Tarea 1:
Pirámides de Gauss y Laplace

Fecha de entrega: Domingo 05 de septiembre, 23:59 hrs.

Estudiante: Francisco Molina L.
Profesor: Javier Ruiz del Solar
Auxiliar: Patricio Loncomilla Z.
Semestre: Primavera 2021

Índice

1. Introducción

En el presente informe se desarrollan las actividades correspondientes a la Tarea 1 del curso EL7008-1 del semestre de primavera de 2021. Este informe comprende el estudio, implementación y el análisis de las pirámides de Gauss y Laplace, y de la reconstrucción de imágenes a partir de estas.

El principal objetivo de este informe es lograr implementar mediante programación de bajo nivel las representaciones multi-escala (Gauss y Laplace) y la reconstrucción de imágenes a partir de dichas representaciones.

La **Sección 2** contiene un breve resumen de qué son las pirámides y los pseudo-códigos necesarios para implementarlas. En las **Secciones 3, 4 y 5** se detallan las funciones implementadas en Python, adjuntas en el notebook Tarea1.ipynb, necesarias para calcular y mostrar las pirámides de Gauss, las pirámides de Laplace, y la reconstrucción de las imágenes. En la **Sección 6** se presentan los resultados obtenidos de la implementación en Python, en la **Sección 7** se realiza un análisis de dichos resultados, y en la **Sección 8** se presentan las conclusiones del informe.

2. Marco Teórico

2.1. Pirámides en Procesamiento Digital de Imágenes

Las pirámides son representaciones multi-resolución calculadas a partir de una sola imagen. Estas representaciones consisten en un conjunto de imágenes organizadas en niveles, donde cada nivel tiene la mitad del ancho y alto que el nivel anterior [1], por ende el nombre de pirámides, como se muestra en la **Figura 1**.

Esta tarea se centra en implementar el cálculo de dos tipos de pirámides, las pirámides de Gauss y las de Laplace, y en el proceso de reconstrucción de imágenes a partir de dichas pirámides.

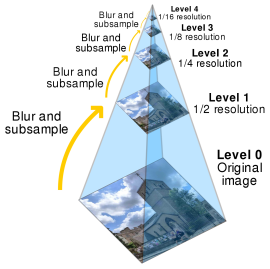

Figura 1: Pirámide

2.2. Pirámides de Gauss

Las pirámides de Gauss contienen las imágenes que representan las distintas resoluciones posibles de una misma imagen [1]. Para calcular cada nivel de la pirámide se deben seguir los siguientes pasos:

1. Suavizar la imagen del nivel anterior
2. Submuestrear la imagen suavizada.
3. Almacenar la imagen resultante en la pirámide de Gauss.

2.3. Pirámides de Laplace

Las pirámides de Laplace contienen la información que se pierde al ir bajando la resolución en el proceso de formación de la pirámide de Gauss. Además, se suele agregar en el último nivel la última imagen de la pirámide de Gauss [1]. Para calcular cada nivel de la pirámide se deben seguir los siguientes pasos:

1. Elegir y suavizar la imagen de la pirámide de Gauss correspondiente.
2. Restar la imagen de la pirámide de Gauss antes y después de suavizarla.
3. Almacenar el resultado de la resta en la pirámide de Laplace.
4. [Sólo para el último nivel] Almacenar la última imagen de la pirámide de Gauss en la pirámide de Laplace.

2.4. Reconstrucción

A partir de la pirámide de Laplace se puede reconstruir la imagen original utilizada para crear las imágenes [1]. Para lograr esto se deben seguir los siguientes pasos:

1. Elegir el último piso de la pirámide de Laplace.
2. Para cada nivel restante, en orden descendiente:
   a) Duplicar el tamaño de la imagen.
   b) Sumar la imagen correspondiente de la pirámide de Laplace.

3. Cálculo de las pirámides de Gauss

3.1. Calcular la convolución

Se programa una función que calcule la convolución entre una imagen de entrada y una máscara. Para optimizar el tiempo de cómputo, esta función debe ser programada con *cython*. Se adjunta la función programada en el anexo, en el **Listing** 1.

Hacer una convolución consiste en aplicar el paradigma de la ventana deslizante, calculando el producto punto entre la máscara y los pixeles de la imagen que están dentro de la ventana, como se muestra en la **Figura** 2:

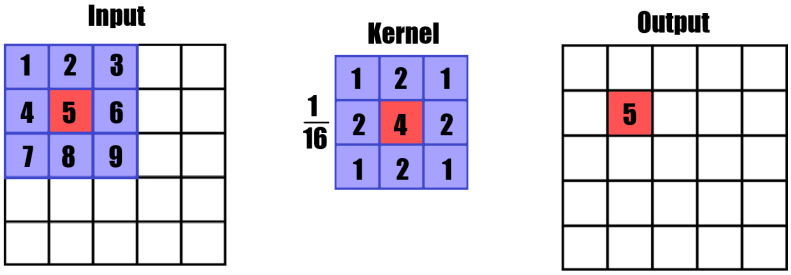

Figura 2: Convolución

Realizar la convolución de este modo implica que la imagen resultante será de menor tamaño [2], pues el kernel se sale de la imagen en los bordes. En general, para una imagen de dimensiones (N,N) y un kernel de dimensiones (k,k), la imagen resultante tiene un tamaño de:

$$(N - k + 1, N - k + 1)$$

Esto puede evitarse aplicando **padding**, rellenar los pixeles que quedan fuera de la imagen con algún valor arbitrario, como se muestra en la **Figura** 3:

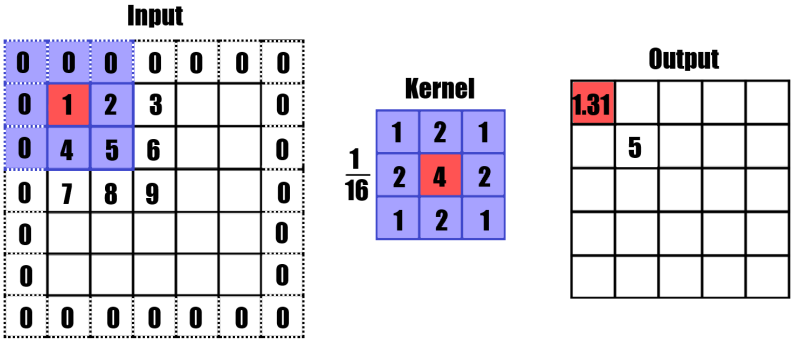

Figura 3: Convolución con zero padding

De esta forma, el kernel se puede aplicar en todos los pixeles de la imagen y se logra preservar el tamaño original de esta. La implementación mostrada en el Listing 1 utiliza zero padding (rellenar con ceros).

