



UNIVERSIDAD DE CASTILLA-LA MANCHA  
ESCUELA SUPERIOR DE INFORMÁTICA

DISEÑO DE INFRAESTRUCTURA DE RED

GRADO EN INGENIERÍA INFORMÁTICA  
3º - INGENIERÍA DE COMPUTADORES  
2019 – 2020

---

**Aplicación de filtros a imágenes con *MPI*  
y *Python***

---

*Autor:*

Mario Pérez Sánchez-Montaño

*Fecha:*

8 de junio de 2020

## Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Aplicación de filtros sencillos</b>	<b>3</b>
<b>3. Filtros avanzados</b>	<b>4</b>
3.1. Explicación del flujo . . . . .	5
3.2. Aplicación del filtro . . . . .	7
3.3. Unión de los píxeles calculados . . . . .	10
3.4. Estudio de la eficiencia del algoritmo . . . . .	10
3.5. Filtros implementados . . . . .	11
3.5.1. Filtros de desenfoque . . . . .	12
3.5.2. Filtros de detección de bordes . . . . .	13
3.5.3. Otros filtros . . . . .	14
<b>Referencias</b>	<b>16</b>

## 1. Introducción

El objetivo del trabajo es aplicar filtros a imágenes de forma distribuida, utilizando *MPI* en *Python*.

Un filtro se podría definir como una modificación de los píxeles de la imagen original, dando lugar una nueva imagen modificada, respecto de la original. Estos filtros van desde un cambio de color a un desenfoque o detección de bordes.

Los filtros básicos, como por ejemplo, convertir una imagen a escala de grises, se basan en operaciones básicas con los valores originales de los píxeles de la imagen, con lo cual el código no tiene mucha complejidad de cálculo.

La mayoría de filtros de una cierta complejidad utilizan una matriz de convolución. Se llama convolución al tratamiento de una matriz por otra que se suele llamar *kernel*. La primera matriz será parte de la imagen a tratar y la segunda será el *kernel*, que dependerá del efecto deseado en la imagen resultante (ver Figura 1).

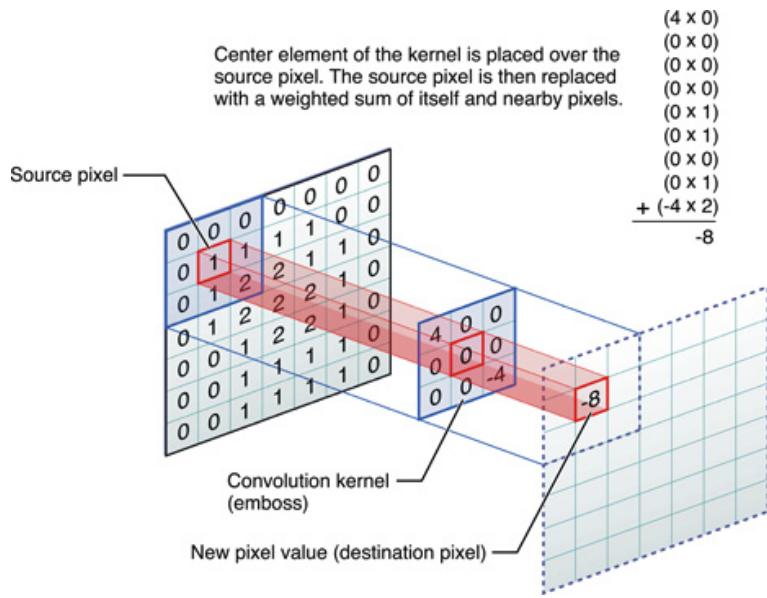


Figura 1: Ejemplo de Convolución [1]

Antes de comenzar con el programa desarrollado para aplicar este tipo de filtros, se comentará el desarrollo de un programa de aplicación de filtros básicos. A continuación, se explicará el desarrollo seguido para el programa de filtros avanzados. Por último, un pequeño estudio del comportamiento del programa en un entorno multiprocesador.

## 2. Aplicación de filtros sencillos

Como introducción a la aplicación de filtros a imágenes con *MPI* se ha desarrollado un programa para aplicar cuatro tipos de filtros: escala de grises, sepia y aplicación de un solo principal (rojo, verde y azul).

