Departamento de Ingeniería Eléctrica
Facultad de Ciencias Físicas y Matemáticas
Universidad de Chile
EL7008-1 Procesamiento Avanzado de Imágenes

Tarea 1

Piramides de *Gauss* y *Laplace*

Diego Irarrázaval I.

Profesor:
Javier Ruiz del Solar.

Auxiliar:
Patricio Loncomilla Z.

Fecha:
28 de septiembre de 2020

Índice

Índice de figuras

Índice de tablas

Índice de Códigos

1. Introducción

En esta tarea, se implementará el cálculo de pirámides de *Gauss* y *Laplace* de una imagen y, luego, se reconstruirán a partir de dichas pirámides. Para lograr esto, se deberá implementar también la convolución en dos dimensiones.

Los principales objetivos corresponden en primer lugar a introducir algunas formas de representaciones multi-resolución calculadas a partir de una images e implementar operaciones desde cero (por ejemplo la convolución) que usualmente se cargan con librerías.

El informe comienza con el marco teórico donde se expone brevemente sobre la convolución, la pirámide de Gauss, la pirámide de Laplace y la reconstrucción de la imagen.

A continuación, la sección desarollo se divide en tres sub-secciones:

1. Pirámide de Gauss.

2. Pirámide de Laplace.

3. Reconstrucción imagen.

En las secciones enumeradas anteriormente, se incluye tanto el código implementado como análisis teórico de lo desarrollado.

Finalmente, se presentan las conclusiones del desarrollo de la tarea 1.

2. Marco Teórico

2.1. Cnvolución

En matemáticas, la convolución es una operación que recibe dos funciones (f y g) y entrega una tercera función ($f * g$) que describe como la forma de una es influida por la otra [1].

En procesamiento de señales, se puede entender como afecta se ve afectada señal pasar por un filtro. En este caso, la señal corresponde a una imagen y el filtro será otra imagen (le llamaremos indistintamente *kernel*, *mask* o máscara). A continuación, se muestra una ilustración que se utilizará para explicar la operación convolución en dos dimensiones:

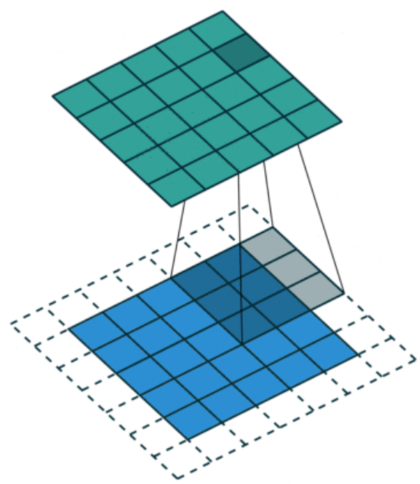


Figura 1: Convolucion en dos dimensiones con padding.[2]

En la figura 1 se observa una imagen de entrada (representada por la matriz de cuadrados azules) con 5 pixeles de ancho por 5 de largo. Adicionalmente, se agrega un pixel de ancho y uno de largo de color blanco. Éstos representan el *padding*.El kernel por otro lado, se representa por los pixeles sombreados (kernel de 3x3).

Para el cálculo de la imagen de salida, se utiliza la siguiente ecuación:

$$y[i,j] = \sum_{m}^{M} \sum_{n}^{N} x[i-m, j-n] \cdot h[m,n] \quad \forall i \in [0, Height],\ j \in [0, Width] \tag{1}$$

En la ecuación 1, la imagen de salida (y), se construye con la operación del kernel (h) sobre todos los pixeles de la imagen de entrada (x) y los vecinos correspondientes.

En el ejemplo de la figura, se realiza *padding*. Esto consiste en agrandar la imagen de entrada para que la salida sea de la misma dimension. En esta ocación, se realizará padding con ceros, es decir, se rellenará con ceros donde sea necesario.

2.2. Pirámides

Como se menciono anteriormente, las pirámides corresponden a representaciones multi-resolución calculadas a partir de imágenes. En éstas, la señal o imagen de entrada es filtrada y submuestreada tantas veces como niveles se quieran.