Para implementar la convolución en Python es necesario utilizar 4 *for loops*. Dos para recorrer, pixel a pixel, la imagen original, y los otros dos para recorrer el kernel cuando ya se está en el pixel central. Sin embargo, esto no contempla cómo se recorre la sub-sección de la imagen bajo el kernel (la sombra del kernel), es decir, cómo se encuentran los pixeles que hay que multiplicar con el kernel.

Para esto en la implementación se utilizan ***offsets***, valores que indican cuantos pixeles hay que desplazarse desde el pixel central, tanto en X como en Y, para llegar a la esquina superior izquierda de la sombra del kernel.

Los kernels son de dimensiones impares, para que así puedan tener un pixel central. Analizando el caso del kernel 3x3 y el kernel 5x5, como se muestra en la **Figura 4**:

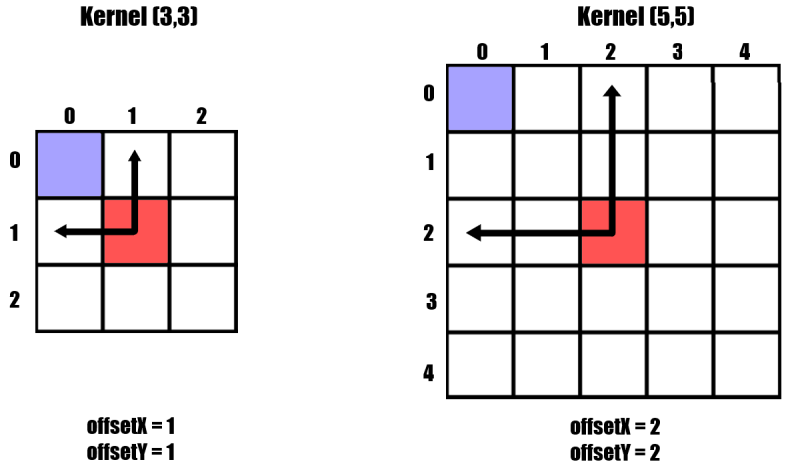

Figura 4: Offsets

Es fácil darse cuenta que el pixel en la esquina superior izquierda del kernel de dimensiones (k,k) siempre está a una distancia:

$$(\left\lfloor \frac{k}{2} \right\rfloor, \left\lfloor \frac{k}{2} \right\rfloor)$$

del kernel central.

Es por esto que en el código mostrado en el Listing 1 se definen las variables:

```
posX = (x - offsetX) + kernelX
posY = (y - offsetY) + kernelY
```

Donde (x,y) son las coordenadas del pixel central, restarles offsetX y offsetY realizan la traslación a la esquina superior izquierda, y los índices kernelX y kernelY son los que recorren, al mismo tiempo, el kernel y su sombra.

3.2. Generar una máscara Gussiana bidimensional

Para suavizar las imágenes se necesita que el kernel usado en la convolución sea Gaussiano. Por lo que hay que encontrar un modo de discretizar una función Gaussiana en dos dimensiones.

En primer lugar, la función Gaussiana en dos dimensiones está definida por [3]:

$$G(x,y) = A \cdot exp(-(\frac{(x-x_0)^2}{2{\sigma_X}^2} + \frac{(y-y_0)^2}{2{\sigma_Y}^2})) \tag{1}$$

Donde (x_0, y_0) corresponden al centro de la Gaussiana, y σ_X y σ_Y sus respectivas desviaciones. Si la Gaussiana se centra en $(0,0)$, se tiene el mismo σ para x e y, y se normaliza, la función es:

$$G(x,y) = \frac{1}{2\pi\sigma^2} \cdot exp(-\frac{x^2+y^2}{2\sigma^2}) \tag{2}$$

Esta función se representa en la implementación en el código mostrado en el **Listing 2**.

Ahora, para discretizar esta función, basta con generar una matriz de un tamaño arbitrario, centrada en cero, y evaluarla en la función gaussiana. En la **Figura 5** se muestra un ejemplo de cómo generar un kernel Gaussiano de (3,3) con un σ de 1:

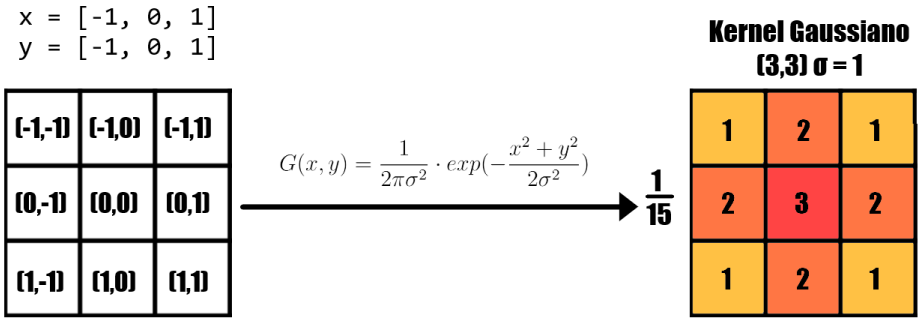

Figura 5: Kernel Gaussiano (3,3), $\sigma = 1$

Este procedimiento se representa en la implementación en el código mostrado en el **Listing 3**. Se recorren dos *for loops*, uno para x y el otro para y, con valores en $(0, k)$, donde k es el tamaño deseado del kernel; Se utiliza el mismo truco del *offset* explicado en la Sección 3.1 para que la matriz quede centrada en $(0,0)$; Se evalúan los valores correspondientes en la función *gauss2d*, y se van sumando a la variable *sum*, para al final normalizar la matriz y así obtener el kernel.