Los pasos que se siguen en el programa son los siguientes:

- Obtener los píxeles de la imagen original indicada por argumentos, transformándolos en una matriz, dónde cada posición es un vector con 3 posiciones para cada color principal (nodo principal).
- Obtener el filtro elegido por argumentos (nodo principal).
- Dividir los píxeles de la imagen según los nodos de ejecución, dando como resultado una lista del número de nodos, en la cual cada posición almacena los píxeles que debe calcular el correspondiente nodo (nodo principal).
- Enviar (*scatter*, Figura 2) los píxeles correspondientes de cada nodo (nodo principal hacia el resto de nodos)
- Enviar (*broadcast*, Figura 3) el filtro a utilizar (nodo principal hacia el resto de nodos)
- Crear los nuevos píxeles aplicando el filtro indicado (cada nodo)
- Enviar (*gather*, Figura 4) los nuevos píxeles (cada nodo hacia el nodo principal)
- Crear la nueva imagen (nodo principal)

Las operaciones colectivas aplicadas con *MPI* se pueden ver en las siguientes imágenes:

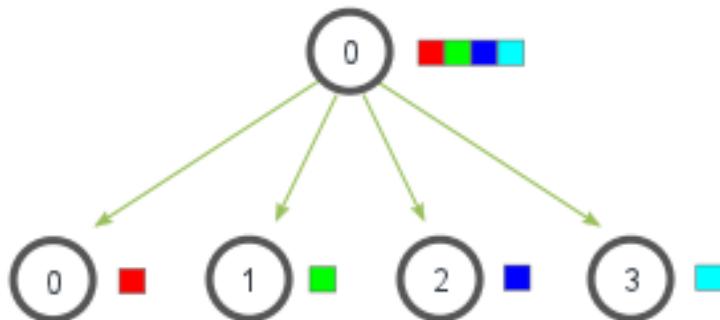
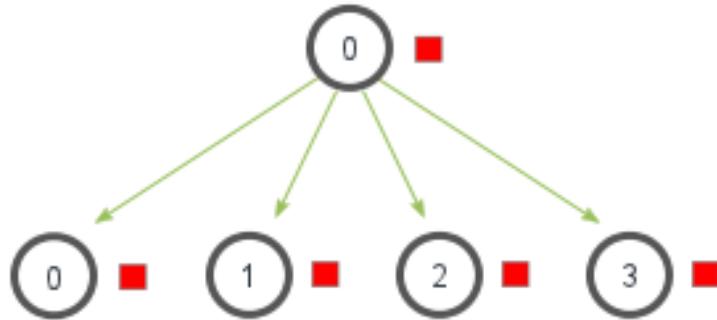
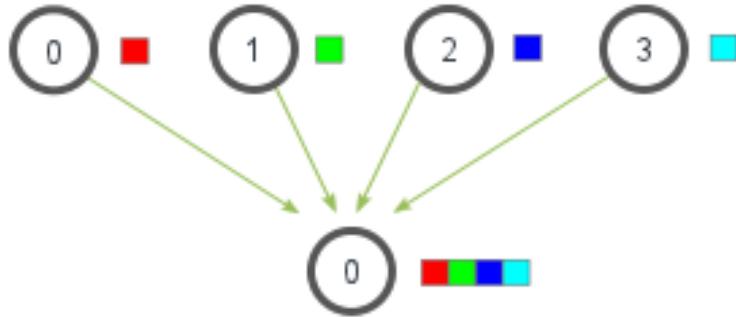


Figura 2: *Scatter* [2]

Figura 3: *Broadcast* [2]Figura 4: *Gather* [2]

### 3. Filtros avanzados

Como he comentado en la sección 1 los filtros complejos requieren una serie de operaciones mucho más costosas, en cuanto a procesamiento, que los filtros básicos que hemos visto en la sección anterior. Por ello, conviene optimizar al máximo posible nuestro programa, por eso en este caso es primordial que el desarrollo sea distribuido.