Inicialmente, las pirámides se utilizaron para la detección e identificación de objetos que pueden o no aparecer a distintas escalas. La idea, es crear distintas copias del objeto cambiando el tamaño (o escala) y resolución de estos. En la figura 2 se observan distintas resoluciones del mismo objeto para obtener otras representaciones.

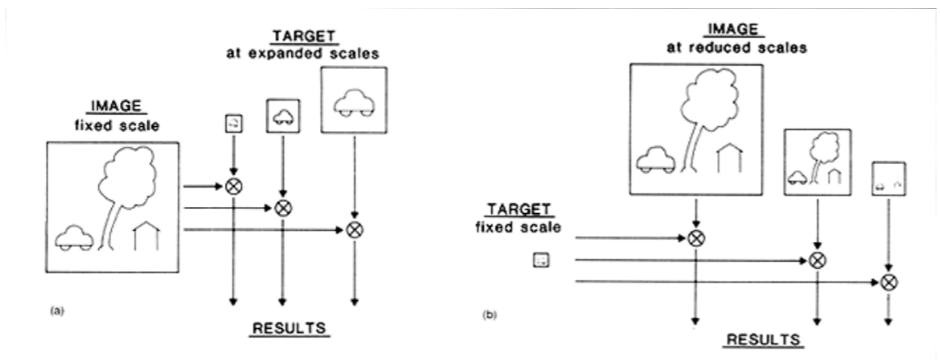


Figura 2: Distintas formas de utilizar las pirámides. En (a), copias del objeto a detectar se escalan. En (b), la imagen se trabaja completa. [3]

2.2.1. Piramide de Gauss

La pirámide de gauss corresponde a un caso particular de las pirámides en las que el filtro aplicado es uno pasabajo, con semejanzas a un filtro con distribución *gaussiana*.

En estos, se aplica un filtro *gaussiano* a cada imagen y luego se sub-muestrea. El filtro *gaussiano* también se conoce como *blur* y recibe como parámetro la desviación estándar a partir de la cual se originará el filtro. En la siguiente figura, se observa el efecto de variar la desviación estándar usada para el cálculo del filtro:

(a) Imagen original (en blanco y negro). (b) Imagen con filtro *blur* aplicado con $\sigma = 5$. (c) Imagen con filtro *blur* aplicado con $\sigma = 10$.

Figura 3: Misma imagen con distintos filtros *blur* aplicados (distinto σ mismo tamaño de kernel, 10).

En la figura se observa el efecto de aumentar σ. Por otro lado, en los bordes de las figuras 3 (a) y (b), se observan los bordes negros que corresponden al *padding* mencionado anteriormente.

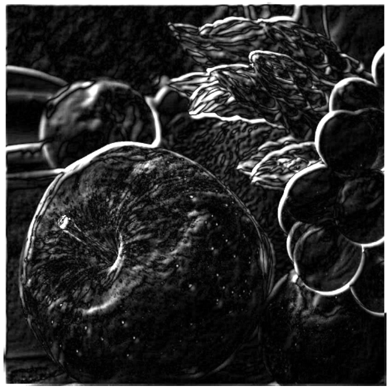
2.2.2. Piramide de Laplace

Estas pirámides se construyen con la información perdida en la pirámide de Gauss. Para esto, se resta el nivel actual de la pirámide de Gauss con el siguiente (antes del sub-muestreo). El último nivel de la pirámide de Laplace corresponde al último nivel de la pirámide de gauss.

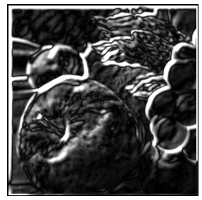
En las figuras 4 y 5, se muestran 3 niveles de la piramide de gauss y laplace respectivamente. Éstas fueron generadas a partir de la implementación de la tarea:

(a) Nivel 1. (b) Nivel 2. (c) Nivel 3.

Figura 4: Pirámide de Gauss de 3 niveles.



(a) Nivel 1.



(b) Nivel 2.

(c) Nivel 3.

Figura 5: Pirámide de Laplace de 3 niveles.