3.3. Suavizar una imagen

Como se mencionó en la **Sección 3.2**, suavizar una imagen consiste en realizar una convolución entre la imagen y un kernel Gaussiano.

Dado que en la **Sección 3.1** se explicó cómo realizar una convolución y en la **Sección 3.2** cómo generar un kernel Gaussiano, en esta sección basta con combinar ambas implementaciones. Por ende, la implementación mostrada en el **Listing 4** simplemente usa las funciones *convolution_cython* y *compute_gauss_mask_2d*.

3.4. Submuestrear una imagen

Se requiere una función que submuestree una imagen, a la mitad de su tamaño, para lo cual hay que copiar sólo los pixeles pares en la imagen de salida [1]. En la **Figura 6** se muestra un ejemplo del resultado de aplicar la función *do_subsample* a una imagen de entrada de dimensiones (5,5):

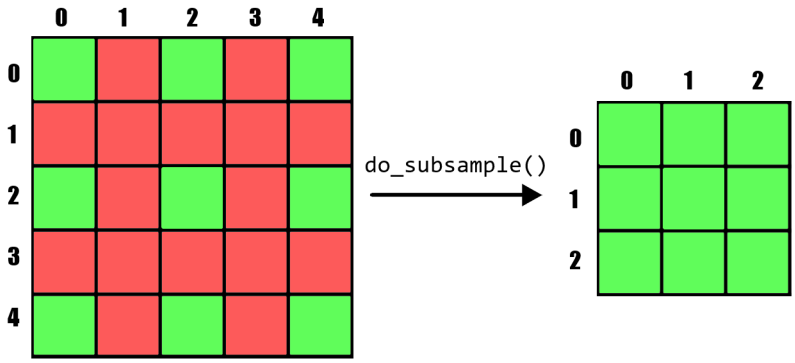

Figura 6: Subsample de entrada (5,5)

La implementación de este procedimiento se muestra en el **Listing 5**. Dado que sólo se eliminan los pixeles con índices impares, es fácil darse cuenta que el tamaño de salida de una imagen de dimensiones (N,N) está dado por:

$$(\left\lceil \frac{N}{2} \right\rceil, \left\lceil \frac{N}{2} \right\rceil) \tag{3}$$

La única dificultad que presenta esta función es cómo recorrer, con sólo dos *for loops*, la imagen de entrada y la de salida. Para esto, se utilizan los *for loops* para recorrer la imagen de entrada, y para recorrer la imagen de salida se utilizan las variables *xIndex* e *yIndex*. Ambas parten en 0, *yIndex* aumenta cuando se copia un pixel, y xIndex aumenta cuando se termina una fila, momento en el cual *yIndex* debe volver a 0. Sólo se recorre una fila si el índice del primer *for* es par, y sólo se copian pixeles si el índice del segundo *for* también es par.

3.5. Calcular la priámide de Gauss

El procedimiento para calcular las pirámides de Gauss se explicó en la **Sección 2.2**, y en las **Secciones 3.3 y 3.4** se explicó cómo suavizar y submuestrear imágenes. Por lo que en esta sección basta con poner todo en conjunto.

La implementación de este procedimiento se muestra en el **Listing 6**. Se almacena la imagen original en una lista *gausspyr* y se recorre un *for loop* del tamaño de la cantidad de niveles que se deseen. Dentro del *loop* se aplica el suavizado a la imagen del piso anterior, se submuestrea y se guarda el resultado en *gausspyr*.

3.6. Mostrar pirámides de Gauss

Se utiliza la función *subplots* de la librería *matplotlib*, para mostrar todas las imágenes de una pirámide de Gauss en una misma fila. Los resultados de esta función, mostrada en el **Listing 7** se muestran en la **Sección 6.1**.

4. Cálculo de las pirámides de Laplace

4.1. Restar dos imágenes

La resta de imágenes se hace pixel a pixel, por lo que se debe imponer que el tamaño de las imágenes sea el mismo. La implementación de esta función se muestra en el **Listing 8**, donde lo único que cabe destacar es que los pixeles de *input1* hacen de minuendo y los de *input2* hacen de sustraendo.

4.2. Calcular la priámide de Laplace

El procedimiento para calcular las pirámides de Laplace se explicó en la **Sección 2.3**. El suavizado de imágenes se explicó en la **Sección 3.3** y la resta en la **Sección 4.1**, por lo que en esta sección basta con juntar ambas funciones.

La implementación de este procedimiento se muestra en el **Listing 9**. El procedimiento es análogo al cálculo de la pirámide de Gauss explicado en la **Sección 3.5**, sólo difiriendo en que aquí se almacena la resta entre la imagen actual y su versión suavizada.

4.3. Obtener valor absoluto y escalar

Se requiere una función que aplique la función valor absoluto y luego aplique un factor de escalamiento a todos los pixeles de una imagen. Para esto basta con recorrer la imagen pixel a pixel, aplicar la función *absolute* de la librería *numpy* y luego multiplicar por el factor de escalamiento.

La implementación de este procedimiento se muestra en el **Listing 10**.

4.4. Mostrar pirámides de Laplace

Nuevamente, se utiliza la función *subplots* de la librería *matplotlib*, para mostrar todas las imágenes de la pirámide de Laplace en una misma fila. Cabe destacar que antes de mostrar las pirámides, sus imágenes deben ser escaladas con la función *abs_then_scale*, mencionada en la **Sección 4.3**, donde de manera arbitraria se escoge 4 como factor de escalamiento.

Los resultados de esta función, mostrada en el **Listing 11** se muestran en la **Sección 6.2**.

5. Reconstrucción de imágenes

5.1. Sumar dos imágenes

Al igual que la resta de imágenes explicada en la **Sección 4.1**, la suma de imágenes se hace pixel a pixel. Sin embargo, en este caso no se puede imponer que las imágenes que se sumen sean del mismo tamaño, pues se sabe que en algunos casos las imágenes necesariamente van a tener diferencias de tamaño.

