Tarea 1 Gauss & Laplace Pyramid

Estudiante: Luis Escares
Profesor: Javier Ruiz del Solar
Auxiliar: Patricio Loncomilla
21 de Agosto de 2019

I. Introducción

En este informe se describe la implementación del filtrado Gaussiano de imágenes por convolución, la construcción de las pirámides de Gauss y de Laplace usando ese filtrado; y la reconstrucción de la imagen que origino dichas pirámides mediante un procesos de interpolación (promediación). En este informe es describen los efectos de dichas implementaciones sobre imágenes clásicas del procesamiento de imágenes, que poseen características especiales como muchos bordes o gran cantidad de contrastes de iluminación.

Para realizar esta experiencia se utiliza OpenCV 2.3, que es una biblioteca para la visión computacional desarrollada por Intel. Tiene interfaces en C++, Java, Python y Matlab. Además, esta soportada en Linux, Windows y Mac OS. En estos experimentos se usa para manejar las imágenes haciendo uso de su objeto central que es "Mat". Su ventaja es que permite trabajar fácilmente con imágenes representándolas como matrices. Además, debido a que los códigos de esta experiencia también están escritos en C++, el tiempo de procesamiento es extremadamente bajo.

Las secciones del informe corresponden a un marco teórico donde se expone el objetivo general de las operaciones de convolución y se aterriza a la aplicación del caso discreto en imágenes bidimensionales. Luego, se describe que son las pirámides de Gauss y de Laplace , para que es lo que sirven y se explica a grandes rasgos como se calculan. Por último se describe el proceso de reconstrucción de la imagen que genera estas pirámides a partir de la pirámide de Laplace y el uso de interpolación lineal simple, es decir, usando promedios entre píxeles.

La sección de desarrollo se separa en tres partes: pirámide de Gauss, pirámide de Laplace y reconstrucción de imagen. En general en cada parte se describe el código que implementa las funciones y se presentan las imágenes resultantes. En la sección de la pirámide de Gauss se describe la implementación de la convolución, el cálculo de máscaras Gaussianas para esta, el submuestreo, la implementación de la pirámide, se prueba y analiza su desempeño en 4 imágenes distintas. En la sección de la pirámide de Laplace se describe la resta de imágenes, el cálculo de la pirámide de la Laplace y se prueba y analiza el desempeño con las mismas 4 imágenes. Finalmente, en la reconstrucción se describe la suma de imágenes, la interpolación (upsampling), la reconstrucción de la imagen y se analiza el desempeño restando la imagen reconstruida con la imagen original; se muestra la imagen resta y se hacen análisis.

En la conclusión se hace una mirada más general de los resultados obtenidos y de sus análisis. Se discute hasta qué punto se observan los comportamientos teórico esperados para los filtros. Se comenta sobre las dificultades superadas para realizar esta experiencia y se finaliza con los aprendizajes y habilidades desarrolladas.

II. Marco Teórico

II-A. Convolución

La operación de convolución es un operador matemático que aplica una señal llamada filtro sobre una señal. Lo que resulta en una funcion filtrada. En el caso más general se expresa con la ecuación 1 y a groso modo representa la magnitud en que se superpone la señal es s(t) sobre la función el filtro h(t) que se invierte y retrasa en el tiempo.

$$s(t) * h(t) = \int_{-\infty}^{\infty} s(\eta)h(t-\eta)d\eta \tag{1}$$

Lo importante para este campo es que una señal posee varias frecuencias. Entonces, el objetivo general de la convolución permitir quitar rangos frecuencias convolucionando la señal con un filtro. Aterrizando este elevado concepto al caso discreto de las imágenes; En este caso la señal es una imagen que se representa como una matriz, con valores entre 0(negro) y 255(blanco). El filtro se denomina mascara o ventana y es una matriz mucho más pequeña que la imagen.

La convolución de imágenes es un operador local, es decir, modifica el valor de cada píxel ponderando los valores de la vecindad del píxel por los valores de la mascara. Esta se interpreta como: posicionarse en un píxel x,y. Sobreponer la mascara sobre el píxel, entonces, quedara un numero de la mascara superpuesto con un píxel de la vecindad. Luego, multiplicar esos números que quedaron superpuestos y sumar todas esas multiplicaciones. Finalmente, el valor del píxel es dicha suma. En la figura 1 se muestra un ejemplo numérico.

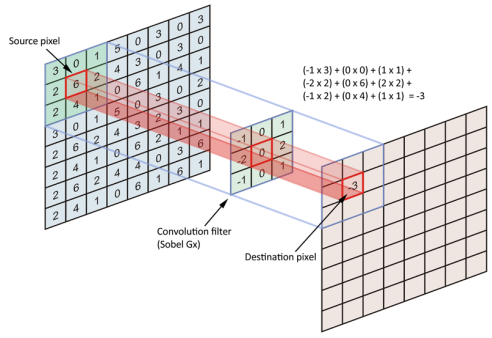

Figura 1. pie de imagen

Al procesar los píxeles pertenecientes a los bordes y esquinas de la imagen, aparece el problema de que no existen alguno números correspondientes a la superposición de la ventana con el píxel. No existe solución para este problema. Lo que si existen son paliativos llamados Padding: como el rellenar los bordes de la imagen con píxeles de valor 0, llamado Zeropadding, la extrapolación de valores de los píxeles agregados con los originales o simplemente omitir los píxeles problemáticos, lo que disminuye el tamaño de la imagen.

