Assignment 2

For the second assignment I utilized python 3.7 alongside opencv2, numpy, and matplotlib to read in images, run them through various filters, and display the results. For every smoothing algorithm specified, I ran each required image through them at various kernel sizes, then displayed them side by side for comparison. Additionally, the Laplacian sharpening algorithm was run on the two given images and the results were displayed alongside each other. Finally, a gaussian pyramid was created using the “lenna” image and displayed according to the slide specified in the instructions.

I began by creating a separate module for each part of the assignment: s*moothing.py, sharpening.py,* and *pyramid.py*. Additionally, a *display.py* module was created for formatting and displaying the results and a utility.py module for holding shared utility functions. Then, in the main I began by reading in all the images to be used throughout the program using opencv2. All were read in as grayscale to obtain single-channel images e.g.:

*fig\_312 = cv2.imread('resources/Fig0312(a)(kidney).tif', 0)*

The rest of main consists only of display functions which call functions from the display module to run the various algorithms on a given list of images and display the results e.g.:

*display.display\_med\_filter([fig\_312, fig\_333, fig\_334, fig\_335])*

To begin with, there are two notable functions within the utility module that are called repeatedly thoughout the program. These functions are *prepare\_image()* and ­*pp\_image()*. They prepare a given image for processing and postprocess a given image, respectively. The image preparation algorithm primarily creates a new image with the original image’s dimensions for use in the image processing process and pads the original image according to the input specification. This padding can either be zeroes or copy the nearby pixel. It then sets the type of the images and returns them. The post processing algorithm’s primary focus is preparing the given image for display, typically simply by setting the type and returning it.

For part 1 of the assignment, all code was placed in the smoothing module. It consists of three main functions: *avg\_filter(), med\_filter(),* and *gauss\_filter().* For running an averaging smoothing algorithm, the function avg\_filter() is used. It takes in the image to be smoothed, the kernel size, a Boolean for whether to use a weighted mask, and whether to use the alternate weighted mask. Its definition can be seen here:

*def avg\_filter(im: np.ndarray, kernel: int = 1, weighted: bool = False, w\_alt: bool = True):*

If a weighted algorithm is not specified, a simple box algorithm smoothing is run on the given image. Using the given kernel size, a mask is created using the *numpy.ones()* function:

*\_filter = np.ones((2\*kernel+1, 2\*kernel+1), dtype='int')*

For example, a given kernel size of 1 would create a 3x3 matrix of all ones i.e. [[1,1,1],[1,1,1],[1,1,1]]. Next a function from the utility module is used on the given image in order to zero pad the image based on the kernel size and create a new image which will hold the smoothed image e.g.:

*im, im\_proc = prepare\_image(im, kernel, 'zero')*

A divisor is then calculated by looping through the mask and adding up its values. Next, the mask is convolved with the image and then divided by the previously calculated divisor in order to normalize it. While I originally used my own algorithm to obtain the convolution, I later switched to the *scipy.signal.convolve2d()* as it is far more efficient. My original logic can still be seen in the *avg\_filter\_defunct()* function within the same smoothing module. See below a convolving example:

*im\_proc=scipy.signal.convolve2d(im,\_filter,'valid')  
im\_proc /= div*

Finally, the processed image is run through a final postprocessing function and returned. The procedure is nearly identical if a weighted algorithm is specified, although it is only performed as a 3x3 mask. If a weighted average is specified, a mask from an additional *masks.py* module is selected based on the input for the *w\_alt* Boolean. For example, the standard mask used consists of a center weighted 3x3 matrix i.e. [[0,1,0],[1,2,1],[0,1,0]]. This serves as a rough imitation of a gaussian smoothing algorithm. Following this, all other procedures are identical as those in the box filter.