Esto pues, al submuestrear una imagen de dimensiones impares, (25,26) por ejemplo, la imagen resultante, según la **Ecuación 3**, va a ser de dimensiones (13,13). Luego al duplicar el tamaño de dicha imagen, como se explica más adelante en la **Sección 5.2**, va a quedar una imagen de dimensiones (26,26).

Más adelante se explica que es necesario sumar imágenes como en el ejemplo mencionado, la original con dimensiones impares (25,26) con la misma imagen que pasó por un proceso de submuestreo y luego de agrandamiento y que resulta tener dimensiones pares (26,26).

Es por esto que en la implementación mostrada en el **Listing 12**, se impone que el tamaño de la imagen de salida va a ser el mismo de la imagen de entrada más pequeña. De este modo, si las filas o las columnas no coinciden, simplemente se va a sumar las que estén dentro del rango de la más pequeña.

5.2. Duplicar el tamaño de una imagen

Este procedimiento es un poco parecido al de submuestrear una imagen, explicado en la **Sección 3.4**, pero de manera inversa. En este caso los pixeles de la imagen original se copian en las posiciones correspondientes al doble de los índices de los pixeles originales, como se muestra en la **Figura 7**:

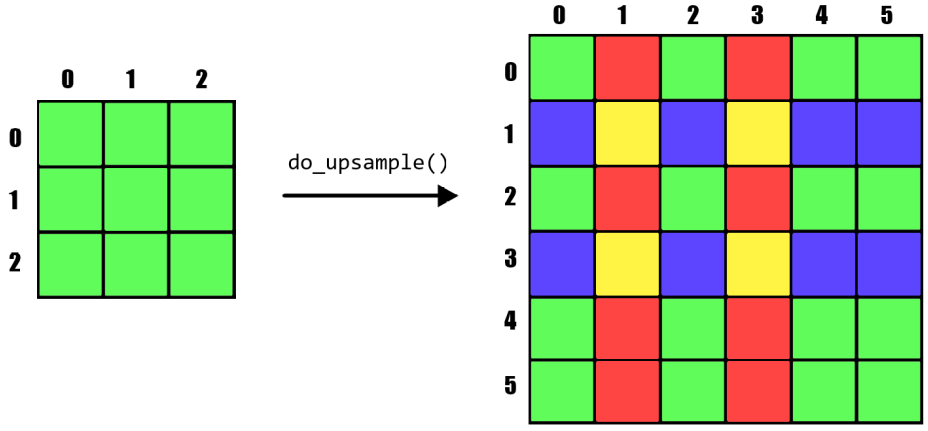

Figura 7: Duplicar tamaño imagen

Pero luego de hacer eso queda un montón de pixeles vacíos, que hay que decidir cómo se rellenan.

Se presentan tres casos, ilustrados en la Figura 7, que dependen de los valores de los índices de los pixeles, I (fila) y J (columna):

1. **I par, J impar**: Representado en la Figura 7 como los pixeles rojos, estos se rellenan promediando los valores de los pixeles que tienen al lado en la misma **fila**. Por ejemplo, el pixel (0,1) es el promedio de los pixeles (0,0) y (0,2).

2. **I impar, J par**: Representado en la Figura 7 como los pixeles azules, estos se rellenan promediando los valores de los pixeles que tienen al lado en la misma **columna**. Por ejemplo, el pixel (1,0) esel promedio de los pixeles (0,0) y (2,0).

3. **I impar, J impar**: Representado en la Figura 7 como los pixeles amarillos, se rellenan promediando los valores de los 4 pixeles originales que tienen alrededor. Por ejemplo, el pixel (1,1) es el promedio de los pixeles (0,0), (0,2), (2,0) y (2,2).

Dado que en la última fila y columna no hay más pixeles con los que promediar, simplemente se copian los pixeles de la fila y columna anterior.

La implementación de este procedimiento se muestra en el **Listing 13**. Se recorre la salida con 3 *for loops*, uno para las filas, separando en pares e impares, y en el interior de ambos casos uno para las columnas, donde también se separa en caso par e impar.

Dado que se recorre la salida, hay que determinar cómo se encuentran los pixeles de la imagen de entrada que se utilizan en los promedios descritos anteriormente. Para esto es conveniente tomar como ejemplo pixeles puntuales de la Figura 7, denotando la imagen de entrada como *img* y la de salida como *out*.

Tomando como ejemplo para el caso 1 el pixel de la imagen de salida (0,3), se puede ver que:

$$out[0,3] = \frac{img[0,1] + img[0,2]}{2} = \frac{img[\frac{0}{2}, \lfloor \frac{3}{2} \rfloor] + img[\frac{0}{2}, \lceil \frac{3}{2} \rceil]}{2}$$

Por lo que es claro que la regla para el caso (I par, J impar) es:

$$out[I,J] = \frac{img[\frac{I}{2}, \lfloor \frac{J}{2} \rfloor] + img[\frac{I}{2}, \lceil \frac{J}{2} \rceil]}{2} \tag{1}$$

Tomando como ejemplo para el caso 2 el pixel de la imagen de salida (1,2), se puede ver que:

$$out[1,2] = \frac{img[0,1] + img[1,1]}{2} = \frac{img[\lfloor \frac{1}{2} \rfloor, \frac{2}{2}] + img[\lceil \frac{1}{2} \rceil, \frac{2}{2}]}{2}$$

Por lo que es claro que la regla para el caso (I impar, J par) es:

$$out[I,J] = \frac{img[\lfloor \frac{I}{2} \rfloor, \frac{J}{2}] + img[\lceil \frac{I}{2} \rceil, \frac{J}{2}]}{2} \tag{2}$$