II-B. Pirámide de Gauss

La pirámide de Gauss es una herramienta que se utiliza en el área de procesamiento de imágenes en multiresolución. La pirámide de Gauss de una imagen se conforma con las distintas resoluciones posibles para una imagen. En la figura 2 se observa como el ancho y el largo se reducen a la mitad al subir 1 piso de esta.

Dicha pirámide se calcula de la siguiente manera: Su primer piso es la misma imagen, por lo que es la imagen más grande. Luego sus pisos siguientes se construyen iterativamente haciendo siempre lo mismo. El primer piso es con n=1. Para el piso n, se toma la imagen del piso n-1 y se suaviza aplicándole un filtro Gaussiano, luego, esa imagen filtrada se submuestrea tomando los valores de los píxeles pares consiguiendo una imagen de la mitad de las dimensiones de la imagen n-1. Finalmente, se almacena la imagen submuestreada como la imagen del piso n.

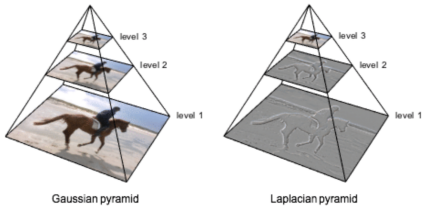

Figura 2. pie de imagen

II-C. Pirámide de Laplace

La pirámide de Laplace también se utiliza en el procesamiento multiresolución de imágenes y contiene la información que se pierde al ir bajando la resolución de la imagen original al construir la pirámide de Gauss. Dicha información puede ser bordes y otras estructuras que se pierden en el proceso. El último piso de la pirámide de Laplace es igual al último piso de la pirámide de Gauss.

Para calcularla se sigue un proceso iterativo. Para el piso n se toma la imagen de la pirámide de Gauss del piso n y se suaviza con filtro Gaussiano. Luego se resta la imagen de la pirámide de Gauss del piso n y la imagen suavizada. Se almacena ese resultado en el piso n de la pirámide de Laplace. Por último, para el **último piso de la pirámide de Laplace** no se hace la resta, el último piso de la pirámide de Laplace es el mismo que el de la pirámide de Gauss.

II-D. Reconstrucción de imagen original

A partir de la pirámide de Laplace es posible reconstruir la imagen usada para construir dichas pirámides (obviamente, conociendo los filtros usados) y usando técnicas de interpolación que agregan error.
Para la reconstrucción se va bajando por los pisos de la pirámide de Laplace: Primero se toma la imagen original (igual a la del último piso de la pirámide de Laplace), se le duplica el ancho y largo y los píxeles que aparecen son calculados mediante interpolación simple (un promedio).En la sección de desarrollo se explicita la formula con que se interpola y la solución de los casos de borde. Luego, esta imagen upsampleada se suma píxel a píxel con la imagen del piso de abajo. Así hasta llegar al piso superior al piso más bajo de la pirámide, en que al hacer la suma se retorna la imagen ya reconstruida y se termina el proceso.

III. Desarrollo

III-A. Pirámide de Gauss

1) ***Implementación código convolución****:* Se implementa la función convolución. La implementación consiste en primero solucionar el problema de los bordes, se calcula cuantos píxeles hay que agregar a los bordes con:

```
1 int anchoMask=mask.rows/2;
2 int altoMask=mask.cols/2;
```

Luego se crea una matriz de ceros cuyo tamaño es el de la imagen original más los bordes que se agregaron dependiendo de las dimensiones del filtro, la que se denominara matriz aumentada. Luego con el ciclo for se rellena la matriz de ceros con la información de la imagen, fijándose que los bordes queden en cero.

```
1 Mat fondo = Mat::zeros(input.rows+2*anchoMask,
2 input.cols+2*altoMask,CV_32FC1);
```

```
  for(int a=0; a<input.rows; a++){
    for(int b=0; b<input.cols; b++){
          fondo.at<float>(a+anchoMask,b+altoMask)
          =input.at<float>(a,b);}}
```

Se implementa la convolución con cuatro ciclos for. Los primeros 2 ciclos for recorren la matriz aumentada, partiendo desde la esquina superior izquierda que corresponde a la imagen original. Se crea la variable float 'pixel' que sirve para guardar el producto punto entre la vecindad del píxel de la imagen y los coeficientes correspondientes del filtro, que se van recorriendo con los restantes 2 ciclos for. Finalmente se asigna el valor de la suma ponderada a la imagen de salida.

```
for (int f=anchoMask; f<fondo.rows-anchoMask; f++){
  for (int c=altoMask; c<fondo.cols-altoMask; c++){
    float pixel=0.0;
          for(int mf=0; mf<mask.rows; mf++){
            for (int mc=0; mc<mask.cols; mc++){
                  pixel+=fondo.at<float>(f+mf-anchoMask,
                  c+mc-altoMask)*mask.at<float>(mf,mc);}}
        output.at<float>(f-anchoMask,c-altoMask) = pixel;}}
```

2) ***Cálculo de las máscaras****:* Se debía implementar el calculo de una máscara horizontal de *1 x n* y otra vertical de *n x 1*; para implementar la convolución por una y luego por la otra lo que reduce el tiempo de procesamiento a que si se involucionara soló una vez por la multiplicación de ambas máscaras (resultando una máscara grande de *n x n*).