2.3. Reconstrucción de la imagen original

El proceso de reconstrucción de la imagen a partir de la pirámide de Laplace consiste en, a partir del nivel más profundo de la pirámide repetir lo siguiente:

1. Duplicar el tamaño de la imagen.

2. Sumar la imagen duplicada con el siguiente piso de la pirámide de Laplace.

Para duplicar el tamaño de la imagen, se utiliza interpolación. Es importante destacar que, la imagen reconstruida contiene ciertos artefactos en particular en los bordes. Esto se debe a que al hacer padding (basicamente rellenar con ceros), se pierde información de la imagen original:

Figura 6: Reconstrucción de la imagen de frutas a partir de la pirámide de Laplace de 3 niveles.

En la siguiente sección, se encuentra la implementación de todo lo descrito. Por cada ítem, se muestra el código, explicación breve donde sea necesario y una prueba de la función implementada o el resultado que arroja esta.

3. Desarrollo

3.1. Pirámide de *Gauss*

3.1.1. Convolución

A continuación, se presenta el código de la implementación de la convolución en *Cython*:

```
cpdef float[:, :] convolution_cython(float [:, :] input, float [:, :] mask):
cdef int a, b, r, c, rows, cols, row_init, col_init, i, j,
cdef float sum
# Imagen de salida
cdef np.ndarray output=np.zeros([input.shape[0], input.shape[1]], dtype = np.float32)

# Posicion a partir de la cual se puede realizar convolucion:
# Ejemplo 1: Para un kernel de 3x3, es (1,1).
# Ejemplo 2: Para un kernel de "a" x "b" es ("r"//2, "c"//2)
a = mask.shape[0]
b = mask.shape[1]

row_init = a // 2
col_init = b // 2

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

sum = 0

# Recorremos la imagen input:
for r in range(row_init, rows - row_init):
  for c in range(col_init, cols - col_init):
    # Se recorre la mascara o kernel:
    for i in range(a):
      for j in range(b):
        sum += mask[i,j] * input[r-i,c-j]
    # Guardamos el resultado de la suma correspondiente en el arreglo output:
    output[r, c] = sum
    sum = 0
return output
```

Código 1: Implementación de convolución en Cython.

La implementación en 1, corresponde a la convolución en dos dimensiones con padding. En el código, la sección más importante corresponde a los 4 ciclos de iteraciones anidados. Estos son los que recorren en primer lugar la imagen (`for r in rows` y luego `for c in columns`) y luego el *kernel* guardando en la variable `sum` el resultado de operar para cada pixel `(r,c)` de la imagen, el resultado de la operación convolución.

Para comparar la eficiencia de la función `convolution_cython`, se comparó contra una implementada en el módulo *scipy*. Para esto, se utilizó el comando `%timeit`. Éste ejecuta una linea cierta cantidad de veces (100 en este caso) y guarda el mejor tiempo obtenido. Adicionalmente, utiliza distintos *cores* para verificar que sea el mejor resultado:

	Cython	***Scipy***
Timpo [ms]	15.8	3.08

Tabla 1: Tiempos de ejecución de operación convolución.

Se observa en la tabla 1 que la función del módulo *scipy* es varias veces mas rápida (puede variar cuánto más rapida es). De todas formas, se continuará utilizando la función implementada en *cython*, siguiendo las instrucciones del enunciado.

A continuación se muestra un ejemplo de la ejecución de la función `convolution_cython` con un *kernel* de la siguiente forma (es un *kernel* util para la detección de bordes):

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{2}$$

(a) Imagen original.

(b) Imagen obtenida.

Figura 7: Salida obtenida al usar la función *convolution_ cython*.

