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:
27 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:

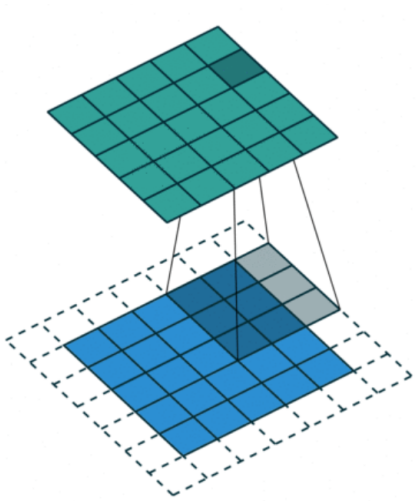


Figura 1: Convolucion en dos dimensiones con padding.[2]

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.

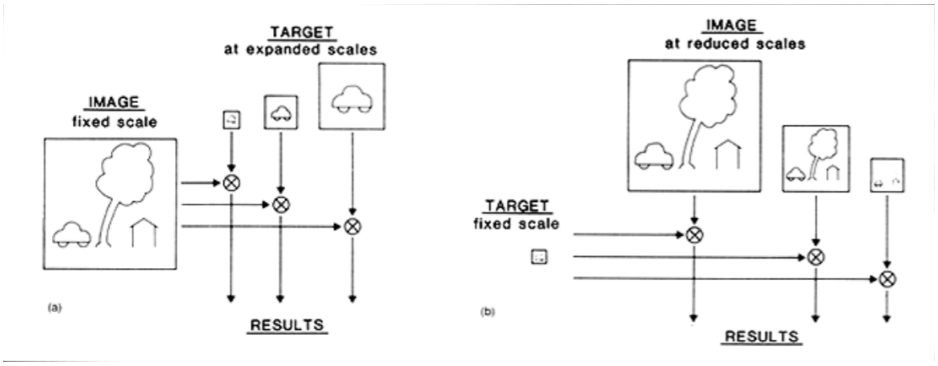


Figura 2: Distintas formas de utilizar las pirámides. En (a), copias del objeto a detectar se escalan. En (b), la imagen se trabaja completa. [3]

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.

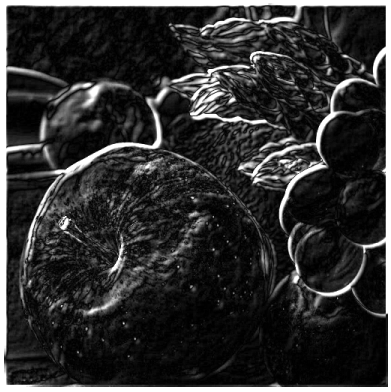
En las figuras 4 y 5, se muestran 3 niveles de la piramide de gauss y laplace respectivamente. Éstas fueron generadas a partir de la implementación de la tarea:

(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:

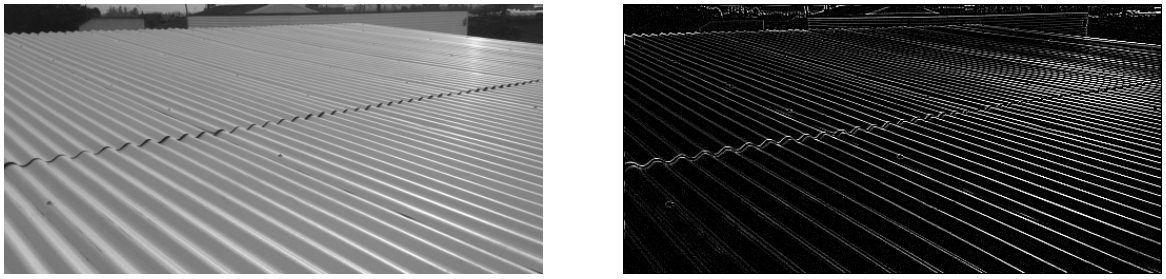
	Cython	***Scipy***
Timpo [ms]	15.8	3.08

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))
```

```
11
12    # normalizamos
13    return mask / mask.sum()
14
15  def compute_gauss_vert(sigma, height):
16    mask = np.zeros((height, 1), np.float32)
17    # no es necesario implementar nuevamente la exponencial, solo trasponer el
        resultado
18    # de la funcion comute_gauss_horiz
19    hor_mask = compute_gauss_horiz(sigma = sigma, width = height)
20    for i in range(height):
21      mask[i][0] = hor_mask[0][i]
22    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

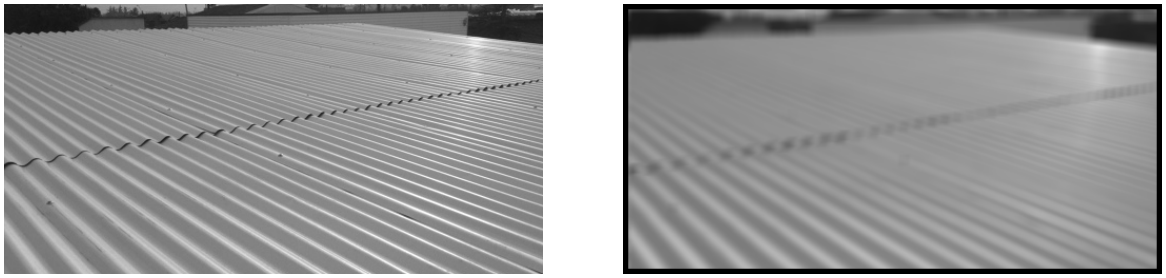
```
1  def do_blur(input, sigma, height):
2    # Obtenemos las mascaras:
3    hor_mask = compute_gauss_horiz(sigma, height)
4    vert_mask = compute_gauss_vert(sigma, height)
5    # Convolucion entre la imagen de entrada "input" y la mascara horizontal
```

```
6   output = np.float32(convolution_cython(input, hor_mask))
7
8   # Calcular convolucion entre la imagen resultante y la mascara vertical
9   result = np.float32(convolution_cython(output, vert_mask) )
10
11  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 X, 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*.

- Describir implementación de submuestreo, incluyendo código - Describir implementación de pirámide de Gauss, incluyendo código - Describir implementación: graficar pirámide de Gauss, incluyendo código - Prueba del sistema de cálculo de pirámide de Gauss sobre 4 imágenes entregadas, incluir las imágenes de las pirámides resultantes en el informe - Análisis del desempeño del cálculo de la pirámide de Gauss, analizando las imágenes resultantes

3.2. Pirámide de *Laplace*:

- Describir implementación de resta de imágenes, incluyendo código - Describir implementación de pirámide de Laplace, incluyendo código - Describir implementación de valor absoluto y escalamiento, incluyendo código - Describir implementación: graficar pirámide de Laplace, incluyendo código - 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

3.3. 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