La implementación se hace con el siguiente código. Primero se calcula la mitad del ancho para armar un arreglo de coordenadas x a evaluar en la función:

$$g[x] = e^{-\frac{x^2}{2\sigma^2}} \tag{2}$$

El coeficiente raíz de pi no importa porque al normalizar los coeficientes finales se termina simplificando. Si la máscara es de 1xn se crea un arreglo del tipo: -int(n/2),..,0,..,int(n/2). Esto asegura que la media sea 0.

Lo más importante es que los coeficientes de la máscara se calculan evaluando los números de dicho arreglo en una Gaussiana unidimensional de media 0 y varianza según indique lo requerido. Luego se normalizan sumando los números que dieron las evaluaciones. En el segundo for se recorre la máscara resultante y se divide cada uno de sus términos por la suma del total. Con lo que se asegura que resulta una máscara normalizada.

```
Mat mask = Mat::zeros(1, width, CV_32FC1);
int anchoMask=mask.cols/2;
float suma = 0.0;
float mask_coef[width];
for (int i = -anchoMask ; i<= anchoMask ; i++){
  mask_coef[i+anchoMask] = 1/exp((i*i)/(2*sigma*sigma));
  suma = suma + exp(-(i*i)/(2*sigma*sigma));}
for (int c = 0 ; c < width ; c++){
  mask.at<float>(0,c) = mask_coef[c]/suma;}
return mask;
```

La implementación para la máscara vertical es análoga solo que en este caso se crea una máscara de *nx1* y se recorre variando las filas.

3) ***Suavizado de imágenes****:* El suavizado de imágenes se implementa con 2 convoluciones que es la forma más rápida de hacerlo. Primero se convoluciona la imagen origina con la cara horizontal, luego, ese resultado se convoluciona con la máscara vertical. Resultando una imagen suavizada usando un filtro Gaussiano.

```
Mat do_blur(Mat input, double sigma, int height){
  Mat mask_horiz, mask_vert;
  mask_horiz = compute_gauss_horiz(2,7);
  mask_vert = compute_gauss_vert(2,7);
  input = convolution(input , mask_horiz);
  result = convolution(input , mask_vert);
  return result;}
```

Un punto importante es que la función do_blur recibe un una variable tipo double como sigma. Es importante aclarar que no se producen problemas al operar variables de distinto tipo porque ese double va evaluado dentro de la función exp()(en la función que hace la máscara). Como exp() es nativa no presenta problema al recibir un double. En general el código anterior es solo llamar funciones.

4) ***Submuestreo****:* El submuestreo recibe una imagen input y crea una imagen (un objeto Mat) de la mitad de las dimensiones. El caso borde es cuando al menos una de las dimensiones es impar, en este caso la dimensión resultante es la parte entera de la mitad de la dimensión original. Esto se implementa con la división por int *input.rows/2* al dividir por el int 2; si se dividiera por el float 2.0 resultaría un decimal y abría un error porque las dimensiones tienen que ser números enteros porque representan píxeles.

```
Mat result = Mat::zeros(input.rows/2,input.cols/2,CV_32FC1);
  for (int f=0; f<result.rows; f++){
        for (int c=0; c<result.cols; c++){
          result.at<float>(f,c) = input.at<float>(2*f,2*c);}}
return result;
```

El submuestreo es un doble ciclo *for* anidado que primero se posiciona en la fila 0 de la imagen de salida; para cada una de sus columnas(de salida), se posiciona en la fila 0 de la imagen de entrada y va recorriendo las columnas pares de esta, para copiarlas en la imagen resultante (*Mat result*). Luego salta a la siguiente fila de la imagen de salida, salta a la siguiente fila par de la imagen de entrada (fila 2) y hace la mismo copia con las columnas pares. Es importante que se considera la fila 0 y la columna 0 como pares.

5) ***Pirámide de Gauss****:* La pirámide de Gauss se implementa con un arreglo dinámico de objetos Mat que se crea con *vector<Mat> gausspyramid;*. La función recibe la imagen de entrada *Mat input* y un int nlevels que es la cantidad de niveles. El primer nivel es el nivel 0 y es el objeto Mat del índice 0 del arreglo de salida que representa la pirámide de Gauss.

Se crea una imagen llamada current igual a la imagen de entrada (usando *.clone()*) y luego se agrega a la pirámide

con *.push_back(current);*. El método *array.push_back(a)* agrega el objeto a la última posición del array: *array*. Como al crearlo se crea vació, al usar *.push_back()* sobre un array vació, el objeto que queda en el índice 0 del array. Esto es lo que queremos al poner la imagen original en el primer nivel de la pirámide.