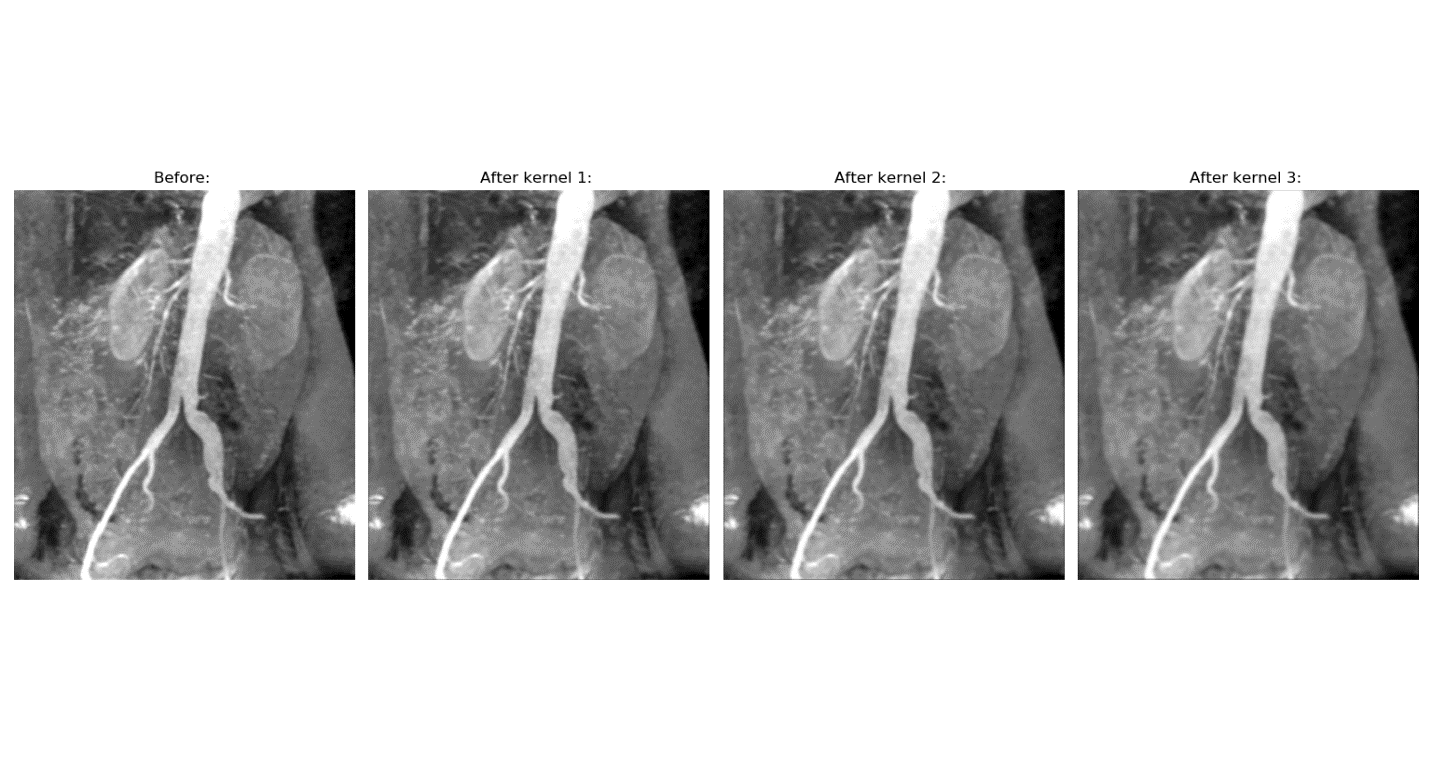


Figure : Box filter at various kernel sizes on image 312

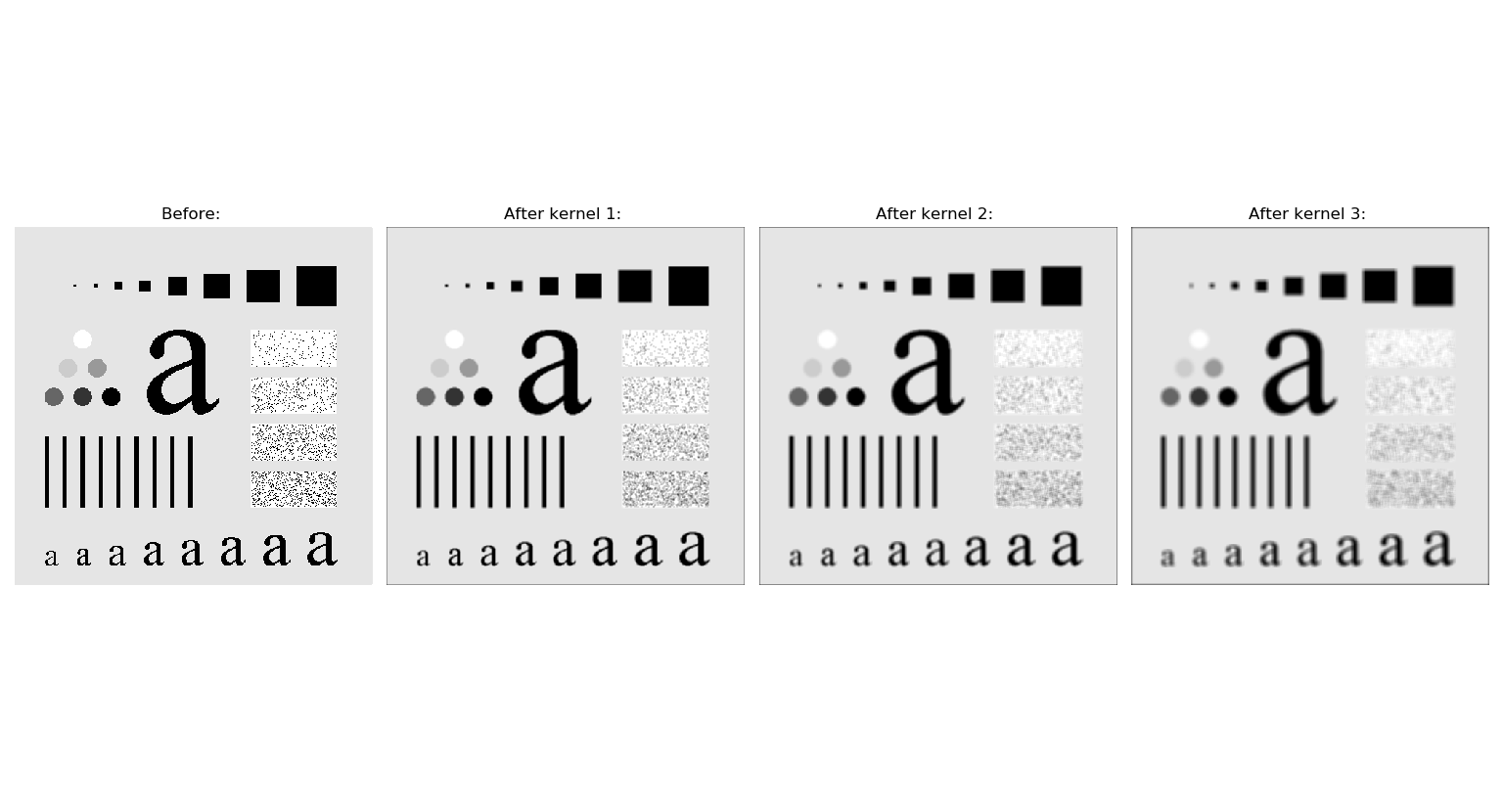


Figure : Box filter at various kernel sizes on image 333

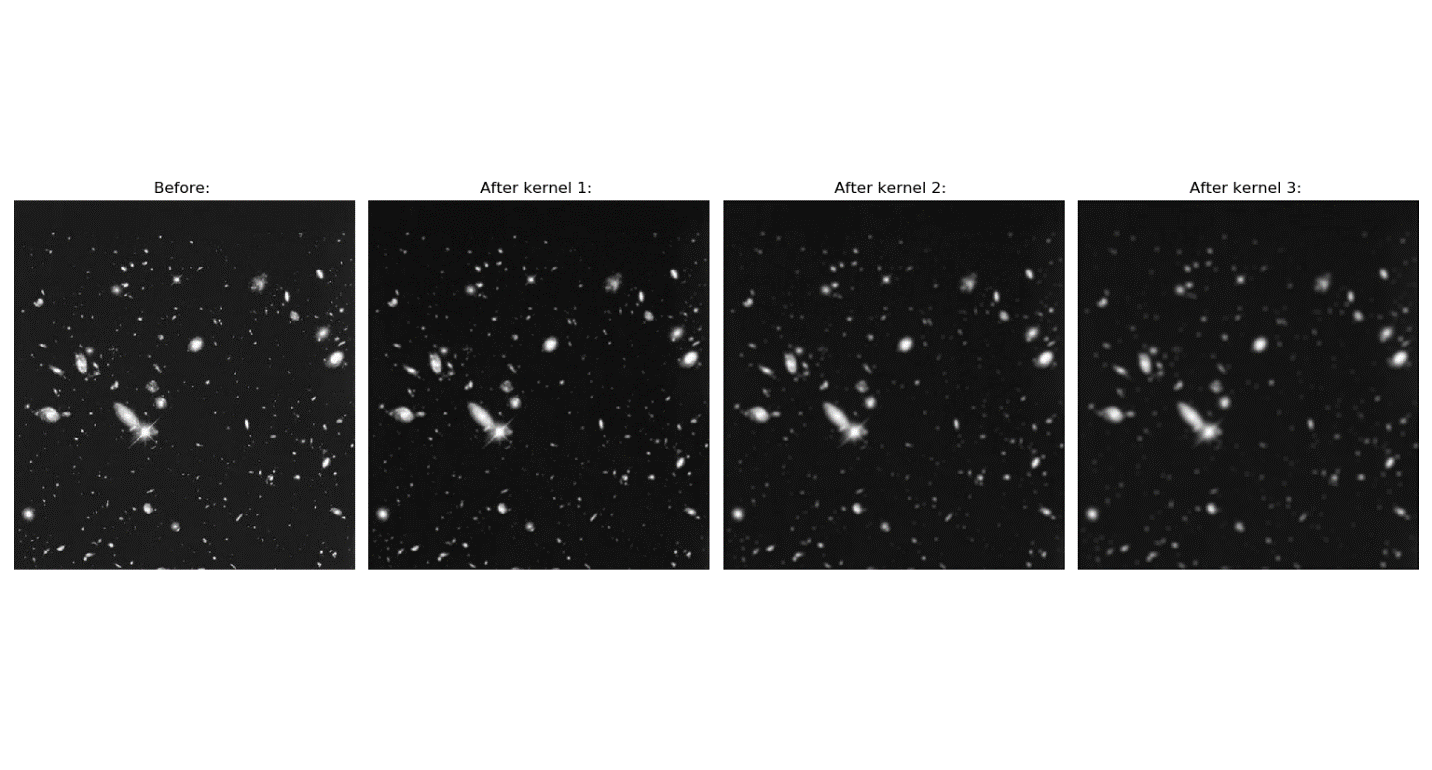


Figure : Box filter at various kernel sizes on image 334

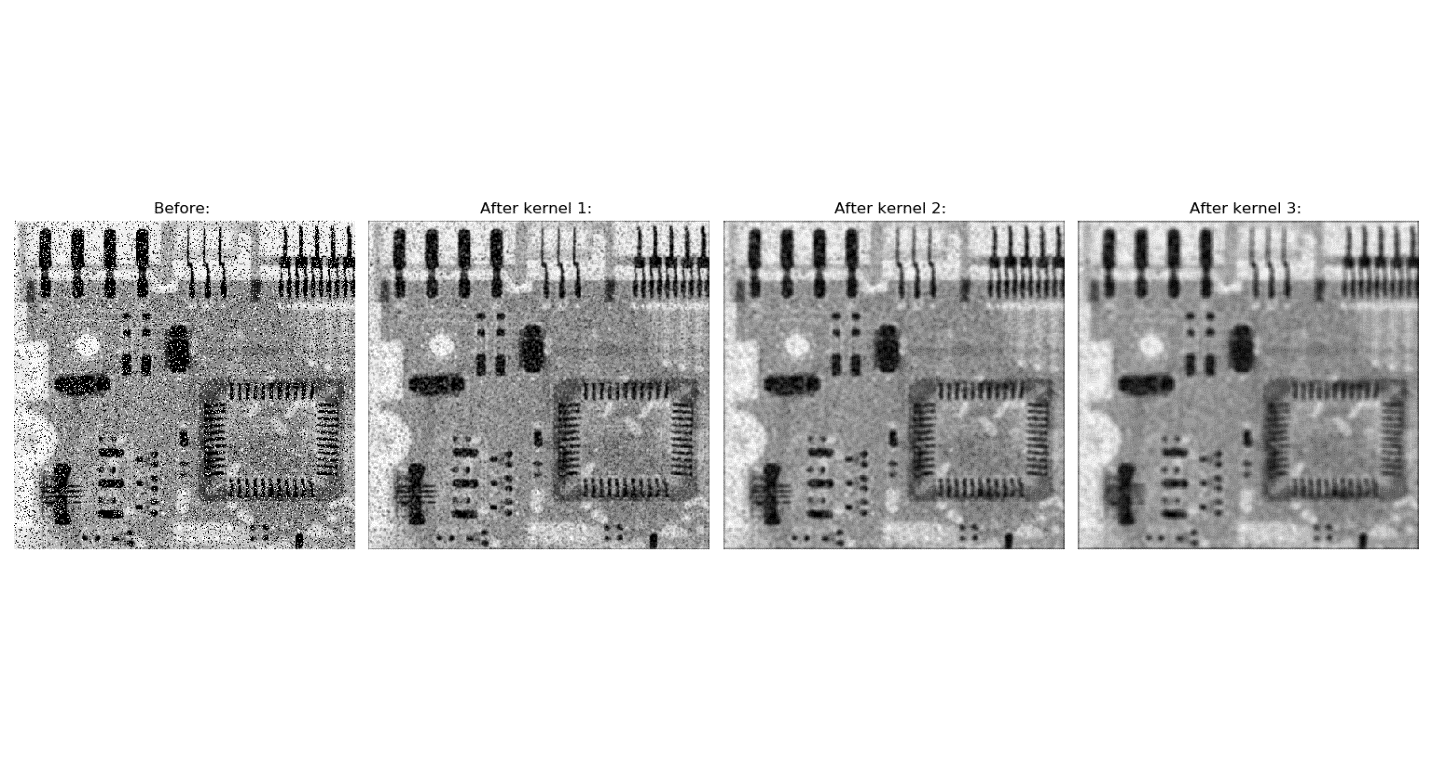


Figure : Box filter at various kernel sizes on image 335

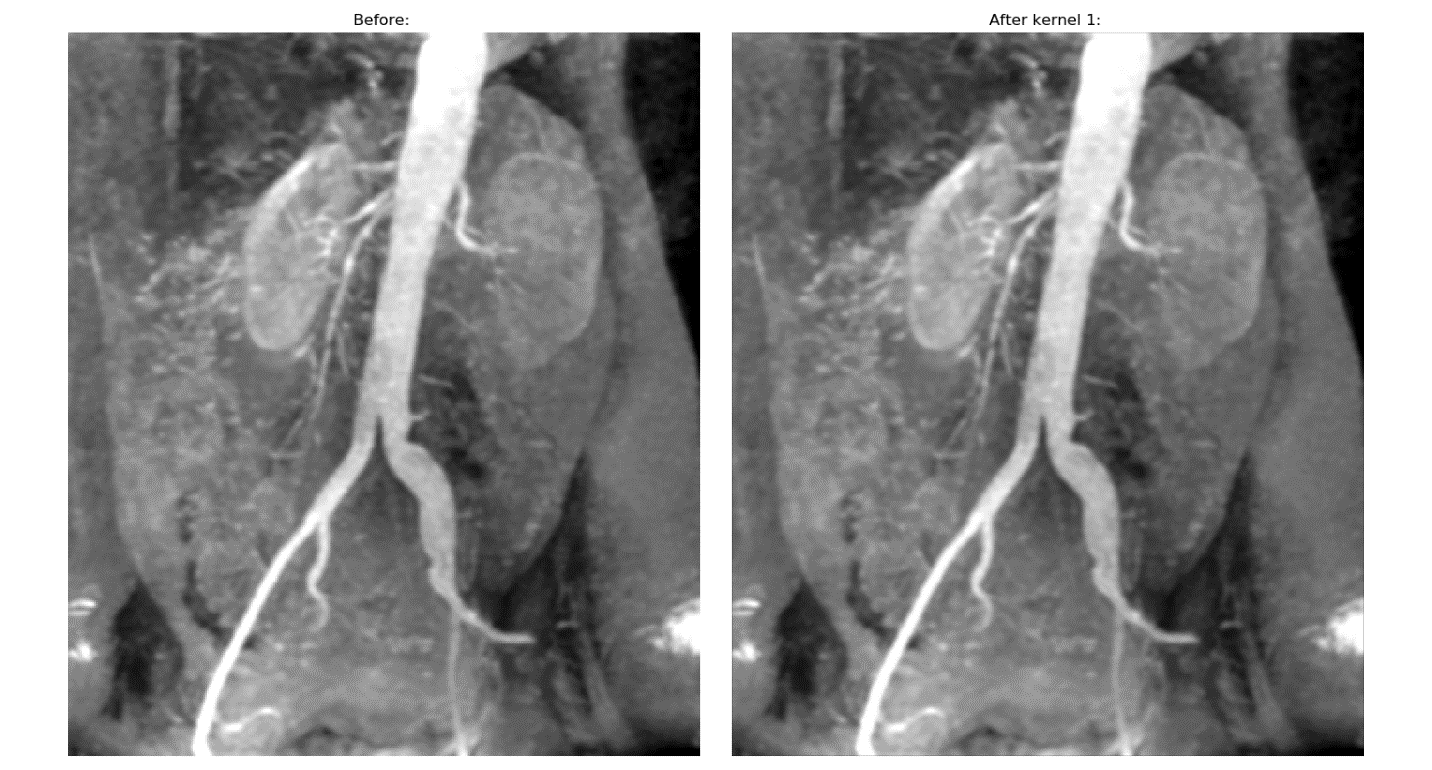