3.1.2. Cálculo de máscaras

```
def compute_gauss_horiz(sigma, width):
  mask = np.zeros([1, width], np.float32)


  coef = 1 / (sigma * np.sqrt(2 * np.pi))

  for i in range(width):
    mask[0][i] = coef * np.exp(- np.square(i - width//2) / (2 * sigma*sigma)
    )
    # Para debuggear
    #print((i - width//2))
```

```

  # normalizamos
  return mask / mask.sum()

def compute_gauss_vert(sigma, height):
  mask = np.zeros((height, 1), np.float32)
  # no es necesario implementar nuevamente la exponencial, solo trasponer el
    resultado
  # de la funcion comute_gauss_horiz
  hor_mask = compute_gauss_horiz(sigma = sigma, width = height)
  for i in range(height):
    mask[i][0] = hor_mask[0][i]
  return mask
```

Código 2: Cálculo de las máscaras en Python.

Para el cálculo de los coeficientes del *kernel*, se utilizó la siguiente ecuación:

$$G(x, \sigma) \;=\; \frac{1}{\sigma\sqrt{2\,\pi}}\exp(\tfrac{-x^2}{2\sigma^2}) \tag{3}$$

Para que el cálculo sea correcto, se debe hacer un cambio de variable. Si el kernel es de una dimensión y de largo n, es de la forma:

$$g(x) \;=\; [g_0,\; g_1,\; g_x,\; ...,\; g_n], \qquad \forall x \;\in\; [0, n] \tag{4}$$

Pero la ecuación 3, toma cada valor de x con el centro del *kernel* como referencia. Para resolver esto, en el código se hace el siguiente cambio de variables:

$$x \;=\; i \;-\; n//2 \tag{5}$$

De esta forma, al recorrer desde $[0, n]$, se obtiene el valor correcto. El último paso para obtener el kernel es normalizar. A continuación, se muestra el resultado obtenido del cálculo de una máscara de una dimensión con $\sigma \;=\; 1$ y largo 3:

$$mask_{horizontal} \;=\; [[\,0.27406862,\; 0.45186278,\; 0.27406862\,]]$$

Para el cálculo de la máscara vertical, se obtiene una máscara horizontal y se transpone. El resultado entregado por la función *compute_gauss_vert* con los mismos parámetros anteriores son:

$$mask_{vertical} \;=\; [[0.27406862],\; [0.45186278],\; [0.27406862]]$$

3.1.3. Suavizado de imágenes

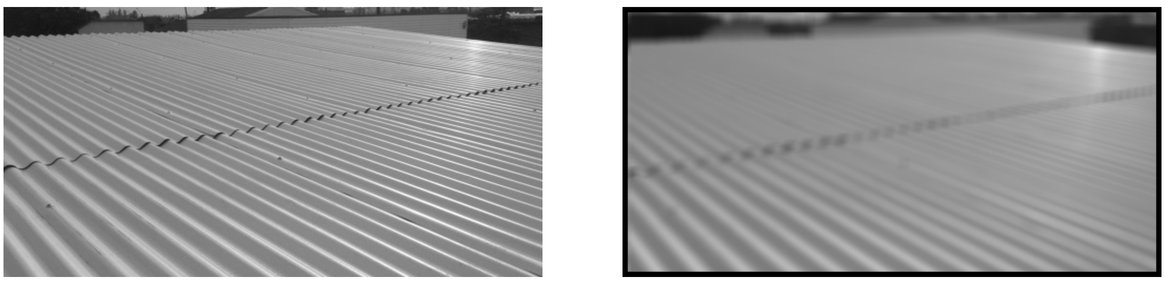
```
def do_blur(input, sigma, height):
  # Obtenemos las mascaras:
  hor_mask = compute_gauss_horiz(sigma, height)
  vert_mask = compute_gauss_vert(sigma, height)
  # Convolucion entre la imagen de entrada "input" y la mascara horizontal
```

```
    output = np.float32(convolution_cython(input, hor_mask))

    # Calcular convolucion entre la imagen resultante y la mascara vertical
    result = np.float32(convolution_cython(output, vert_mask) )

    return result
```

Código 3: Suavizado de imágenes.

El suavizado de imágenes consiste en concatenar filtros: primero se aplica el filtro horizontal y luego el vertical. De esta forma se obtiene el suavizado (o *blur*), que corresponde a un filtro pasabajo.

En la figura 8, se muestra el resultado de aplicar *blur* a la imagen del techo:



(a) Imagen original.