Los siguientes niveles se calculan iterativamente dentro del ciclo *for*. Primero se crean dos imágenes (*convolucionada y submuestreada*) para el procesamiento. Luego, cada piso se calcula tomando el piso anterior (*gausspyramid[i-1]*) y aplicándole el filtrado Gaussiano con una máscara de 7x7 y sigma=2 (en el código línea 1). Luego, se submuestrea la imagen suavizada (en código linea 2). Finalmente, se agrega esa imagen submuestreada a la pirámide con *.push_back()*(en código linea 3). Así sucesivamente, hasta completar la cantidad de niveles requerida.

```
1  vector<Mat> compute_gauss_pyramid(Mat input, int nlevels){
2    vector<Mat> gausspyramid;
3    Mat current = input.clone();
4    gausspyramid.push_back(current);
5    for (int i = 1; i < nlevels; i++){
6      Mat convolucionada, submuestreada;
7      convolucionada = do_blur(gausspyramid[i-1],2.0,7);//(1)
8      submuestreada = subsample(convolucionada);//(2)
9      gausspyramid.push_back(submuestreada);}//(3)
10 return gausspyramid;}
```

6) ***Prueba de calculo de pirámide de Gauss:*** Para calcular las pirámides de Gauss de las 4 imágenes entregadas se crean dos códigos: el que invoca 4 veces a la función que calcula la pirámide de Gauss con diferentes imágenes e entrada y el que guarda la pirámide de Gauss.

Para automatizar el llamado a la función *compute_gauss_pyramid* se implementa el siguiente código. Primero se cargan las imágenes en GrayScale en formato cv_32FU1, luego estas imágenes son agrupadas en un arreglo de objetos Mat que contiene las imágenes cargadas. Luego, para guardar los resultados,se crea un arreglos de string con los nombres de las imágenes. Luego, con un ciclo *for* de 4 iteraciones, en cada iteración se calcula la pirámide de Gauss de 5 niveles y se guarda en un arreglo dinámico de objetos Mat. Luego, cada imagen de ese arreglo se guarda (se actualiza sino crea un nuevo archivo) usando la función save_gauss_pyramid().

```
Mat images_mat_names[4]={figurasLLenas, figurasBorde,
monalisa, corteza};
string images_str_names[4] = {"figurasllenas",
"figurasborde", "monalisa" , "corteza"};
for (int i=0; i<4 ;i++){
  vector<Mat> gausspyramid =
  compute_gauss_pyramid(images_mat_names[i], 5);
  save_gauss_pyramid(gausspyramid,images_str_names[i]);}
```

La función save_gauss_pyramid() recibe la pirámide de Gauss y un string con el nombre de la imagen. En cada iteración se guarda una imagen, primero se escala la imagen usando la función scale_abs(). Luego crea un objeto de clase *stringstream* llamado *filename* que sirve para concatenar el string que sera el nombre del archivo. Luego, se pasa de *stringstream* a *string* usando el método *.str()*. Finalmente se guarda usando imwrite.

```
void save_laplace_pyramid(vector<Mat> pyramid ,
const std::string& image_name){
  for (int i = 0; i < pyramid.size(); i++){
    Mat scaled = scale_abs(pyramid[i], 5.0);
        stringstream filename;
        filename<<image_name<<"_laplace_"<<i<<".bmp";
        imwrite(filename.str(), scaled);}}
```

La función scale_abs() Es importante escalar las imágenes porque en muchos casos producto del procesamiento las imágenes terminan con píxeles negativos, esta función usa un doble ciclo *for* para ir tomando el valor absoluto a los píxeles, y luego, los multiplica por un factor porque muchas veces los píxeles tienen valores muy bajos y escalarlos sirve para visualizar mejor las imágenes. Un punto importante es que en un principio esta función estaba implementada con el factor de escalamiento como una variable de tipo double, lo que produce errores al multiplicar una variable double por una variable float; esto se soluciona cambiando el tipo de variable con que ingresa el factor de escalamiento de double a float, siendo esto la solución más simple de implementar.

```
Mat scale_abs(Mat input, float factor){
  Mat output = input.clone();
  for(int f=0 ; f<input.rows; f++){
    for(int c=0; c<input.cols; c++){
          output.at<float>(f,c) =
          abs(input.at<float>(f,c)) * factor;}}
  return output;}
```

7) ***Análisis de desempeño de pirámide de Gauss:*** El filtro gaussiano es un filtro pasa bajo, luego deja pasar las frecuencias espaciales bajas y no deja pasar las frecuencias espaciales altas(bordes). Teóricamente, la hipótesis es que se esperan imágenes más borrosas. A continuación se presentan los resultados:

Figura 3. Pirámide de Gauss para "monalisa.png".