Por último, tomando como ejemplo para el caso 3 el pixel de la imagen de salida (3,3), se puede ver que:

$$\begin{aligned} out[3,3] &= \frac{img[1,1] + img[1,2] + img[2,1] + img[2,2]}{4} \\ &= \frac{img[\lfloor \frac{3}{2} \rfloor, \lfloor \frac{3}{2} \rfloor] + img[\lfloor \frac{3}{2} \rfloor, \lceil \frac{3}{2} \rceil] + img[\lceil \frac{3}{2} \rceil, \lfloor \frac{3}{2} \rfloor] + img[\lceil \frac{3}{2} \rceil, \lceil \frac{3}{2} \rceil]}{4} \end{aligned}$$

Por lo que es claro que la regla para el caso (I impar, J impar) es:

$$out[I,J] = \frac{img[\lfloor \frac{I}{2} \rfloor, \lfloor \frac{J}{2} \rfloor] + img[\lfloor \frac{I}{2} \rfloor, \lceil \frac{J}{2} \rceil] + img[\lceil \frac{I}{2} \rceil, \lfloor \frac{J}{2} \rfloor] + img[\lceil \frac{I}{2} \rceil, \lceil \frac{J}{2} \rceil]}{4} \tag{3}$$

En resumen, en el **Listing 13** se copian los pixeles originales en las posiciones correspondientes al doble de sus índices, se rellenan los espacios vacíos usando las **Ecuaciones 1, 2 y 3**, y la última fila y columna se rellenan con una copia de las penúltimas.

5.3. Reconstruir imágenes

El procedimiento para reconstruir las imágenes a partir de las pirámides de Laplace se explicó en la **Sección 2.4**, y en las **Secciones 5.1 y 5.2** se explicó cómo sumar imágenes y cómo duplicar su tamaño. Por lo que para esta sección basta con juntar ambas funciones.

La implementación de este procedimiento se muestra en el **Listing 14**, en cada piso se duplica el tamaño de la imagen y se suma con la del piso siguiente, usando las funciones *do_upsample* y *add*.

6. Resultados

6.1. Pirámides de Gauss

Se calculan las pirámides de Gauss para las 4 imágenes de entrada, el resultado se presenta en la **Figura** 8. En cada fila se presenta una pirámide distinta, comenzando en el piso 0 a la izquierda y terminando en el piso 4 a la derecha:

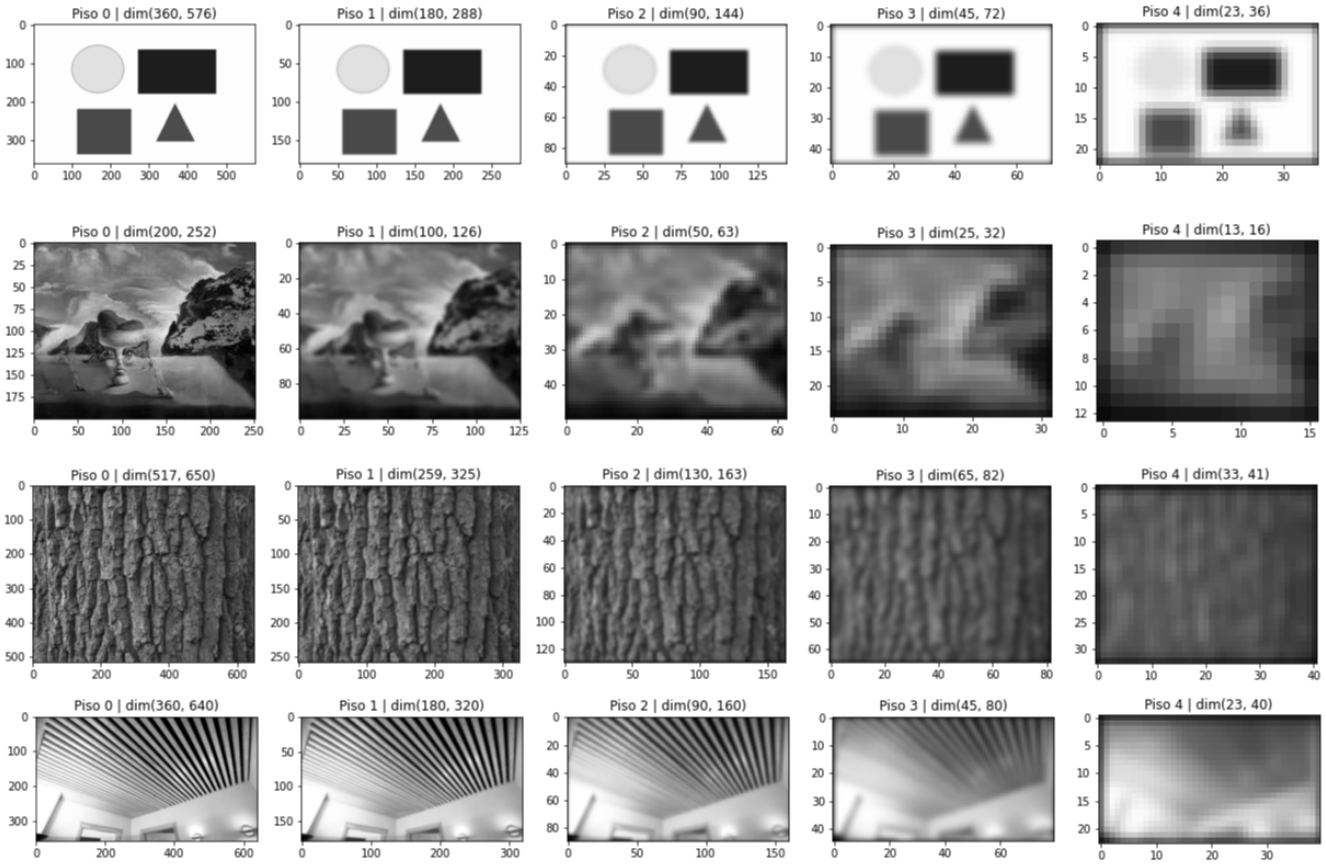

Figura 8: Pirámides de Gauss para las 4 imágenes de entrada

Cabe destacar que el tamaño de las figuras está forzado a ser el mismo, pero las figuras efectivamente se vuelven más pequeñas a medida que se avanza en los pisos de la pirámide, evidenciado por el cambio de escala en los ejes.