(b) Imagen obtenida.

Figura 8: Salida obtenida al usar la función *do_ blur*.

3.1.4. Submuestreo de la imagen

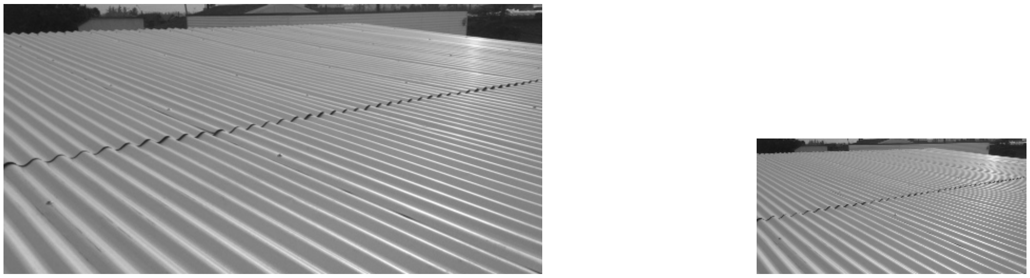
```
def subsample(input):
    # Creamos el output. Tiene la mitad del ancho y del largo de la imagen
     original.
    result = np.zeros(shape = [input.shape[0] // 2, input.shape[1] // 2],
     dtype = np.float32)

    # Nos quedamos con todos los pixeles pares (y el 0,0).
    for i in range(result.shape[0]):
      for j in range(result.shape[1]):
        result[i][j] = input[i*2][j*2]

    return result
```

Código 4: Submuestreo de imágenes.

Para el submuestreo, se solicitó que se guardaran sólo las columnas y las filas pares. Esto se asegura al recorrer la imagen original con el índice de la imagen de salida multiplicado por dos:

$$Output[i, j] \; = \; Input[2i, \; 2j],$$

Esto para los i y j mientras sean menores que $alto/2$ y $largo/2$. En la siguiente figura, se observa el resultado de aplicar el submuestreo a una imagen:



(a) Imagen original. (b) Imagen obtenida.

Figura 9: Salida obtenida al usar la función *subsample*.

Es importante destacar que no solamente se escaló la imagen en el documento, sino que, al comprobar las dimensiones de cada una en Python, se obtiene el resultado esperado:

```
# Prueba de la funcion subsample
im_reduced = subsample(input)
print("entrada.shape = ", input.shape)
print("Salida.shape = ", im_reduced.shape)
# -----------Output generado en python-------------------
entrada.shape =  (256, 512)
Salida.shape =  (128, 256)
```

Código 5: Prueba del suavizado de imágenes.

3.1.5. Piramide de Gauss: computo y grafico

```
# Calculo de la piramide de gauss:
def compute_gauss_pyramid(input, nlevels):
  gausspyramid = []
  current = np.copy(input)
  gausspyramid.append(current)
  for i in range(1,nlevels):
    current = subsample(do_blur(input = gausspyramid[i-1], sigma = 2, height
    = 7))
    gausspyramid.append(current)
  return gausspyramid

# Display de la piramide de Gauss:
def show_gauss_pyramid(pyramid):
  for i, im in enumerate(pyramid):
    lvl = i + 1
    print("Imagen para nivel ", lvl)
    cv2_imshow( im )
```

Código 6: Implementación pirámide de Gauss.

La implementación de la pirámide de Gauss es simplemente concatenar lo realizado antes e iterar la cantidad de niveles que se deseen. De esta forma, en el primer nivel está la imagen original y en los siguentes se realiza lo siguente:

1. Suavizar imagen del nivel anterior.
2. Submuestrear imagen obtenida en 1.
3. Agregar el resultado a la pirámide.

Para graficar la pirámide de Gauss, solo se iteró sobre cada imagen que esta contiene y se utilizó la función `cv2_imshow`.

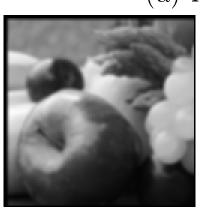
3.1.6. Resultados pirámide de Gauss

En la siguiente sección, se muestran los resultados de aplicar lo anterior con los parámetros solicitados:

Frutas:

(a) Nivel 1.