La imagen de la monalisa se caracteriza por presentar varias regiones contantes, con lo que predominan las frecuencias espaciales bajas, por ejemplo: frente, pómulos, cuello y fondo de cielo. Se observa como se va poniendo

más borrosa debido a la aplicación sucesiva del filtro pasa bajo. Es destacable que para submuestrear es imprescindible filtrar con un pasa bajo antes para de alguna forma "esparcir la información de los píxeles en sus vecindades". De esta manera se conserva la forma de la imagen y como se observa en las figuras 3 se reconoce el contenido de la imagen, es decir, se reconoce a que clase o que instancia de una clase pertenece la imagen. Además, como se convoluciona por un filtro pasa bajo, se dejan pasar las frecuencias bajas que son las que mayormente describen la clase de esta imagen de una cara y eso permite que se reconozca la figura de la monalisa a lo largo de las imágenes la pirámide.

La imagen de corteza presenta una gran cantidad de bordes, es decir, se caracteriza por un alto contenido en frecuencias altas. Se observa que se construye satisfactoriamente la pirámide de Gauss. Sin embargo, hasta el segundo piso es posible reconocer el contenido de la imagen. Desde el piso 3 a sus superiores es más complicado reconocer que se trata de una corteza sin tener información a priori del contenido de la imagen. Esto sucede porque el filtrado Gaussiano, al ser pasa bajo, no deja pasar las frecuencias altas que determinan los bordes de la imagen. Por eso a partir de la tercera imagen se observan en su mayoría las frecuencias bajas de la imagen original, no se observan las frecuencias altar y por eso es más difícil identificar que es una corteza.

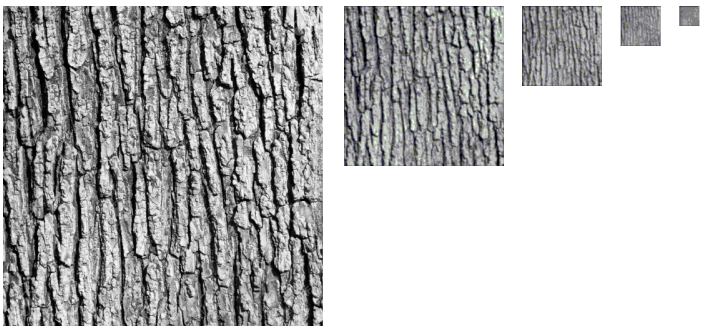

Figura 4. Pirámide de Gauss para "corteza.png".

La figura de bordes llenos presenta regiones de textura constante, por lo que en ella predominan las frecuencias bajas. Se logra construir la pirámide de Gauss. A medida que se va convolucionando se dejan pasar las frecuencias bajas lo que logra que se conserve la información de las formas. Se observa que se difuminan los bordes de las figuras geométricas por el filtrado pasa bajo. En el último piso de la pirámide no se distinguen bien las formas porque la imagen de la que proviene es de 32x16, que al convolucionarla con el filtro equivalente a una máscara de 7x7, no se procesan todos los bordes y se suaviza demasiado, con lo que no se pueden distinguir las formas en la imagen submuestreada resultante de 16x8.

Figura 5. Pirámide de Gauss para "figurasllenas.png".

Finalmente, en la imagen de soló los bordes se presentan más frecuencias altas que la anterior debido a los bordes. Se logra construir la pirámide y se observa que a medida que subimos de nivel los bordes se van haciendo más difusos. En el segundo piso se pierde la información de las letras y ya en el último piso se pierde la información de las formas, debido al gran tamaño de la máscara 7x7 en comparación al tamaño de las formas.

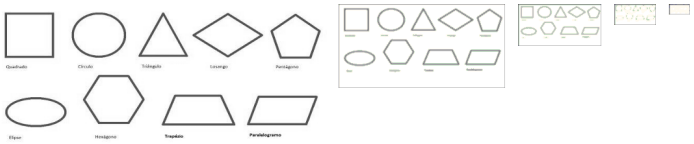

Figura 6. Pirámide de Gauss para "figurasborde.png".

III-B. Pirámide de Laplace

1) ***Resta de imágenes****:* La resta de imágenes se implementa con un doble ciclo *for* que va restando y guardando el resultado píxel a píxel y guardando en la imagen output, que para que tenga las misma dimensiones que las imágenes de entrada se crea usando la función .clone(). Debido a que se están restando valores, los casos de borde se dan cuando quedan píxeles negativos. Estos se tienen que tratar para efectos de guardar la imagen en la función *save_laplace_pyramid* esta función recibe la pirámide de Laplace, en los casos de borde con píxeles negativos, y les aplica la función *scale_abs* lo que les calcula el valor absoluto y los pondera. Esto soluciona los casos de píxeles negativos y permite guardar las imágenes en archivos bitmap (bmp). Sin embargo, para efectos de reconstrucción de la imagen original es necesario conservar estos píxeles negativos en el array de mat que representa la pirámide de Laplace. La parte más importante del código es la siguiente linea:

```
output.at<float>(f,c)=input1.at<float>(f,c)-input2.at<float>(f,c);
```

Además se cuenta con un bloque *if* para verificar que las imágenes a restar tengan la misma dimensión, para que de esta forma la resta tenga sentido:

```
if(input1.cols!=input2.cols || input1.rows !=input2.rows){
  cout << "subtract() called with different image sizes";
  return Mat();}
```