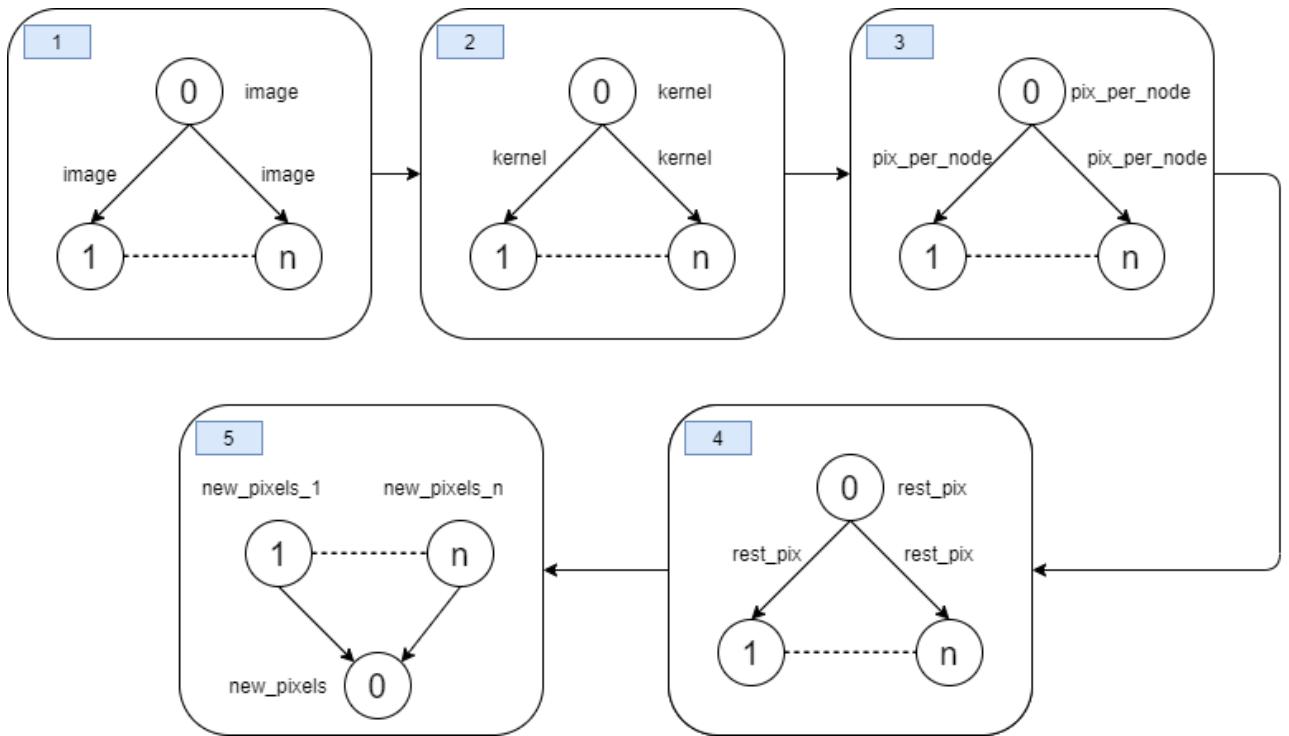
Como hemos visto en la Figura 1 se basan en multiplicar una matriz que es una parte de la imagen con otro que se considera el filtro a aplicar. Una vez aplicada la operación obtendremos un valor que será parte de la nueva imagen. En total, se han usado trece *kernels*, uno de tamaño  $5 \times 5$  y el resto de  $3 \times 3$ , en base a [3].

La idea del algoritmo es que el nodo principal creará una matriz vacía que representará la nueva imagen, con el filtro aplicado. Cada nodo calculará la parte de la imagen que le corresponde y la enviará al nodo principal (utilizando *gather*, Figura 4). Una vez recibidas las distintas partes de la imagen el nodo principal las juntará, formándose la nueva imagen.

### 3.1. Explicación del flujo

El flujo que sigue el programa durante la ejecución sería el siguiente:

- **Obtención de la imagen (nodo 0):** el nodo principal cargará la imagen (indicada en el primer argumento) y la convertirá a escala de grises utilizando *OpenCV*
- **Obtención del *kernel* a utilizar (nodo 0):** el nodo principal cargará el *kernel* indicado en el segundo argumento.
- **División de píxeles (nodo 0):** el nodo principal calculará las dimensiones de la imagen resultante en base a la imagen original y la dimensión del *kernel*. Con estos datos realizará una división equitativa de los píxeles entre los nodos de ejecución. En el caso de que la división no sea exacta, los píxeles restantes se le asignarán al último nodo.
- ***Broadcast* de la imagen (nodo 0 => nodo *i*):** se realiza una retransmisión de la imagen desde el nodo principal hacia el resto de nodos (Paso 1 de la Figura 5).
- ***Broadcast* del *kernel* (nodo 0 => nodo *i*):** de igual forma que en el anterior paso se realiza una retransmisión del *kernel* (Paso 2 de la Figura 5).
- ***Broadcast* de la cantidad de píxeles por nodo (nodo 0 => nodo *i*):** retransmisión de la cantidad calculada anteriormente por el nodo principal (Paso 3 de la Figura 5). Los píxeles restantes se envían de la misma forma con otra retransmisión (Paso 4 de la figura 5).
- ***Gather* de los nuevos píxeles calculados (nodo *i* => nodo 0):** cada nodo enviará su parte de los píxeles calculada al nodo principal y éste último juntará todas las partes en una lista. Aunque, esta unión no es correcta para crear la nueva imagen, porque se trata de una unión de listas en una lista y para construir la imagen se necesita una matriz con todos los píxeles (Sección 3.3).