(b) Nivel 2.



(c) Nivel 3.

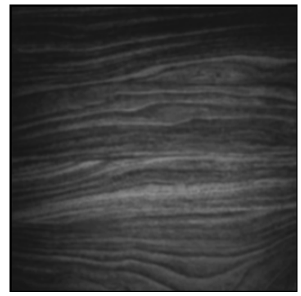
(d) Nivel 4.

(e) Nivel 5.

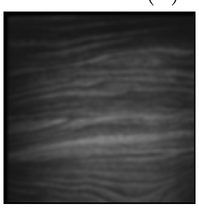
Figura 10: Pirámide de Gauss de la imagen `frutas.png`.

Madera:

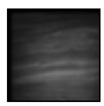
(a) Nivel 1.



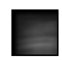
(b) Nivel 2.



(c) Nivel 3.



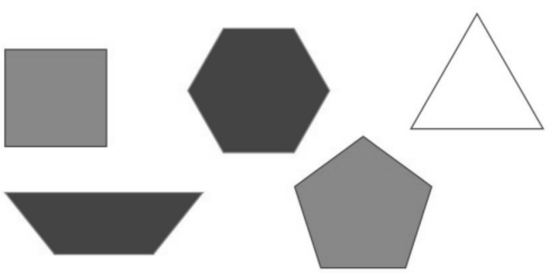
(d) Nivel 4.



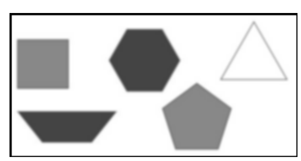
(e) Nivel 5.

Figura 11: Pirámide de Gauss de la imagen `madera.png`.

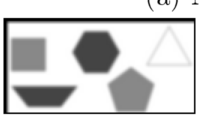
Poligonos:



(a) Nivel 1.



(b) Nivel 2.



(c) Nivel 3.

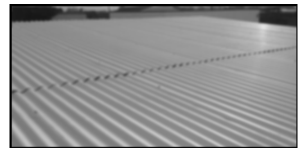
(d) Nivel 4.

(e) Nivel 5.

Figura 12: Pirámide de Gauss de la imagen `poligono.png`.

Techo:

(a) Nivel 1.



(b) Nivel 2.

(c) Nivel 3.

(d) Nivel 4.

(e) Nivel 5.

Figura 13: Pirámide de Gauss de la imagen `techo.png`.

3.2. Analisis pirámide de Gauss

En primer lugar, es importante destacar que el resultado obtenido en la pirámide de Gauss depende de varios parámetros:

- **Niveles:** Cantidad de niveles que tendrá la pirámide.
- σ**:** Desviación estándar para el filtro *blur*.
- ***Height*****:** Tamaño del filtro *blur*.

Por otro lado, se observa que los resultados son similares independiente de la imagen: En todos los casos, se observa la evidente reducción del tamaño (recordar que además estan escaladas en el documento, de otra forma se los ultimos niveles no se distinguían) y se observa también el efecto del filtro *blur*. El principal efecto es que los bordes se suavizan. Se notó también que a mayor σ o *Height*, este efecto se notaba más en las imágenes de salida.

3.3. Pirámide de *Laplace*

3.3.1. Resta de imágenes

```
def subtract(input1, input2):
  # Verificamos que sean del mismo tamano
  assert (input1.shape == input2.shape), "Imagenes deben tener igual tamano"

  output = input1 - input2
  return output
```

Código 7: Implementación resta de imágenes.

Para la función *subtract*, se debe verificar en primer lugar que ambas imágenes tengan igual dimensión. En caso de no ser así arroja error. Luego, simplemente se resta una imagen con la otra.