2) ***Pirámide de Laplace:*** La pirámide de Laplace se implementa con una función que recibe la imagen original y el numero de niveles, no es necesario introducir la pirámide de Gauss. Primero se crean dos arreglos dinámicos que contendrán la pirámide de Laplace y la de Gauss, esta última se calcula soló para hacer más simple el entendimiento del código, se le pudo haber puesto otro nombre e igual se podría calcular la pirámide de Laplace sin recibir como entrada la pirámide de gauss. Luego, crea una imagen llamada current que sera la imagen que procesaremos he iremos agregando en la pirámide de Laplace. Cada piso se arma iterativamente con un ciclo for. Para el primer nivel n=1, se toma el primer piso de la pirámide de Gauss (igual a la imagen original) y se suaviza. Luego se resta el piso de la pirámide de Gauss sin suavizar con el suavizado y el resultado se guarda en la pirámide de Laplace usando el método .push_back(). Así iterativamente hasta llegar al penúltimo piso de la pirámide de Laplace, esto se implementa con la condición del for i < nlevels. Para el último piso de Laplace, se toma el último piso de la de Gauss y simplemente se se inserta usando .push_back() al último piso de Laplace (el último piso de Gauss es igual al último piso de Laplace.)

```
vector<Mat> compute_laplace_pyramid(Mat input, int nlevels){
  vector<Mat> gausspyramid;
  vector<Mat> laplacepyramid;
  Mat current = input.clone();
  Mat filtered;
  gausspyramid.push_back(current);
        for (int i = 1; i < nlevels; i++){
          filtered = do_blur(gausspyramid[i-1],2.0,7);
      Mat resultado_resta=subtract(gausspyramid[i-1],filtered);
          laplacepyramid.push_back(resultado_resta);
          current = subsample(filtered);
          gausspyramid.push_back(current);
        }
        laplacepyramid.push_back(current);
        return laplacepyramid;}
```

3) ***Guardado de imágenes de Laplace:*** El guardado de imágenes de Laplace se implementa de forma parecida a guardar las imágenes de Gauss. Soló que en este caso se tiene que escalar las imágenes antes de guardarles, lo que se implementa en la linea (1). También, se crea un objeto stringstream para concatenar el nombre del archivo, que en este caso debe partir con Laplace. Lo importante es que no se tiene que escalar la última imagen de la Laplace, lo que se implementa con los límites del for que llegan según la condición i < pyramid.size()-1. Por ejemplo, si fueran 5 niveles llega a hasta la posición del arreglo i=3.

```
void save_laplace_pyramid(vector<Mat> pyramid,
const std::string& image_name){
  for (int i = 0; i < pyramid.size()-1; i++){
    Mat scaled = scale_abs(pyramid[i], 5.0);(1)
    stringstream filename;
    filename<<"laplace_"<<image_name<<"_"<<i<<".bmp";
    imwrite(filename.str(), scaled);
  }
}
```

4) ***Prueba de calculo de pirámide de Laplace:*** La automatización se la pruebas se hace con la misma estructura que con la pirámide de Gauss cambiando los nombres de funciones pertinente.

5) ***Prueba de calculo de pirámide de Gauss:*** En el fondo las imágenes de Laplace son las resultantes de restar la imagen sin suavizar con la suavizada. Entonces, estas imágenes contiene lo que se pierde al aplicar el filtrado pasa bajo. La imagen de la monalisa presenta valores altos (blancos) en los bordes de la cabeza de la monalisa. Esto se debe a que el filtro por ser pasa bajo no dejo pasar las frecuencias altas de esos bordes y es por eso que las vemos en la imagen de Laplace. El tema de los bordes blancos de la imagen se explica porque se usa zeropadding al convolucionar, entonces, la imagen suavizada se rellena con ceros en sus bordes no convolucionados, entonces, en la imagen de Laplace vemos los bordes del cuadro blancos porque la diferencia entre un píxel que debe tener algún valor acorde a la imagen y cero es muy grande en comparación a la diferencia entre un píxel y ese mismo píxel convolucionado. El ultimo piso es prácticamente de blanco constante porque es igual al ultimo piso de la pirámide de Gauss (producto de una convolución pasa bajo) y tiene los bordes negros por el rellenos con ceros.

Figura 7. Pirámide de Laplace para "monalisa.png".

La imagen de corteza presenta muchas más altas frecuencias, entonces, en la imagen de Laplace vemos mucho más contenido de alta frecuencia porque es lo que NO deja pasar el filtro Gaussiano pasa bajo. También, se observa que la última imagen es casi en su totalidad blanca, destacándose en un circulo rojo el píxel negro que sobrevivió a las convoluciones, esto se explica porque al convolucionar una imagen de color casi constante por un filtro pasa bajo, se esparcen aún más los colores y se ve un color constante bastante cercano al blanco. No es que la imagen final sea blanca, soló que, como el fondo de hoja es blanco, es difícil percibir los colores cercanos al blanco.