Figura 5: Flujo *MPI*

El flujo principal se puede observar en el siguiente bloque de código:

```

1 if __name__ == '__main__':
2     comm = MPI.COMM_WORLD
3     rank = comm.Get_rank()
4     size = comm.Get_size()
5
6     if rank == 0:
7         start_time = time.time()
8         img = cv2.cvtColor(cv2.imread(sys.argv[1]), cv2.COLOR_BGR2GRAY)
9         kernel = k[sys.argv[2]]
10        image_h = img.shape[0]
11        image_w = img.shape[1]
12        dim = kernel[0].shape[0]
13        new_h = (image_h - dim) + 1
14        new_w = (image_w - dim) + 1
15        pix_per_node = new_h // size
16        rest_pix = new_h % size
17    else:
18        empty_pixels = None
19        img = None
20        kernel = None
21        pix_per_node = None

```

```

22     rest_pix = None
23
24     img = comm.bcast(img, root=0)
25     kernel = comm.bcast(kernel, root=0)
26     pix_per_node = comm.bcast(pix_per_node, root=0)
27     rest_pix = comm.bcast(rest_pix, root=0)
28
29     new_pixels_div = convolve2d(img, pix_per_node, rest_pix, kernel=kernel[0], mult=kernel[1])
30
31     new_pixels = comm.gather(new_pixels_div, root=0)
32
33     if rank == 0:
34         new_img = join_pixels(new_pixels)
35         cv2.imwrite(sys.argv[2] + '.' + sys.argv[1].split('.')[1], new_img)
36         print(round(time.time() - start_time, 2))

```

### 3.2. Aplicación del filtro

Esta es la parte fundamental del algoritmo y la que más me ha costado desarrollar, por ello en un primer momento el algoritmo lo desarrollé de forma convencional, sin ser distribuido.

En una primera versión la operación de convolución se realizaba con cuatro bucles, como se puede ver en el siguiente bloque de código:

```

1 def convolve2d(image, kernel, mult):
2     dim = kernel.shape[0]
3     image_h = image.shape[0]
4     image_w = image.shape[1]
5     new_h = (image_h - dim) + 1
6     new_w = (image_w - dim) + 1
7     output = np.zeros((new_h, new_w))
8
9     for x in range(dim, image_h - dim):
10        for y in range(dim, image_w - dim):
11            for m in range(dim):
12                for n in range(dim):
13                    output[x][y] += kernel[m][n] * image[x - dim + m][y - dim + n]
14
15            output[x][y] // mult
16
17    return output

```

Los dos primeros bucles son para recorrer la matriz de la nueva imagen y guardar el resultado de la aplicación del *kernel*. Los dos siguientes bucles son para realizar la

multiplicación del *kernel* con la submatriz de la imagen original [4]. Consciente de la alta complejidad de este algoritmo, comencé a pensar la forma de mejorarlo.

Los dos primeros bucles deben estar, así que me centre en eliminar los dos siguientes bucles, porque éstos al recorrer una matriz que es siempre cuadrada se pueden aprovechar las propiedades de las matrices cuadradas y recorrer el *kernel* en un solo bucle. Como se puede ver en la siguiente versión del algoritmo:

```

1 def convolve2d(image, kernel, mult):
2     dim = kernel.shape[0]
3     image_h = image.shape[0]
4     image_w = image.shape[1]
5     new_h = (image_h - dim) + 1
6     new_w = (image_w - dim) + 1
7     output = np.zeros((new_h, new_w))
8
9     for x in range(dim, image_h - dim):
10        for y in range(dim, image_w - dim):
11            for i in range(dim*dim):
12                m = i // dim
13                n = i % dim
14                output[x][y] += kernel[m][n] * image[x - dim + m][y - dim + n]
15
16                output[x][y] // mult
17
18    return output

```

Utilizando la división entera para obtener el índice de las filas y el módulo para el índice de las columnas. De esta forma ahorramos un bucle, aunque aún así tres bucles anidados me parecían demasiada complejidad. Por ello, desarrollé una versión recursiva de la aplicación del *kernel*, que se puede ver a continuación:

```

1 def convolve2d(image, kernel, mult):
2     def mult_matrix(kernel, image, x, y, i, dim):
3         if i < dim * dim:
4             m = i // dim
5             n = i % dim
6             result = kernel[m][n] * image[x - dim + m][y - dim + n]
7             return result + mult_matrix(kernel, image, x, y, i + 1, dim)
8         else :
9             return 0
10
11    dim = kernel.shape[0]
12    image_h = image.shape[0]
13    image_w = image.shape[1]
14    new_h = (image_h - dim) + 1
15    new_w = (image_w - dim) + 1
16    output = np.zeros((new_h, new_w))
17
18    for x in range(dim, image_h - dim):
19        for y in range(dim, image_w - dim):

```

```

20     output[x][y] = mult_matrix(kernel, image, x, y, 0, dim)
21     output[x][y] /= mult
22
23 return output

```

Esta versión esta basada en la idea anterior, aprovechando la propiedad de las matrices cuadradas, con la división entera y el módulo para los índices. De esta forma, el algoritmo mejoró mucho en cuanto al tiempo que tardaba en aplicar el filtro.