6.2. Pirámides de Laplace

Se calculan las pirámides de Laplace para las 4 imágenes de entrada, el resultado se presenta en la **Figura 9**. En cada fila se presenta una pirámide distinta, comenzando en el piso 0 a la izquierda y terminando en el piso 4 a la derecha:

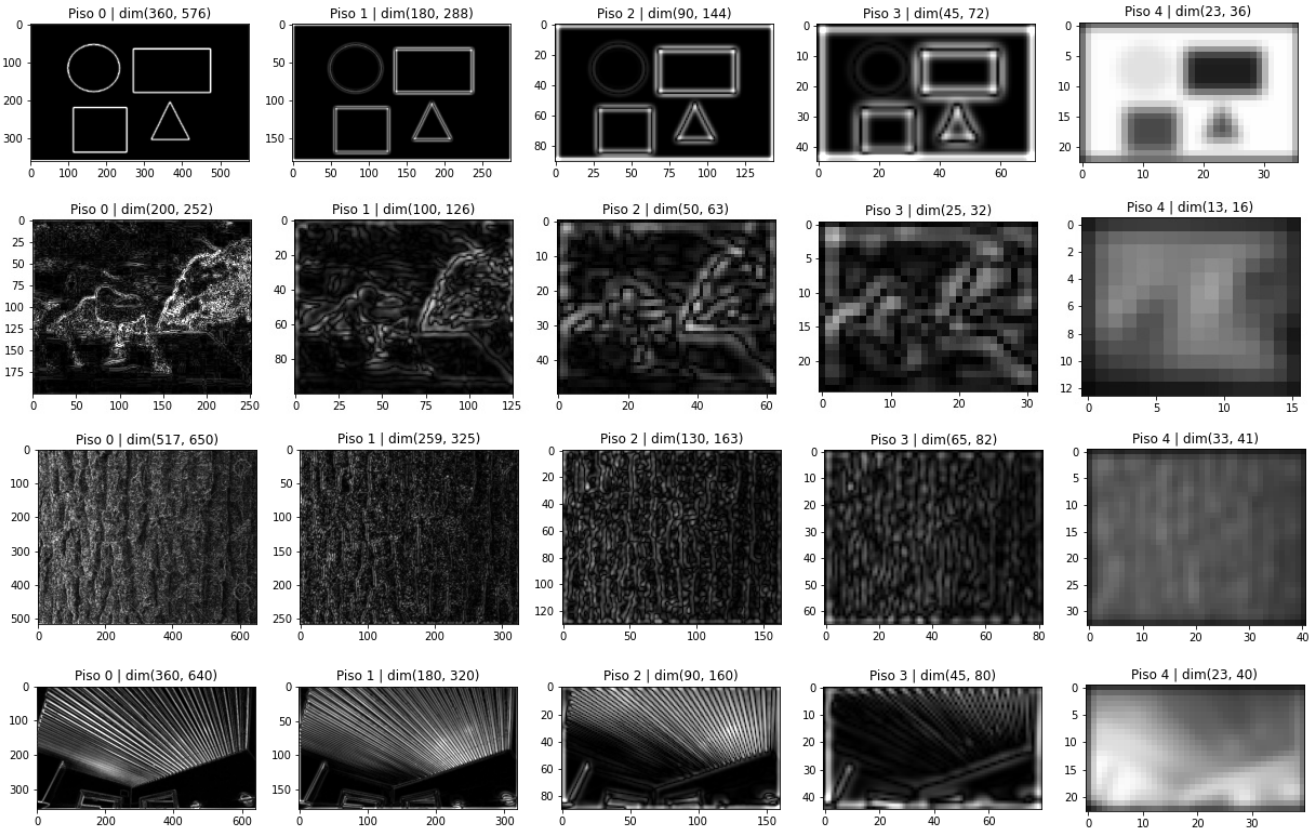

Figura 9: Pirámides de Laplace para las 4 imágenes de entrada

Al igual que en la **Sección 6.1**, las imágenes de las pirámides efectivamente se vuelven más pequeñas, pero la figura está forzada a mantener el tamaño, para lograr apreciar los últimos pisos.

6.3. Reconstrucción

Se calculan las pirámides de Gauss y Laplace para las 4 imágenes de entrada y luego se reconstruyen, el resultado se presenta en la **Figura 10**. En la primera fila se presentan las imágenes originales y en la segunda fila, en sus respectivas columnas, se presenta su contraparte reconstruida:

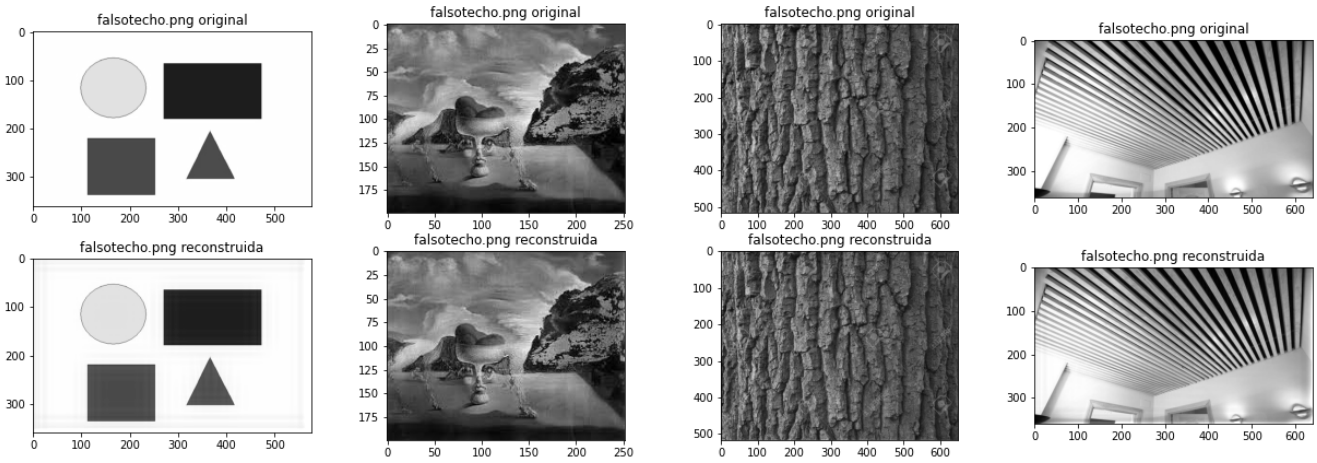

Figura 10: Reconstrucción de las 4 imágenes de entrada

7. Análisis de resultados

Se logra calcular de manera satisfactoria las pirámides de Gauss y de Laplace. En la **Figura 8** se puede apreciar que, tal como se pretendía, las imágenes de la pirámide se van volviendo más borrosas y más pequeñas. Por lo que se comprueba la efectividad de los kernel Gaussianos como filtro pasa bajo.

Asimismo, en la **Figura 9** se puede apreciar que en las imágenes de las pirámides se capturan los bordes de las imágenes de las pirámides de Gauss. Dado que las imágenes de la pirámide de Gauss se obtienen mediante un filtro pasa bajo, es decir, en las imágenes quedan sólo los componentes de baja frecuencia. Y las de Laplace se obtienen mediante restarle dichas imágenes de baja frecuencia a las originales, las imágenes de Laplace sólo almacenan los componentes de alta frecuencia de las imágenes, por lo que actúa como un filtro pasa bajo.

Por último, se puede apreciar en la **Figura 10** que las imágenes reconstruidas son prácticamente iguales a las originales, notándose más las diferencias en zonas con menos detalles, como en *falso-techo.png*. Si bien el efecto de suavizar, subsamplear y convolucionar con zero padding hace que se vaya perdiendo información a medida que se sube en las pirámides, los pisos anteriores contienen la información suficiente como para reconstruir la imágen original de manera satisfactoria.

8. Conclusión

Se logra implementar en Python todas las funciones necesarias tanto para las representaciones multi-escala de Gauss y Laplace, como para la reconstrucción de imágenes. Debido a que la programación realizada es de bajo nivel, es decir, utilizando casi exclusivamente *for loops* y operaciones básicas, se logra adquirir un mayor entendimiento sobre el funcionamiento de estos algoritmos.

También se vuelve evidente que estas representaciones multi-escala son una buena forma de acceder a los componentes de baja y alta frecuencia de una imagen, sin necesidad de recurrir a las transformadas de fourier.

9. Anexo

9.1. Cálculo de pirámides de Gauss

```
%%cython
# import cython
import numpy as np
import math
cimport numpy as np
# La convolucion debe ser implementada usando cython (solo esta funcion en cython)
#@cython.boundscheck(False)
cpdef np.ndarray[np.float32_t, ndim=2] convolution_cython(np.ndarray[np.float32_t,
    ndim=2] input, np.ndarray[np.float32_t, ndim=2] mask):
  cdef int y, x, rows, cols, kernelRows, kernelCols, offset
  cdef float sum, pixel
  # cdef float sum
  cdef np.ndarray[np.float32_t, ndim=2] output = np.zeros([input.shape[0],input.
    shape[1]], dtype = np.float32)

  # tamano de la imagen
  rows = input.shape[0]
  cols = input  .shape[1]

  # tamanno del kernel
  kernelRows = mask.shape[0]
  kernelCols = mask.shape[1]

  # como encontrar los pixeles correspondientes
  offsetX = math.floor(kernelRows/2)
  offsetY = math.floor(kernelCols/2)

  # convolucion
  for y in range(rows):
    for x in range(cols):
      sum = 0
      for kernelY in range(kernelRows):
        for kernelX in range(kernelCols):
          posX = (x - offsetX) + kernelX
          posY = (y - offsetY) + kernelY
          if ((min(posX,posY) < 0) | (posX >= cols) | (posY >= rows)):
            pixel = 0.0
          else:
            pixel = input[posY,posX]
          sum += pixel*mask[kernelY,kernelX]
      output[y,x] = sum
  return output
```

Listing 1: convolution_cython

```
# funcion gaussiana en 2d
def gauss2d(x,y,sigma):
  e = -(x**(2) + y**(2))/(2*sigma**2)
  return np.exp(e)/(2*np.pi*sigma**2)
```

Listing 2: gauss2d

```
# generador del kernel
def compute_gauss_mask_2d(sigma, width):
  gmask = np.zeros((width, width), np.float32)
  # Por hacer: implementar calculo de mascara gaussiana 2d pixel a pixel
  # Se debe normalizar tras calcularla para que las sumas de los pixeles sea igual
   a 1
  offset = math.floor(width/2)
  sum = 0 # normalizar
  for i in range(width):
    for j in range(width):
      value = gauss2d(i-offset,j-offset,sigma)
      gmask[i,j] = value
      sum += value
  return gmask/sum
```

Listing 3: compute_gauss_mask_2d

```
def apply_blur(input, sigma, width):
  # Por hacer:
  # 1) Calcular mascara gaussiana 2d con parametros sigma y width
  mask = compute_gauss_mask_2d(sigma,width)
  # 2) Calcular convolucion entre la imagen de entrada "input" y la mascara 2d
  output = convolution_cython(input, mask)
  return output
```

Listing 4: apply_blur

```
def do_subsample(img):
  # por hacer: implementar submuestreo pixel a pixel
  output = np.zeros((math.ceil(img.shape[0]/2),math.ceil(img.shape[1]/2)), np.
   float32)
  xIndex = yIndex = 0
  for i in range(img.shape[0]):
    if (i%2 == 0):
      for j in range(img.shape[1]):
        if (j%2 == 0):
          output[xIndex,yIndex] = img[i,j]
          yIndex += 1
      xIndex += 1
      yIndex = 0
  return output
```