3.3.2. Pirámide de Laplace

```
def compute_laplace_pyramid(input, nlevels):
  gausspyramid = []
  laplacepyramid = []
  current = np.copy(input)
  gausspyramid.append(current)
  for i in range(1, nlevels):
    # Por hacer:
    # 1) Aplicar do_blur( ) a la imagen gausspyramid[i-1], con sigma 2.0 y ancho 7
    blur_im = do_blur(input = gausspyramid[i-1], sigma = 2, height = 7)
    # 2) Guardar en laplacepiramid el resultado de restar gausspyramid[i - 1] y la imagen calculada en (1)
    laplacepyramid.append(np.float32(subtract(input1 = gausspyramid[i-1], input2 = blur_im)))
    # 3) Submuestrear la imagen calculada en (1), guardar el resultado en current
    current = subsample( input = blur_im )
    gausspyramid.append(current)
  laplacepyramid.append(current)  # Se agrega el ultimo piso de la piramide de Laplace
  return laplacepyramid
```

Código 8: Cómputo pirámide de Laplace.

El cálculo de la pirámide de Laplace, como se mencionó en la sección 2.2.2, consiste en almacenar la información perdida entre un nivel y otro de la pirámide de Gauss. Para esto, se comienza con el primer piso de la pirámide de Gauss y luego se itera en lo siguiente (paso i de la iteración):

1. Suavizar (filtro *blur*) imagen del piso anterior de la pirámide de gauss ($GaussPiramyd[i-1]$).
2. Restar la imagen obtenida con la original ($GaussPiramyd[i-1]$ menos imagen recién obtenida).
3. Agregar lo obtenido a la pirámide de Laplace.

3.3.3. Escalamiento y valor absoluto de la imágen

```python
def scale_abs(input, factor):
  # Creamos imagen de salida del mismo tamano del de la entrada
  output = np.zeros_like(input)
  # Por hacer: aplicar valor absoluto a los pixeles de la imagen pixel a pixel y luego escalar los pixeles usando el factor indicado
  for i in range(input.shape[0]):
    for j in range(input.shape[1]):
      output[i][j] = np.abs(input[i][j]) * factor

  return np.float32(output)
```

Código 9: Función *scale_ abs*.

En la implementación se observa que lo que se hace es simplemente iterar por todos los pixeles calculando el valor absoluto y luego escalando por un factor ($factor$) que es un *input* de la función.

3.3.4. Graficar pirámide de Laplace

```python
1 def show_laplace_pyramid(pyramid):
2   # Por hacer: mostrar las imagenes de la piramide de laplace:
3   #  Las imagenes deben ser escaladas antes de mostrarse usando scale_abs
4   #  Sin embargo, la ultima imagen del ultimo piso se muestra tal cual
5   # Se recomienda usar cv2_imshow( ) para mostrar las imagenes
6   for i, im in enumerate(pyramid):
7     lvl = i + 1
8     print("Imagen para nivel ", lvl)
9     # Se escalan todas menos la ultima
10    if i != len(pyramid)-1:
11      cv2_imshow( scale_abs(im, 5) )
12    else:
13      cv2_imshow( im )
```

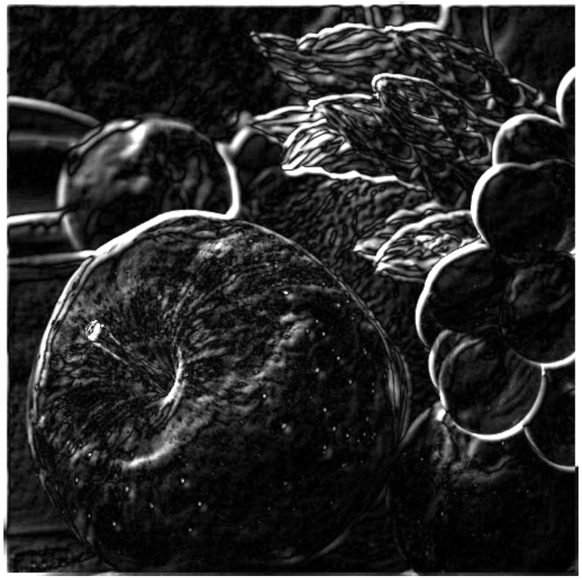
Código 10: Función para graficar pirámide de Laplace.

En esta función cada imagen, excepto la última, debe ser escalada antes de mostrarse. Para esto, se agrega el `if` en la línea 10.