Figure : Weighted average filter at kernel size 1 on image 312

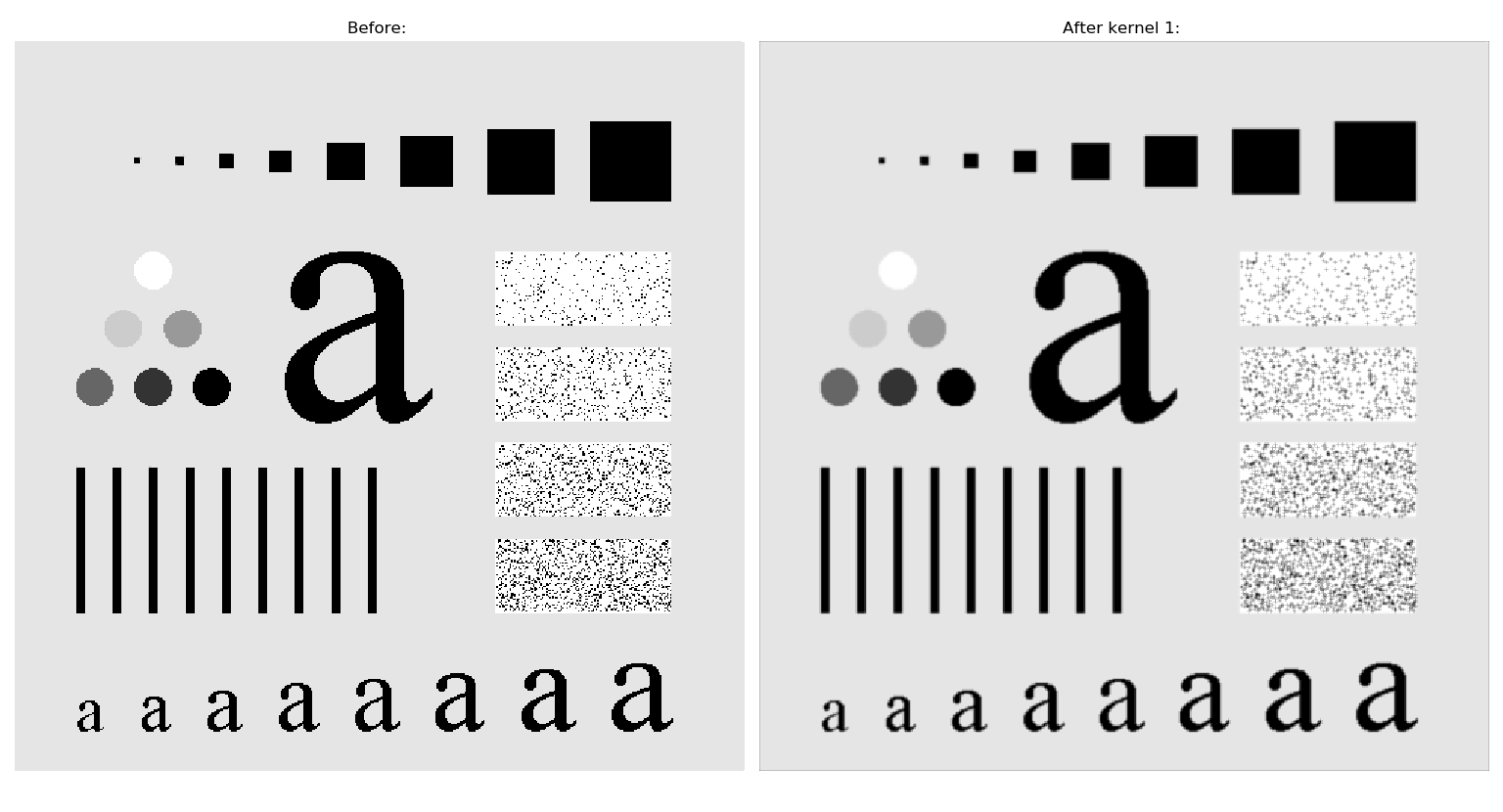


Figure : Weighted average filter at kernel size 1 on image 333

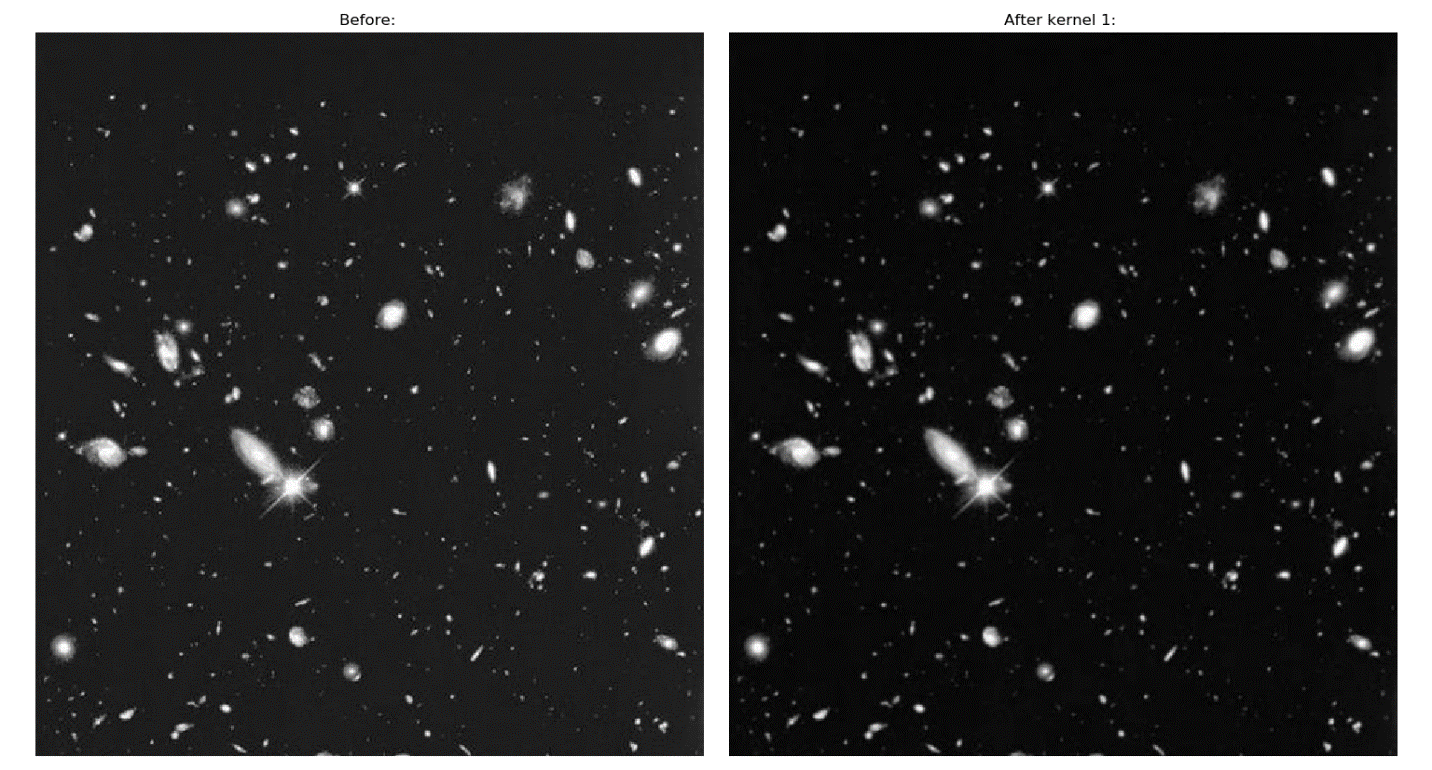


Figure : Weighted average filter at kernel size 1 on image 334

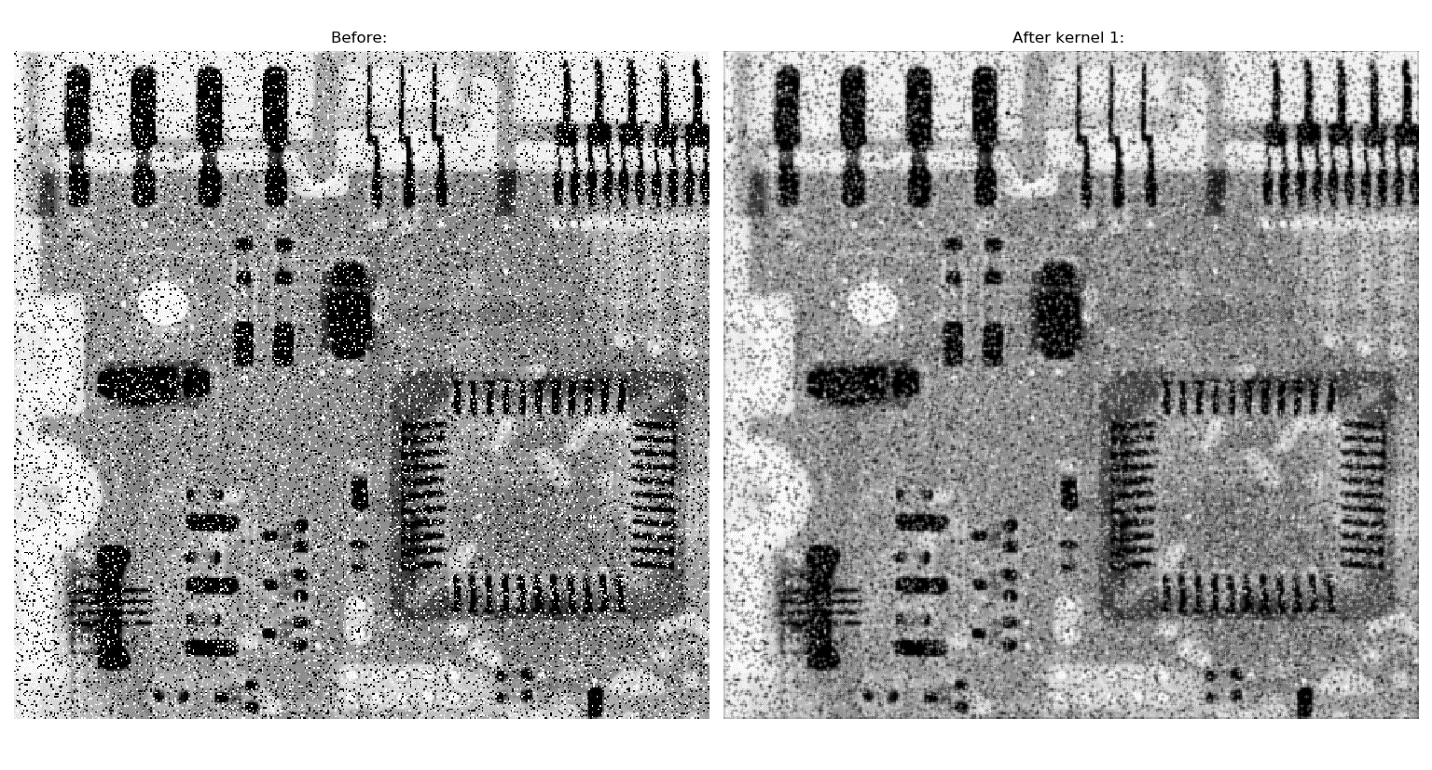


Figure : Weighted average filter at kernel size 1 on image 335

The next smoothing algorithm is the median filter, which uses the *med\_filter()* function. The function takes in the image to be processed and the desired kernel size. The image is then prepared by running it through the utility module’s prepare image function which zero pads the image based on the given kernel size and creates a new image for holding the processed image e.g.:

*im, im\_proc = prepare\_image(im, kernel, 'zero')*

In order to actually process the image, all its pixels are looped through and for each pixel a sliding window is created out of the neighboring pixels. The size of this window is based on the kernel size, and each of its values is placed into a list which is then sorted and the median value is then placed into the new image at the corresponding location. For example, see the following code snippet:

*window = []*

*for i in range(0 + kernel, dimensions[0] - (2 \* kernel)):*

*for j in range(0 + kernel, dimensions[1] - (2 \* kernel)):*

*for k in range(-kernel, kernel + 1):*

*for l in range(-kernel, kernel + 1):*

*window.append(im[i + k, j + l])*

*# sort pixels from sliding window and then place median value in processed image*

*window.sort()*

*im\_proc[i, j] = window[math.ceil(((kernel \* 2 + 1) \*\* 2) / 2)]*

*window.clear() # clear sliding window list for next iteration*

Finally, the processed image is run through the post processing function and returned.