Una vez conseguido el algoritmo era el momento de realizar la versión distribuida del mismo. En un principio parecía que la adaptación era muy sencilla, simplemente que cada nodo aplicará el algoritmo a su parte. El problema es que hay que indicar el comienzo y final de la parte de cada nodo.

```

1 def convolve2d(image, pix_per_node, rest_pix, kernel, mult):
2     def mult_matrix(kernel, image, x, y, i, dim):
3         if i < dim * dim:
4             m = i // dim
5             n = i % dim
6             result = kernel[m][n] * image[x - dim + m][y - dim + n]
7             return result + mult_matrix(kernel, image, x, y, i + 1, dim)
8         else:
9             return 0
10
11    dim = kernel.shape[0]
12    image_w = image.shape[1]
13    start = pix_per_node * rank
14    end = start + pix_per_node if rank != size - 1 else start + pix_per_node + rest_pix
15    new_w = (image_w - dim) + 1
16    output = np.zeros((end - start, new_w), dtype=np.int32)
17
18    for x in range(start, end):
19        for y in range(dim, image_w - dim):
20            output[x - start][y] = mult_matrix(kernel, image, x, y, 0, dim)
21            output[x - start][y] /= mult
22
23 return output

```

El problema se solucionó calculando el rango de píxeles con el que trabajará el nodo correspondiente, teniendo en cuenta que el reparto puede no ser exacto. En este caso, los píxeles restantes se le asignarán al último nodo. Como el resultado se almacena en una matriz parcial, se deberá restar el valor del índice de la fila calculado con el límite inferior que define el comienzo del nodo. Esto no hay que realizarlo para las columnas, porque el reparto se hace en base a la filas, por lo tanto la matriz parcial tendrá el mismo número de columnas que la matriz completa.

Cabe destacar que, la matriz que representa la imagen original, el *kernel*, el número de

píxeles por nodo y los píxeles restantes se reciben del nodo principal (utilizando *broadcast*, Figura 3).

### 3.3. Unión de los píxeles calculados

Cuando cada nodo ha calculado los píxeles que le correspondían se los enviarán al nodo principal utilizando la operación MPI *gather* (Figura 4). El nodo principal recibirá cada parte en una lista conjunta formada tantos elementos como nodos se hayan ejecutado, cada elemento será a su vez una lista con los píxeles calculados. Este formato no es adecuado para construir la nueva imagen con el filtro aplicado. Por ello, el nodo principal convertirá esta lista de sublistas en una matriz, utilizando la siguiente función:

```

1 def join_pixels (new_pixels):
2     pixels = np.zeros((new_h, new_w), dtype=np.int32)
3
4     i = 0
5
6     for node_pixels in new_pixels:
7         pixels [i: i + len(node_pixels)] = node_pixels
8         i += len(node_pixels)
9
10    return pixels

```

Cada parte contenida en la lista recibida por los nodos se almacenará en la correspondiente parte de una matriz, con la cual se construirá la nueva imagen. El bucle se ha optimizado lo máximo posible, realizando solo una iteración por parte o nodo. Posteriormente, el nodo principal construirá la nueva imagen con la matriz de píxeles recibida de la llamada a la función anterior, utilizando de nuevo *OpenCV*.

### 3.4. Estudio de la eficiencia del algoritmo

Una vez creado el algoritmo distribuido es conveniente comprobar su eficiencia, ejecutándolo con distinto número de nodos y comprobando el tiempo que tarda en generar la nueva imagen con el filtro correspondiente aplicado. Se ha realizado un estudio de un *kernel* de  $3 \times 3$  y de  $5 \times 5$  de una imagen de  $1070 \times 713$  con distinto número de nodos, como se puede ver en la Figura 6.

Como se puede apreciar el tiempo de ejecución del algoritmo cuando lo ejecutamos con varios nodos se reduce considerablemente, alrededor de un 50 %. Esta bajada es aún más notable con el *kernel* de  $5 \times 5$ .

Otra consideración que se puede extraer de la Figura 6 es que el aumento de nodos llega un punto en el que no mejora el tiempo de ejecución del algoritmo, incluso se puede