Las figuras llenas presentan altas frecuencias en sus bordes los cuales son posibles de observar en las imágenes de Laplace, incluso en estas, es posible observar como se van difuminando los bordes debido a la convolución pasa bajo. Los bordes de las imágenes (en global) se observan blancos debido al zeropadding, que en este caso funciona

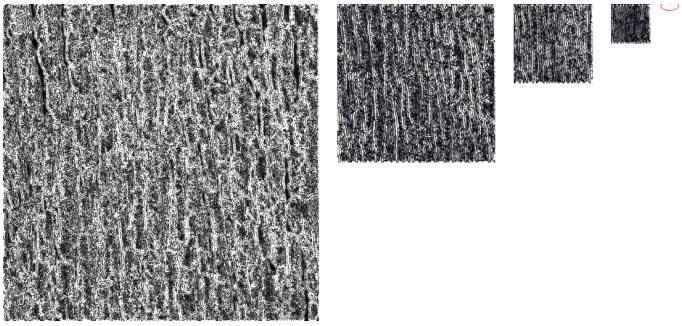

Figura 8. Pirámide de Laplace para "corteza.png".

de la misma forma que se explico en el caso de la imagen monalisa.

Figura 9. Pirámide de Laplace para "figurasllenas.png".

Finalmente, para la imagen con figuras de soló los bordes, al convolucionarlas se espera que los bordes se esparzan luego, al hacer la resta se va a observar valores altos en los píxeles que antes no pertenecían al borde y que al convolucionar "agarran color esparcido". Esto es lo que se observa a lo largo de las imágenes de Laplace en que los bordes se van haciendo cada vez más anchos en proporción a toda la imagen. En el primer piso se observa la información de las letras, no así en el segundo, porque en el segundo piso de la pirámide de Gauss la información (la forma) de las letras ya estaba perdida debido al primer suavizado.

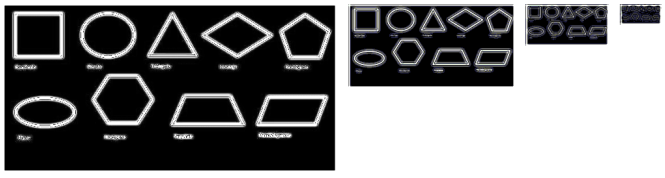

Figura 10. Pirámide de Laplace para "figurasborde.png".

III-C. Reconstrucción de imagen original

1) ***Suma de imágenes:*** La suma de imágenes se implementa de la misma forma que la resta, solo que cambiando el signo menos por un más. También, se le añade el detector de igual tamaño de imágenes. La linea más relevante es:

```
output.at<float>(f,c)=input1.at<float>(f,c) + input2.at<float>(f,c);
```

2) ***Duplicación de tamaño usando interpolación:*** La interpolación se implementa usando el promedio. Se basa en que cada píxel de la imagen nueva puede ser calculado promediando píxeles de la imagen anterior. Primero se parte creando una imagen del doble de tamaño. Luego se recorre esta nueva imagen con un doble for, para cada píxel de la nueva imagen sé calcular los coeficientes col1, col2, row1, row2 que indican que puntos de la imagen anterior determinan el valor del nuevo píxel.

Los casos de borde, se capturan con los if, son en los extremos de la nueva imagen: En la última columna de la nueva imagen no hay una columna más a la izquierda (en l imagen original) para promediar, por lo que se repite el píxel de la ultima columna (de la imagen original) para hacer el promedio.

Análogamente en la última fila, en la imagen pequeña no hay una fila más abajo para hacer el promedio, entonces, se repite el píxel de la última fila de la imagen original. El doble caso borde es en el extremo inferior derecho en que se repiten el píxel de la última fila y de la última columna, con lo que el píxel reconstruido es la cuarta parte del píxel de la esquina inferior derecha de la imagen original. Esto genera que los bordes derecho e inferior de las imágenes reconstruidas tengan lineas negras dependiendo de la cantidad de niveles de las pirámides.

```
Mat upsample(Mat input){
  Mat output = Mat::zeros(input.rows*2,input.cols*2,CV_32FC1);
  for(int f=0 ; f<output.rows ; f++){
    for(int c=0 ; c<output.cols ; c++){
          int col1, col2, row1, row2;
          float col1row1, col1row2, col2row1, col2row2;
          col1 = int(c/2);
          if(c == output.cols-1){
            col2 = int (c/2);
          } else { col2 = int ((c+1)/2); }
          row1 = int (f/2);
          if(f == output.rows-1){
            row2 = int (f/2);
          } else{ row2 = int ((f+1)/2); }
          col1row1=input.at<float>(row1,col1);
          col1row2=input.at<float>(row2,col1);
          col2row1=input.at<float>(row1,col2);
          col2row2=input.at<float>(row2,col2);
          output.at<float>(f,c) = (col1row1 + col1row2 +
          col2row1 + col2row2)/ 4.0;
        }
  }
 return output;}
```