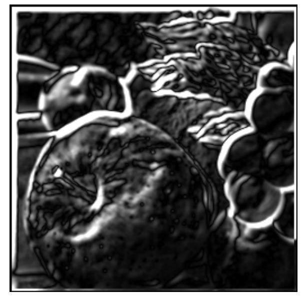
3.4. Resultados pirámide de Laplace

A continuación, se muestran los resultados obtenidos de aplicar el cálculo de las pirámides de Laplace a las 4 imágenes con los parámetros solicitados:

Frutas:



(a) Nivel 1.



(b) Nivel 2.

(c) Nivel 3.

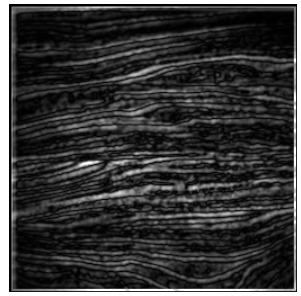
(d) Nivel 4.

(e) Nivel 5.

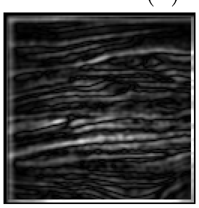
Figura 14: Pirámide de Laplace de la imagen `frutas.png`.

Madera:

(a) Nivel 1.

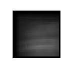


(b) Nivel 2.



(c) Nivel 3.

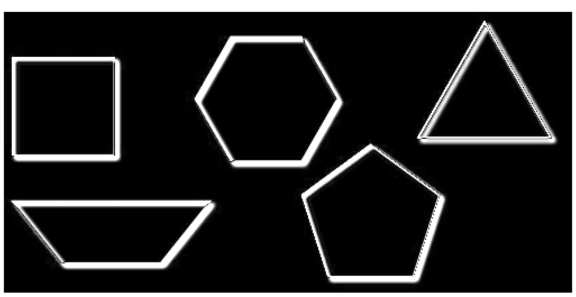
(d) Nivel 4.



(e) Nivel 5.

Figura 15: Pirámide de Laplace de la imagen `madera.png`.

Poligonos:



(a) Nivel 1.

(b) Nivel 2.

(c) Nivel 3.

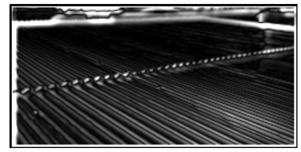
(d) Nivel 4.

(e) Nivel 5.

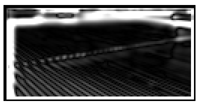
Figura 16: Pirámide de Laplace de la imagen `poligono.png`.

Techo:

(a) Nivel 1.



(b) Nivel 2.



(c) Nivel 3.

(d) Nivel 4.

(e) Nivel 5.

Figura 17: Pirámide de Laplace de la imagen `techo.png`.

- Prueba del sistema de cálculo de pirámide de Laplace sobre 4 imágenes entregadas, incluir las imágenes de las pirámides resultantes en el informe - Análisis del desempeño del cálculo de la pirámide de Laplace, analizando las imágenes resultantes

3.5. Reconstrucción imagen

- Describir implementación de suma de imágenes, incluyendo código - Describir implementación de duplicación de tamaño de imágenes con interpolación, incluyendo código - Describir implementación de reconstrucción de imagen original, incluyendo código - Prueba del sistema de reconstrucción de la imagen original usando las pirámides de las cuatro imágenes entregadas, incluir las imágenes reconstruidas en el informe - Análisis del resultado de la reconstrucción respecto a las imágenes originales

4. Conclusión

Bibliografía

[1] Wikipedia: Convolution.
https://en.wikipedia.org/wiki/Convolution#Visual_explanation

[2] Towards Data Science: Intuitively Understanding Convolutions for Deep Learning. By Irhum Shafkat.
https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning

[3] Pyramid methods in image processing. E. H. Adelson, C. H. Anderson, J. R. Bergen, P. J. Burt, J. M. Ogden. [1984] http://persci.mit.edu/pub_pdfs/RCA84.pdf

5. Anexos