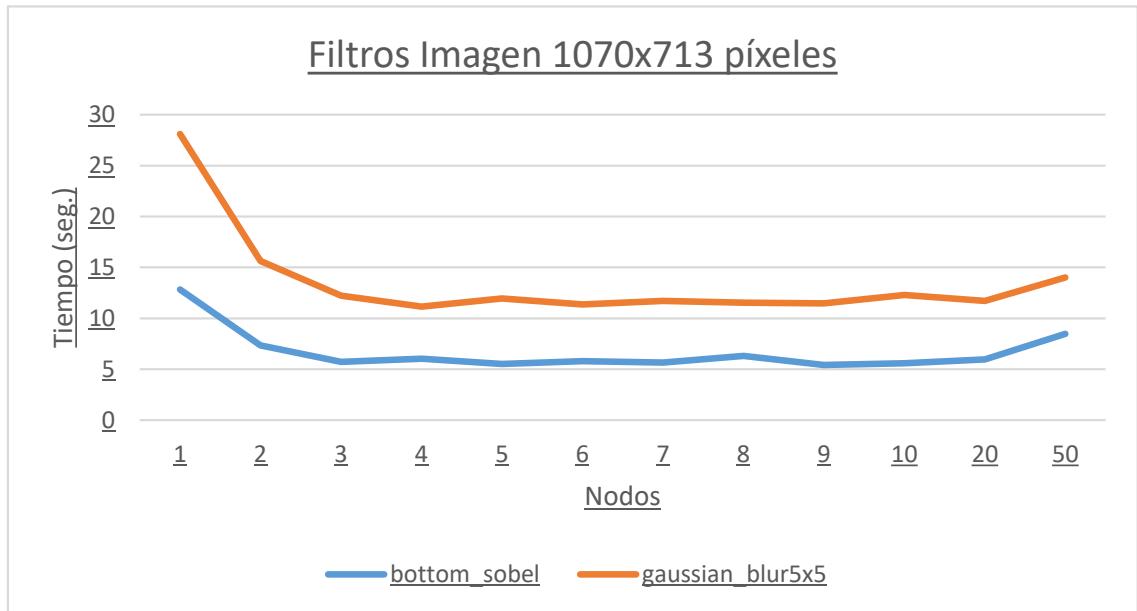


Figura 6: Comparación de ejecución del algoritmo

apreciar una pérdida de rendimiento a partir de veinte nodos. Esto se debe, principalmente, a que el procesador tiene dos núcleos, con lo cuál, una ejecución con muchos más nodos que dos no tiene mucho sentido.

### 3.5. Filtros implementados

En el programa se han implementado un total de trece filtros, de varios tipos, como veremos en las siguientes imágenes.



Figura 7: Imagen original

### 3.5.1. Filtros de desenfoque

Los *kernels* utilizados para este tipo de filtros son los siguientes:

```
1  "blur": [np.array ([[0.0625, 0.125, 0.0625],  
2                  [0.125, 0.25, 0.125],  
3                  [0.0625, 0.125, 0.0625]]), 1],  
4  
5  "box_blur": [np.array ([[1, 1, 1],  
6                      [1, 1, 1],  
7                      [1, 1, 1]]), 9],  
8  
9  "gaussian_blur3x3": [np.array ([[1, 2, 1],  
10                     [2, 4, 2],  
11                     [1, 2, 1]]), 16],  
12  
13 "gaussian_blur5x5": [np.array ([[1, 4, 6, 4, 1],  
14                     [4, 16, 24, 16, 4],  
15                     [6, 24, 36, 24, 6],  
16                     [4, 16, 24, 16, 4],  
17                     [1, 4, 6, 4, 1]]), 256]
```