3) ***Implementación de la reconstrucción:*** La reconstrucción es la funcion reconstruc, que recibe la pirámide de Laplace. Parte con el ultimo nivel haciendo la interpolación que duplica su tamaño. Luego ingresa a un ciclo for que toma la imagen del piso inferior y se la suma a la imagen upsampleada. Así sucesivamente hasta tomar la imagen el ultimo piso, sumarla con la upsampleada y resultando la imagen reconstruida.

```
Mat reconstruct(vector<Mat> laplacepyramid){
  Mat output = laplacepyramid[laplacepyramid.size()-1].clone();
  for (int i = 1; i < laplacepyramid.size(); i++){
        int lev = int(laplacepyramid.size()) - i - 1;
        Mat upsampled;
        upsampled = upsample(output);
    output = add(upsampled,laplacepyramid[lev]);
  }
return output;}
```

*4) **Resultados y análisis de la reconstrucción**:* Cada una de las siguientes imágenes, la imagen original es la de la izquierda, la del centro es la reconstruida y la de la derecha es la diferencia entre la reconstruida y la original, es decir, el delta imagen.Entonces, píxeles cercanos a blancos en la imagen delta indican que en esas zonas hay mucha diferencia con la imagen origina, es decir, es esas zonas la reconstrucción quedo mal.

La reconstrucción de monalisa presenta excelente resultado debido a que general es una imagen en que predominan las frecuencias altas. Se observan manchas negras debido a que se esparció el negro del zeropadding en cada convolución. En el delta imagen se observa bordes blancos debido al relleno con ceros de la convolución.

La imagen de corteza se obtiene un reconstrucción

Figura 11. Reconstrucción de "monalisa.png".

aceptables. se observa que los bordes se aclaran debido a que se suavizan por los filtrados pasa bajo. Al igual que en el caso anterior la imagen delta muestra borde inferior y derecho blanco debido a la aproximación de casos bordes en el upsampling.

Figura 12. Reconstrucción de "corteza.png".

La reconstrucción de la figuras rellenas presenta zonas más oscuras debido a que se esparció el color de las figuras llenas (sobre todo en la parte superior). La imagen delta muestra los errores por el upsampling en los bordes inferior y derecho

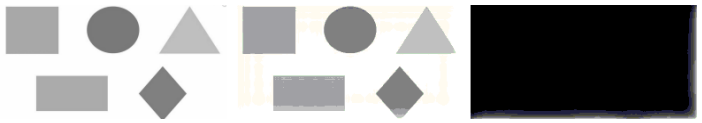

Figura 13. Reconstrucción de "figurasllenas.png".

Finalmente, la reconstrucción de las figuras con solo el borde es bastante buena, soló se esparció el color en los bordes superior y izquierdo, debido a que hay menos color que esparcir porque hay menos frecuencias bajas que con las figuras rellenas. También, en la imagen delta se observa el efecto del upsampling.

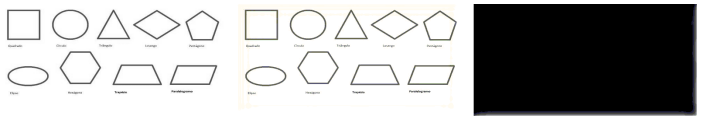

Figura 14. Reconstrucción de "figurasborde.png".

IV. Conclusión

Finalizada la experiencia se logran verificar todas las hipótesis que estaban de acuerdo a la teoría enseñada en clases con respecto al comportamiento del filtro implementado y la reconstrucción implementada.

Se logro implementar el filtro gaussiano convolucional, la reconstrucción y las demás funciones necesarias. Se obtienen todas las imágenes pedidas. Se logra observar el grado en que el filtro pasa bajo gaussiano suaviza las imágenes y su importancia para el submuestreo. Se logro observar la perdida de información al comparar las imágenes resultantes con las iniciales y se explicó el efecto del uso de Zeropadding que "esparce los bordes negros" observando la imagen final. Los resultados de imágenes finales reconstruidas son bastante buenas (observando la imagen diferencia). Con lo que se concluye que se implemento satisfactoriamente todos los procesos.

En esta experiencia se aprendió a programar en c++, se aprendió a utilizar la distribución de Linux Ubuntu, se practicó el uso de Látex. Se programan los primeros scripts de rápida ejecución en la maquina. Aprender a programar en c++ es importante porque se obtiene la mayor velocidad en procesamiento de imágenes en tiempo real.

Las dificultades que se encontraron durante el desarrollo de la experiencia fue que solo se tenia conocimiento de programación en Python, que es un lenguaje de nivel más alto y más cercano al usuario, es decir, más intuitivo de

programar. Para esta experiencia se debía aprender C++ que es un lenguaje de más bajo nivel(no el más bajo). Sin embargo, la programación y el debug se hacen más complejos. También, se presentaron dificultades con la instalación de la biblioteca, del compilador y del entorno de escritura. Finalmente, las dificultades se solucionaron preguntando a los compañeros de clase, investigando en Stackoverflow y en la documentación de OpenCV.

Fue una difícil tarea. Pero cuando reflexiono si existe alguna otra experiencia del procesamiento de imágenes que sea más fundamental, llego a la conclusión de que esta no existe. Estoy contento con mi mismo, porque logre aprender un lenguaje nuevo, ocupar un sistema operativo nuevo y mi mente se abrió a una avalancha de ideas y aplicaciones que voy a abordar gracias a lo que he aprendido en esta tarea.