia `pot`, entre 0 y 1. La reserva de memoria para los píxeles de `pout` se debe realizar usando la función `duplicarImagen`. Se incluye como parámetro la semilla `seed` para el generador de números aleatorios. Se devuelve un valor de tipo `CodigoError` según el caso.
- `CodigoError AgregarRudioGaussiano(const Imagen* pin, Imagen* pout, float pot, int seed)` que produce una imagen apuntada por `pout` que es copia de la imagen apuntada por `pin` a la que se le ha agregado ruido Gaussiano de potencia `pot`, un valor entre 0 y 1 que determina la desviación standard σ de la distribución gaussiana como $\sigma = pot * 256$. Por ejemplo, si $pot = 0{,}1$, entonces $\sigma = 25$. Para generar cada muestra se debe invocar la función `SampleGaussian(0,σ)`. Recuerden que la imagen de salida debe variar entre 0 y 255, por tanto se deberán truncar los valores menores a 0 y mayores a 255. La reserva de memoria para los píxeles de `pout` se debe realizar usando la función `duplicarImagen`. Se incluye como parámetro la semilla `seed` para el generador de números aleatorios. Se devuelve un valor de tipo `CodigoError` según el caso.
- `CodigoError FiltroPromedio(const Imagen* pin, Imagen* pout, int orden)`. Esta función toma como entrada una imagen apuntada por `pin` y le aplica un filtro promediador de orden `orden`. La reserva de memoria para los píxeles de `pout` se debe realizar usando la función `duplicarImagen`. El resultado lo coloca en la imagen de salida apuntada por `pout`. Se devuelve un valor de tipo `CodigoError` según el caso.
- `CodigoError FiltroMediana(const Imagen* pin, Imagen* pout, int orden)`. Esta función toma como entrada una imagen apuntada por `pin` y le aplica un filtro de mediana de orden `orden`. La reserva de memoria para los píxeles de `pout` se debe realizar usando la función `duplicarImagen`. El resultado lo coloca en la imagen de salida apuntada por `pout`. Para hacerlo utiliza la función `qsort`. Se devuelve un valor de tipo `CodigoError` según el caso.

4.2. Programa del obligatorio 1

Deben escribir un programa, **obligatorio1.c**, que contenga solo el `main()` y que invoque las distintas funciones de la biblioteca. Prueben todas las funciones de la biblioteca verificando los resultados mediante la visualización de las imágenes que creen.

Interfaz de linea de comandos La sintaxis será por línea de comando el nombre del ejecutable segido de tres parámetros que se describen a continuación, y se invocará de la misma manera:

```
./obligatorio1 entrada opcion [valor]
```

Los parámetros son:

- el primer parámetro es el nombre del archivo a leer, incluyendo el camino absoluto del directorio donde está el archivo (si fuera necesario). Archivo de entrada.
- el segundo parámetro es un carácter que debe indicar qué función probar según la siguiente nomenclatura:
  - G convertir a gris la imagen de entrada y poner el resultado en la de salida una salida llamada `ImagenGrises.png`.
  - E ecualizar la imagen de entrada (previa conversión a gris si es en color) y enviarla a la salida llamada `ImagenEcualizada.png`.

- RG aplicar a la imagen de entrada (previa conversión a gris si es en color) un ruido Gaussiano cuya potencia está definida en el 3er parámetro. Guardar el resultado en un archivo llamado `ImagenConRuidoGaussiano.png`.
- RSP aplicar a la imagen de entrada (previa conversión a gris si es en color) un ruido de sal y pimienta cuya potencia está definida en el 3er parámetro. Guardar el resultado en un archivo llamado `ImagenConRuidoSalYPimienta.png`.
- H calcular el histograma de la imagen de entrada (previa conversión a gris si es en color) y guardarlo como un archivo de texto llamado `histo.txt`.
- A calcular el histograma acumulado de la imagen de entrada (previa conversión a gris si es en color) y guardarlo como un archivo de texto llamado `histoAcum.txt`. Utilizar parámetros $lut = 1$ y $size$ el tamaño de la imagen.
- P aplicar el filtro promediador a la imagen de entrada (previa conversión a gris si es en color). El orden del filtro está definido por el 3er parámetro. Guardar el resultado en un archivo llamado `ImagenFiltradaPromediador.png`.
- M aplicar el filtro de mediana a la imagen de entrada (previa conversión a gris si es en color). El orden del filtro está definido por el 3er parámetro. Guardar el resultado en un archivo llamado `ImagenFiltradaMediana.png`.

- el tercer parámetro puede existir o no, dependiendo del valor del 2do parámetro. Su significado puede ser:
  - potencia valor flotante entre 0 y 1.
  - orden del filtro, entero entre 1 y 3.

Si el número de parámetros de la línea de comando es incorrecta debe imprimirse un mensaje explicando cómo realizar una invocación correcta y el significado de cada parámetro.

El archivo de entrada puede ser en colores. El programa debe actuar de manera acorde según si la entrada es a colores o blanco y negro. En el caso de que el archivo sea en colores, se debe convertir a grises y todas las funciones aplicarse sobre la versión en grises de la imagen. Si el archivo de entrada es en grises todas las funciones se aplican sobre la imagen tal cual (en grises).

Consideraciones y sugerencias

- Recuerden que si utilizan funciones matemáticas de **math.h** deben luego linkear con la biblioteca de matemática con la opción `-lm` al final de la linea que genera el ejecutable, para que dichas funciones estén definidas.
- Es **fundamental** liberar correctamente toda la memoria que haya sido reservada por el programa. **Parte de la evaluación del funcionamiento correcto de la tarea incluirá verificar que esto se esté haciendo correctamente**.
- Como siempre, implementar y probar de a poco.
- En caso de bugs, el GDB es su mejor amigo.