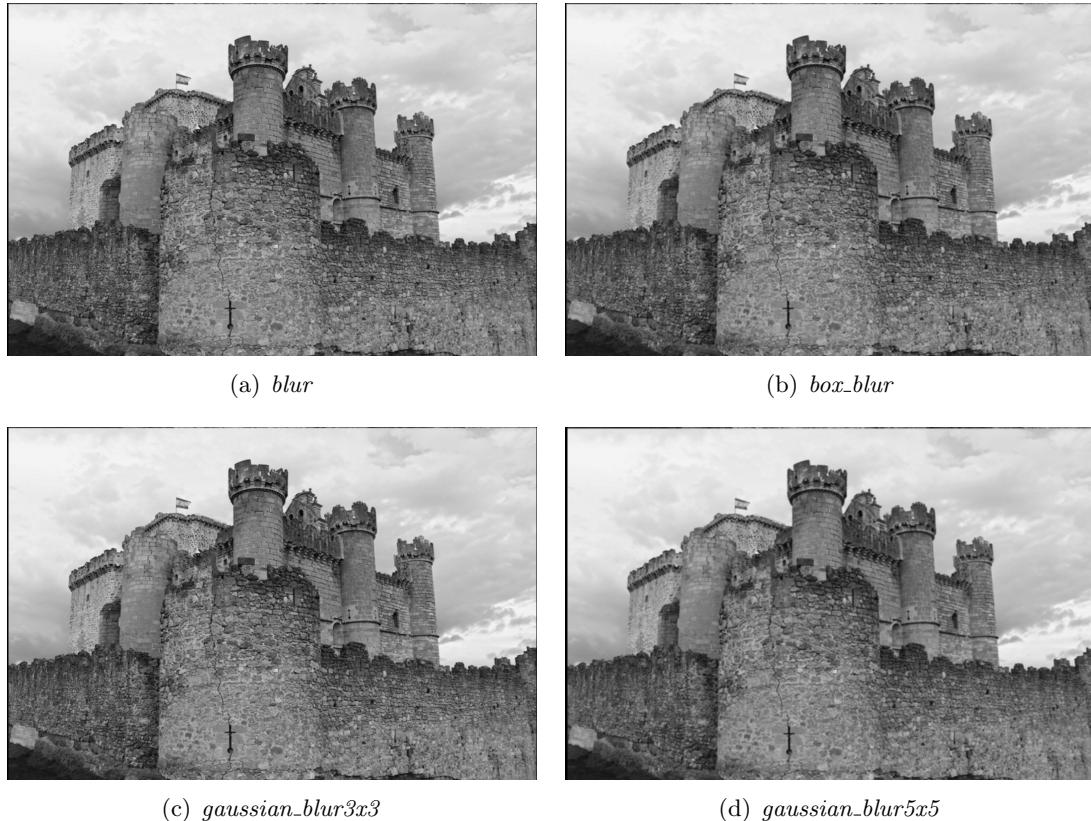


Figura 8: Filtros de desenfoque

### 3.5.2. Filtros de detección de bordes

Los *kernels* utilizados para los filtros de esta categoría se pueden ver a continuación:

```

1   "bottom_sobel": [np.array([[-1, -2, -1],
2                               [0, 0, 0],
3                               [1, 2, 1]]), 1],
4
5   "left_sobel": [np.array([[1, 0, -1],
6                           [2, 0, -2],
7                           [1, 0, -1]]), 1],
8
9   "right_sobel": [np.array([[-1, 0, 1],
10                          [-2, 0, 2],
11                          [-1, 0, 1]]), 1],
12
13  "top_sobel": [np.array([[1, 2, 1],
14                         [0, 0, 0],
15                         [-1, -2, -1]]), 1]

```

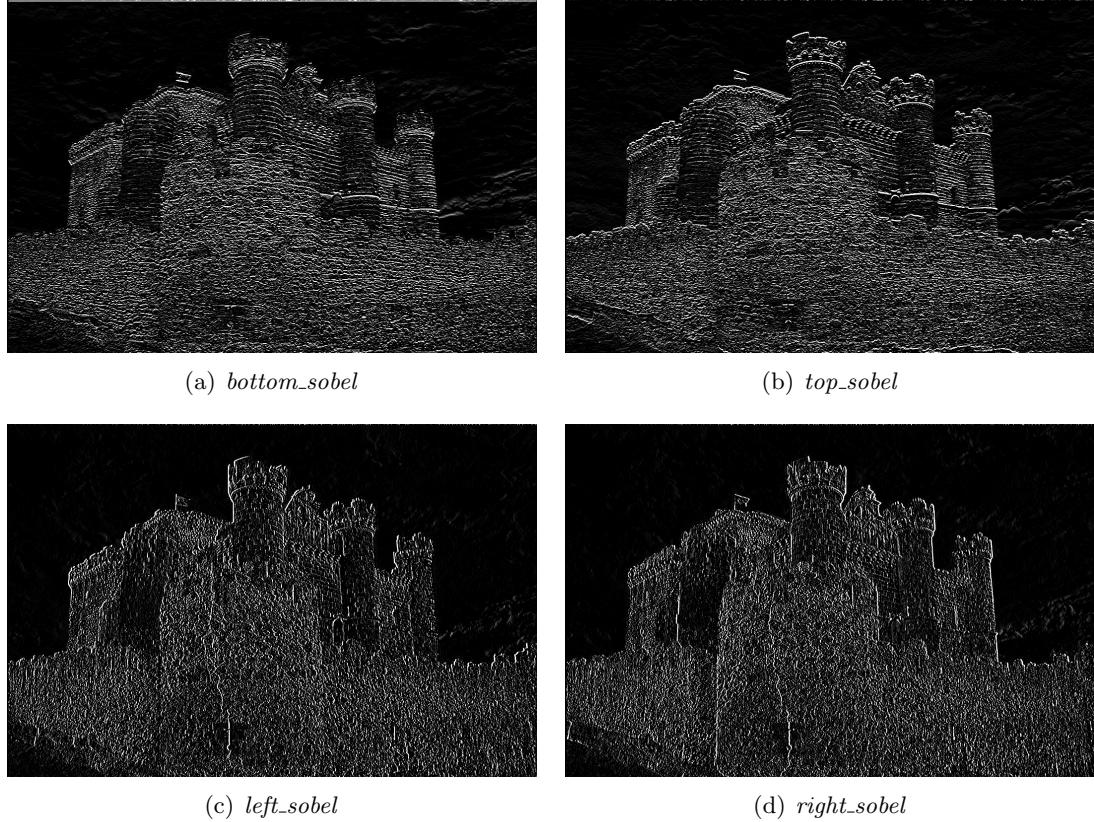


Figura 9: Filtros de detección de bordes

### 3.5.3. Otros filtros

Los *kernels* de otros filtros sin categorizar son los siguientes:

```

1   "emboss": [np.array([[ -2, -1, 0],
2                         [-1, 1, 1],
3                         [ 0, 1, 2]]), 1],
4
5   "identity": [np.array([[ 0, 0, 0],
6                         [ 0, 1, 0],
7                         [ 0, 0, 0]]), 1],
8
9   "outline": [np.array([[ -1, -1, -1],
10                          [-1, 8, -1],
11                          [-1, -1, -1]]), 1],
12
13  "sharpen": [np.array([[ 0, -1, 0],
14                         [-1, 5, -1],
15                         [ 0, -1, 0]]), 1],
16