Listing 5: do_subsample

```python
def calc_gauss_pyramid(input, levels):
  gausspyr = []
  current = np.copy(input)
  gausspyr.append(current)
  for i in range(1,levels):
    # Por hacer:
    # 1) Aplicar apply_blur() a la imagen gausspyr[i-1], con sigma 2.0 y ancho 7
    blur = apply_blur(gausspyr[i-1], 2.0, 7)
    # 2) Submuestrear la imagen resultante usando do_subsample y guardando el
   resultado en current
    current = do_subsample(blur)
    gausspyr.append(current)
  return gausspyr
```

Listing 6: calc_gauss_pyramid

```python
def show_gauss_pyramid(pyramid):
  fig, axs = plt.subplots(1, len(pyramid))
  fig.set_figheight(14)
  fig.set_figwidth(21)
  for i in range(len(pyramid)):
    axs[i].imshow(pyramid[i], 'gray', vmin = 0, vmax = 255)
    axs[i].set_title(f"Piso {i} | dim{pyramid[i].shape}")
```

Listing 7: show_gauss_pyramid

9.2. Cálculo de pirámides de Laplace

```python
def subtract(input1, input2):
  assert(input1.shape == input2.shape), "Las imagenes deben tener las mismas dimensiones"
  out = np.zeros(input1.shape)
  for i in range(input1.shape[0]):
    for j in range(input1.shape[1]):
      out[i,j] = input1[i,j] - input2[i,j]
  return out
```

Listing 8: subtract

```python
def calc_laplace_pyramid(input, levels):
  gausspyr = []
  laplacepyr = []
  current = np.copy(input)
  gausspyr.append(current)
  for i in range(1, levels):
    # Por hacer:
    # 1) Aplicar apply_blur( ) a la imagen gausspyr[i-1], con sigma 2.0 y ancho 7
    blur = apply_blur(gausspyr[i-1], 2.0, 7)
    # 2) Guardar en laplacepyr el resultado de restar gausspyr[i-1] y la imagen calculada en (1)
    laplacepyr.append(subtract(gausspyr[i-1], blur))
    # 3) Submuestrear la imagen calculada en (1), guardar el resultado en current
    current = do_subsample(blur)
    gausspyr.append(current)
  laplacepyr.append(current)  # Se agrega el ultimo piso de la piramide de Laplace
  return laplacepyr
```

Listing 9: calc_laplace_pyramid

```python
def abs_then_scale(img, factor):
  # Por hacer: aplicar valor absoluto a los pixeles de la imagen pixel a pixel y luego escalar los pixeles usando el factor indicado
  output = np.zeros(img.shape,np.float32)
  for i in range(img.shape[0]):
    for j in range(img.shape[1]):
      output[i,j] = np.absolute(img[i,j])*factor
  return output
```

Listing 10: abs_then_scale

```python
def show_laplace_pyramid(pyramid):
  # Por hacer: mostrar las imagenes de la piramide de laplace:
  #  Las imagenes deben ser escaladas antes de mostrarse usando abs_then_scale( )
  #  Sin embargo, la ultima imagen del ultimo piso se muestra tal cual
  scaled = []
  for i in range(len(pyramid) - 1):
    scaled.append(abs_then_scale(pyramid[i],4))
  scaled.append(pyramid[-1])
  # Se recomienda usar cv2_imshow( ) para mostrar las imagenes
  fig, axs = plt.subplots(1, len(pyramid))
  fig.set_figheight(14)
  fig.set_figwidth(21)
  for i in range(len(scaled)):
    axs[i].imshow(scaled[i], 'gray', vmin = 0, vmax = 255)
    axs[i].set_title(f"Piso {i} | dim{scaled[i].shape}")
```

Listing 11: show_laplace_pyramid

9.3. Reconstrucción de imágenes

```
def add(input1, input2):
  # Por hacer: calcular la resta entre input1 e input2, pixel a pixel
  # la imagen resultante es del tamanno de la mas pequenna
  output = np.zeros(min(input1.shape, input2.shape))
  for i in range(output.shape[0]):
    for j in range(output.shape[1]):
      output[i,j] = input1[i,j] + input2[i,j]
  return output
```

Listing 12: add

```
def do_upsample(img):
  output = np.zeros((img.shape[0]*2,img.shape[1]*2), np.float32)
  for i in range(output.shape[0] - 1):
    if i%2 == 0:
      for j in range(output.shape[1] - 1):
        if j % 2 == 0:
          output[i,j] = img[int(i/2),int(j/2)]
        else:
          output[i,j] = (img[int(i/2),math.floor(j/2)] + img[int(i/2),int(math.ceil
    (j/2))])/2
      output[i,-1] = output[i,-2]
    else:
      for j in range(output.shape[1] - 1):
        if j % 2 == 0:
          output[i,j] = (img[int(math.floor(i/2)),int(j/2)] + img[int(math.ceil(i
    /2)),int(j/2)])/2
        else:
          output[i,j] = (img[int(math.floor(i/2)),int(math.floor(j/2))] + img[int(
    math.floor(i/2)),int(math.ceil(j/2))] + img[int(math.ceil(i/2)),int(math.floor(j
    /2))] + img[int(math.ceil(i/2)),int(math.ceil(j/2))])/4
      output[i,-1] = output[i,-2]
  output[i,-1] = output[i,-2]
  output[-1] = output[-2][:]
  return output
```

Listing 13: do_upsample

```
def do_reconstruct(laplacepyr):
  output = np.copy(laplacepyr[len(laplacepyr)-1])
  for i in range(1, len(laplacepyr)):
    level = int(len(laplacepyr)) - i - 1
    # Por hacer: repetir estos dos pasos:
    # (1) Duplicar tamano output usando do_upsample( )
    output = do_upsample(output)
    # (2) Sumar resultado de (1) y laplacepyr[level] usando add( ), almacenar en
    output
    output = add(output, laplacepyr[level])
  return output
```

Listing 14: do_reconstruct

Referencias

[1] Tarea 1 EL7008 (Primavera 2021): Pirámides de Gauss y Laplace, Javier Ruiz del Solar, Patricio Loncomilla.

[2] Convolutional Neural Networks, Vikas Solegaonkar

[3] Función Gaussiana, Wikipedia.