```

```
17     "enhance": [np.array ([[0, 1, 0],  
18                           [1, -4, 1],  
19                           [0, 1, 0]]), 1]
```

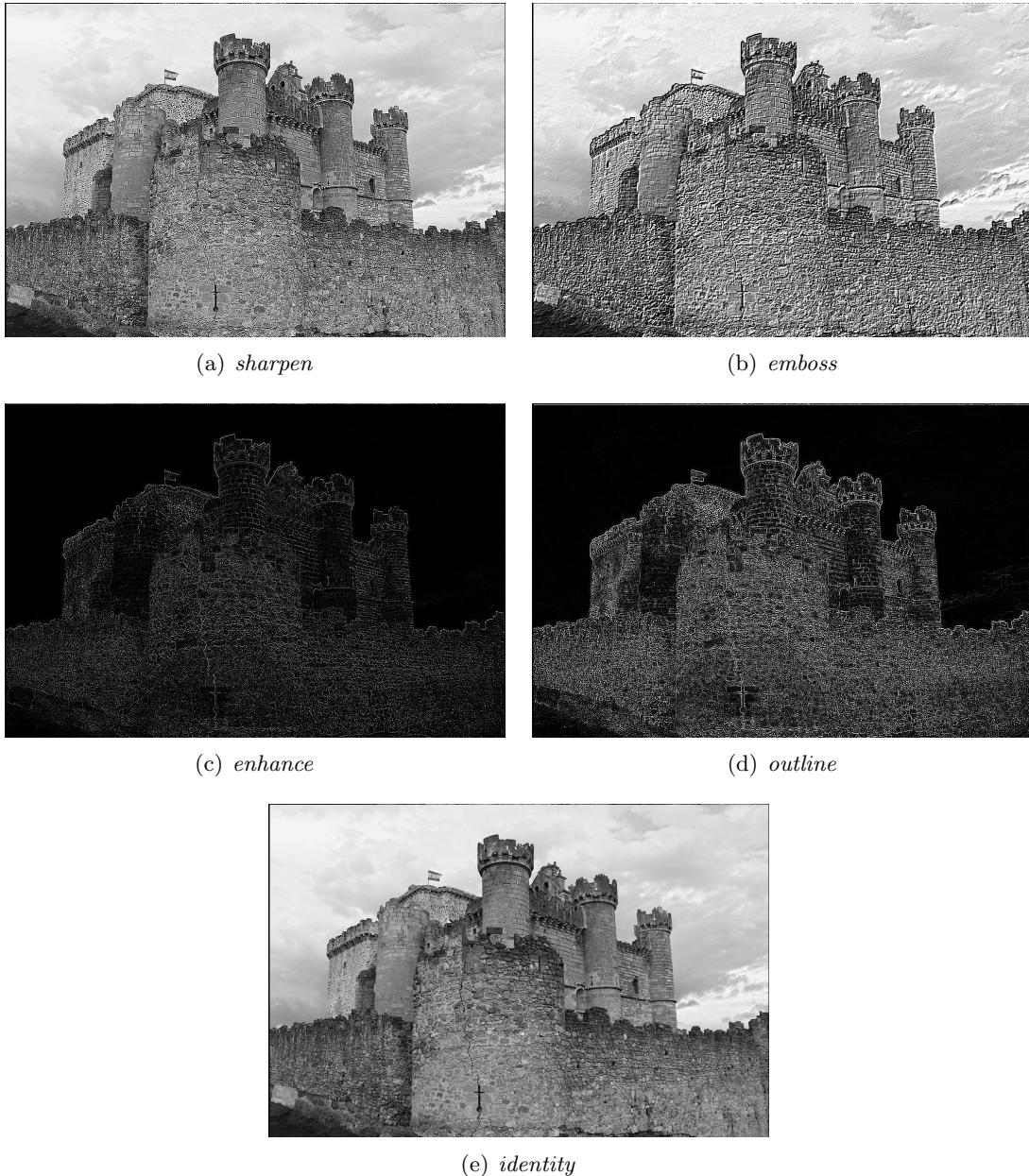


Figura 10: Otros filtros

## Referencias

- [1] <https://numfactory.upc.edu/web/NumMethods/imageProcess/ImageProcess.html>. Ejemplo convolución.
- [2] <https://nyu-cds.github.io/python-mpi/05-collectives/>. Operaciones colectivas con mpi4py.
- [3] <https://setosa.io/ev/image-kernels/>. Aplicación de kernels online.
- [4] <https://www.youtube.com/watch?v=BPBTmXKtFRQ>. Convolución en python.