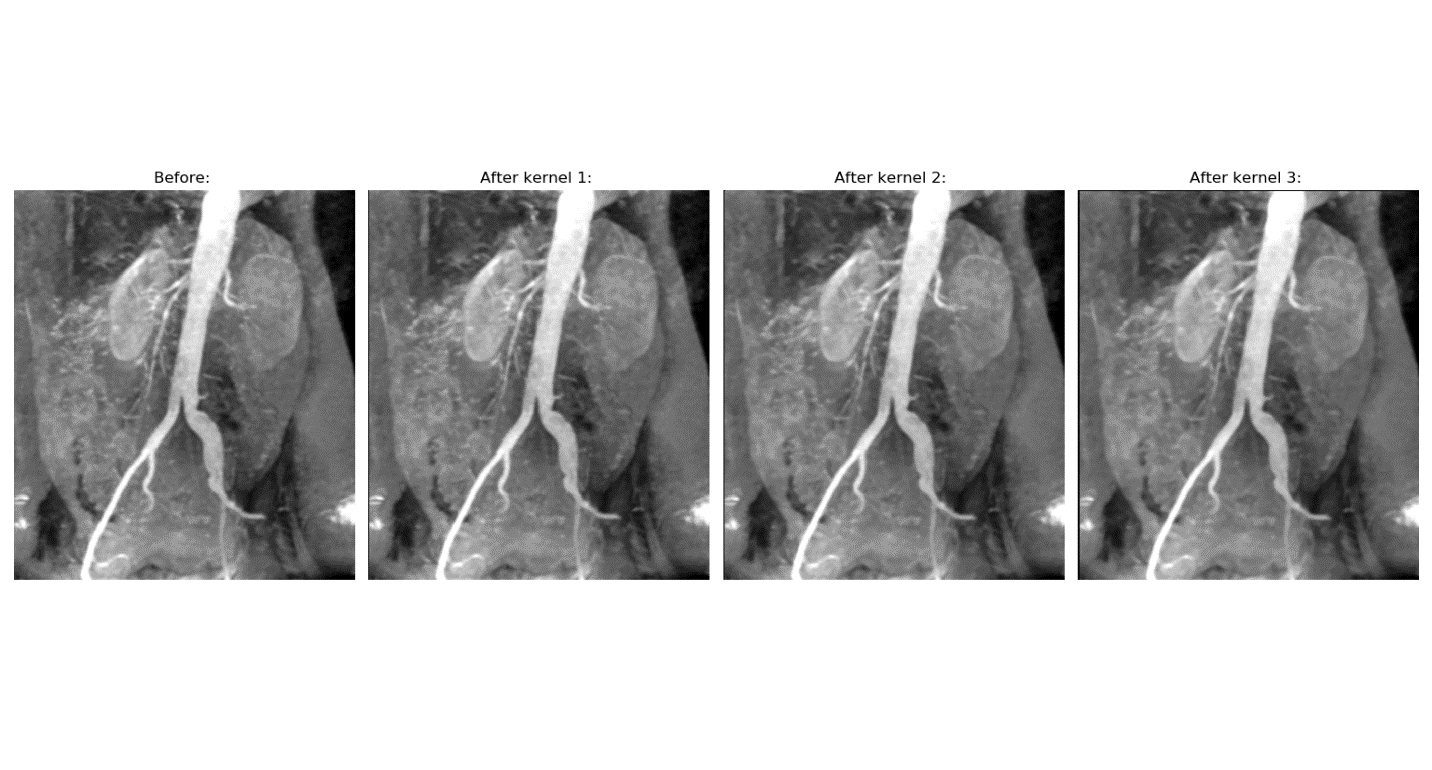


Figure : Median filter at various kernel sizes on image 312

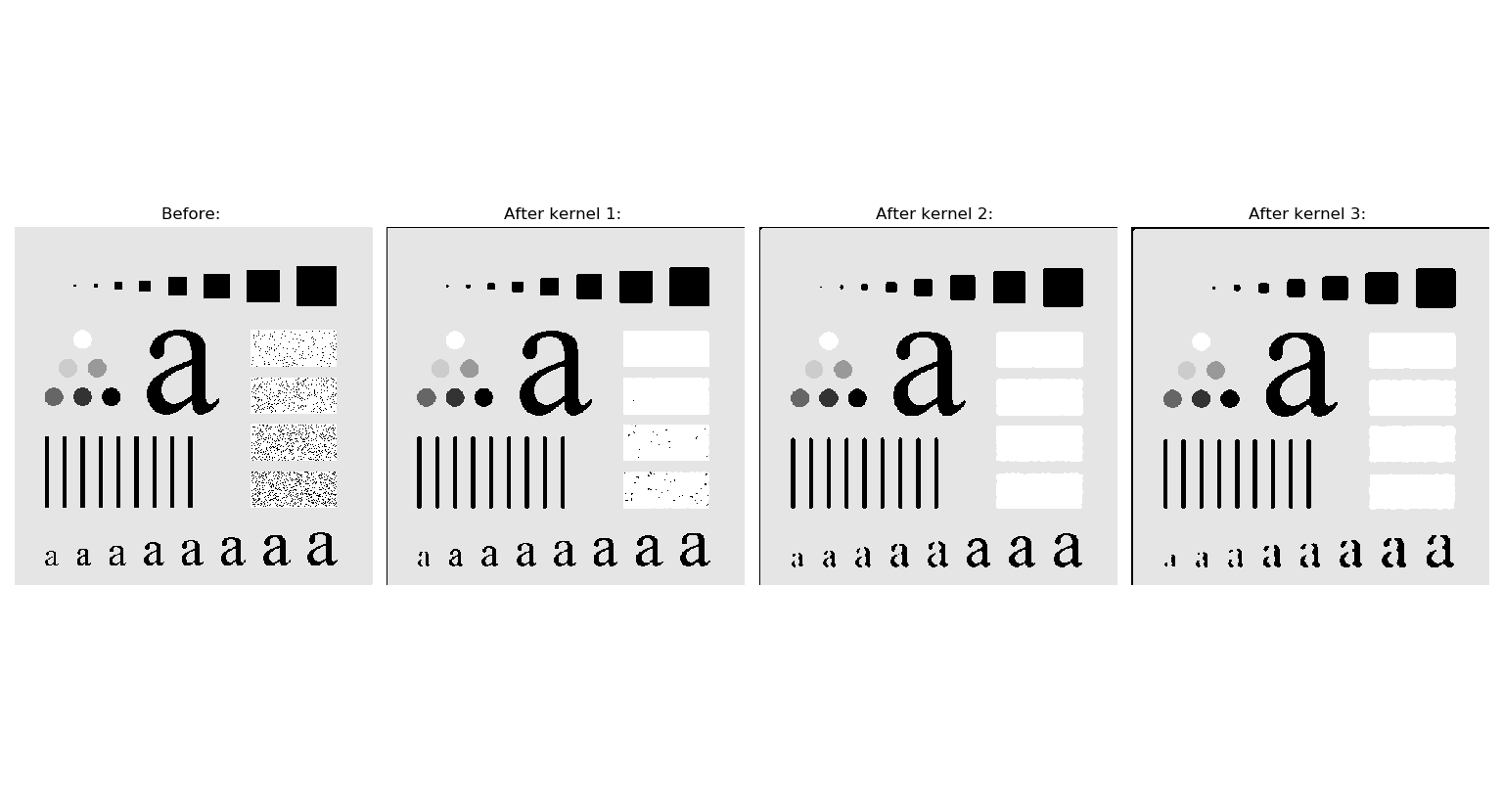


Figure : Median filter at various kernel sizes on image 333

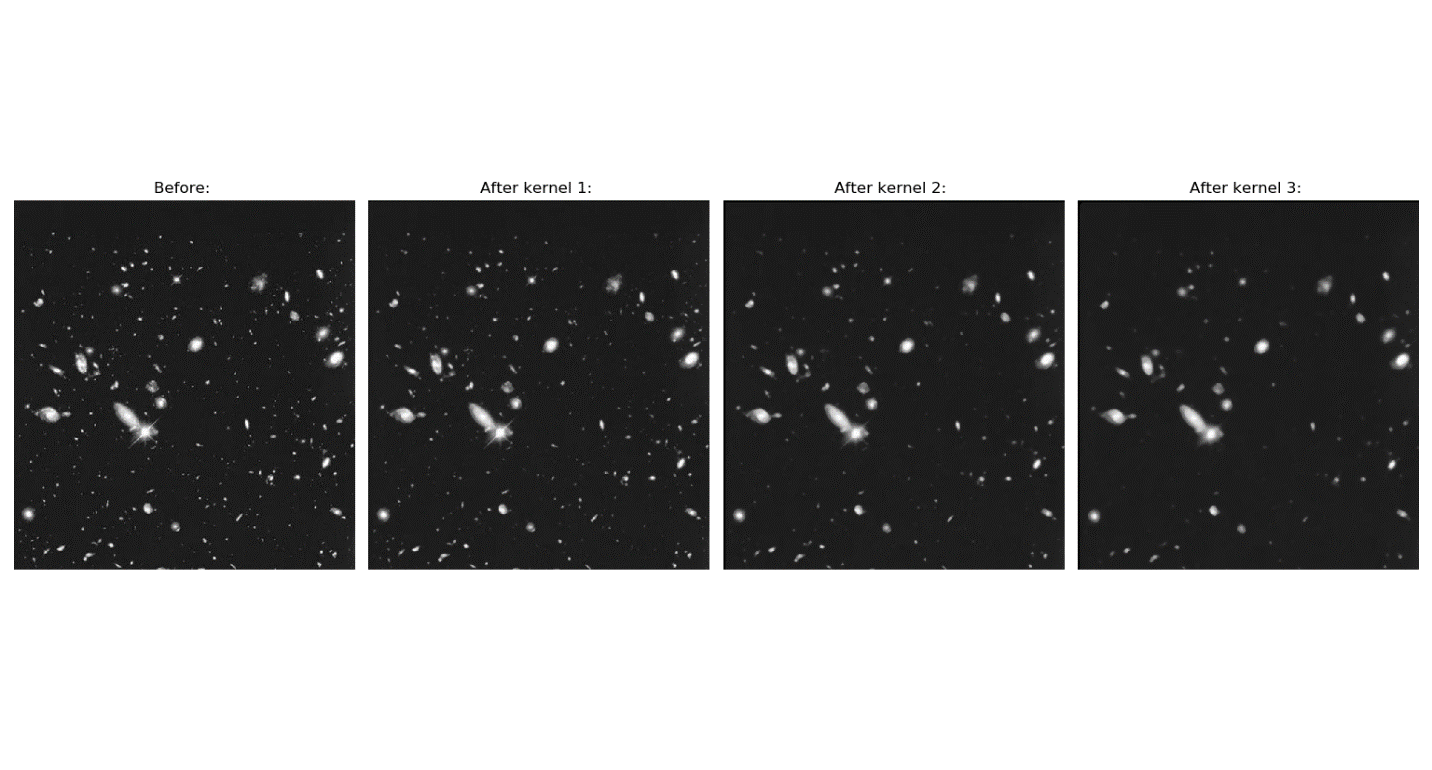


Figure : Median filter at various kernel sizes on image 334

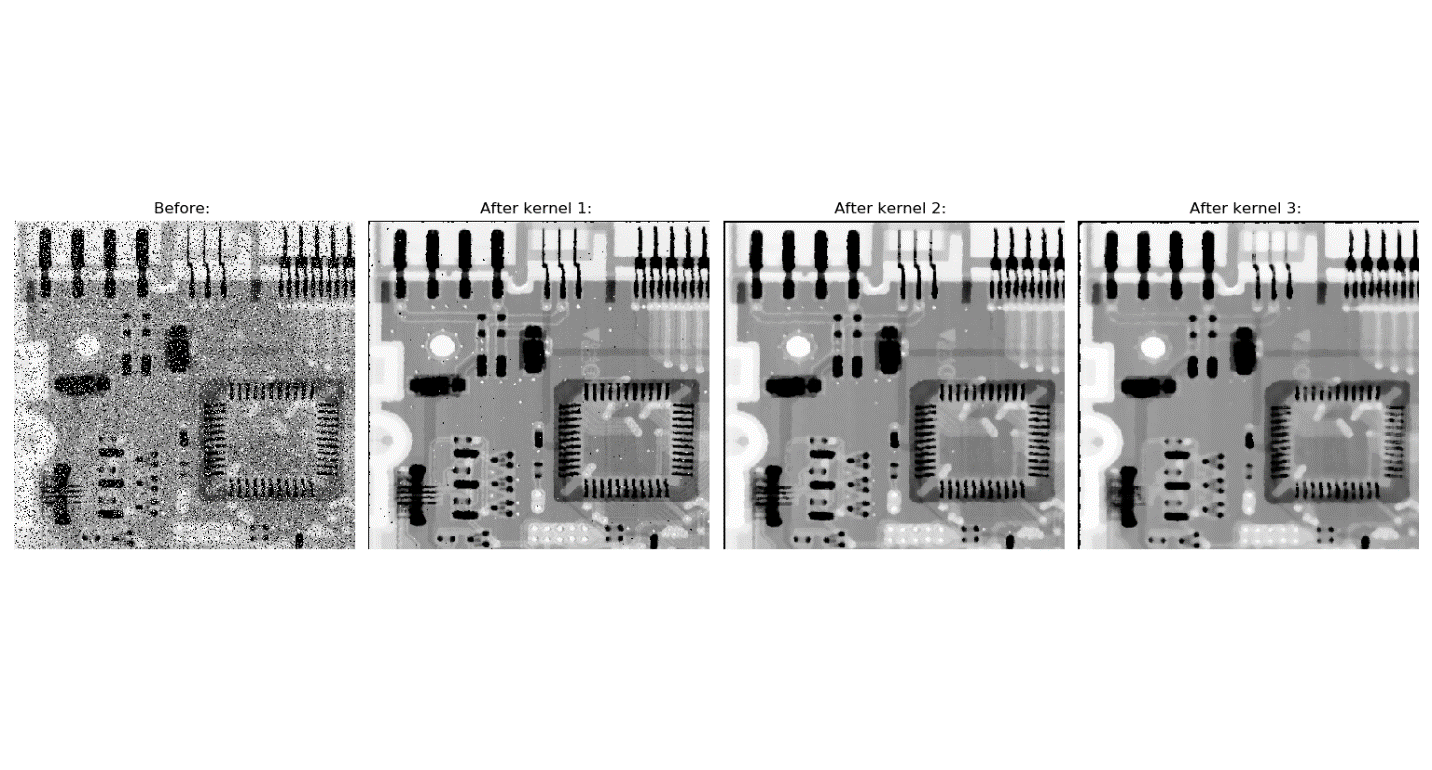


Figure : Median filter at various kernel sizes on image 335

The gaussian smoothing function is much the same. Like the previous two functions, it begins by preprocessing the image:

*im, im\_proc = prepare\_image(im, kernel, 'zero')*

Based on the input Boolean *premade*, it then obtains a mask by either calling a premade mask for the corresponding kernel size from the masks module or it creates a new one at runtime using the function *create\_gauss\_conv()* which takes in the desired kernel size and calculates an appropriate gaussian mask. The code for this function can be seen below:

*def create\_gauss\_conv(kernel: int) -> np.ndarray:*

*# create array to hold guassian array*

*final = np.zeros(shape=(2 \* kernel + 1, 2 \* kernel + 1))*

*tot = 0*

*# loop through array, creating gaussian distribution*

*for i in range(-kernel, kernel + 1):*

*for j in range(-kernel, kernel + 1):*

*# create guassian dividend*

*tmp = -1 \* ((i \*\* 2 + j \*\* 2) / (2 \* (kernel \*\* 2)))*

*# complete gaussian function and place it in dest*

*final[i + kernel, j + kernel] = math.exp(tmp) / (2 \* np.pi \* kernel \*\* 2)*

*# count total for normalization*

*tot = tot + final[i + kernel, j + kernel]*

*# normalize gaussian array*

*final = final / tot*

*return final*

Both the premade masks and the masks created on demand are already normalized, and therefore can be simply convolved with the image:

*im\_proc = scipy.signal.convolve2d(im, \_filter, 'valid')*

The processed image can then be run through the post processing algorithm and returned.

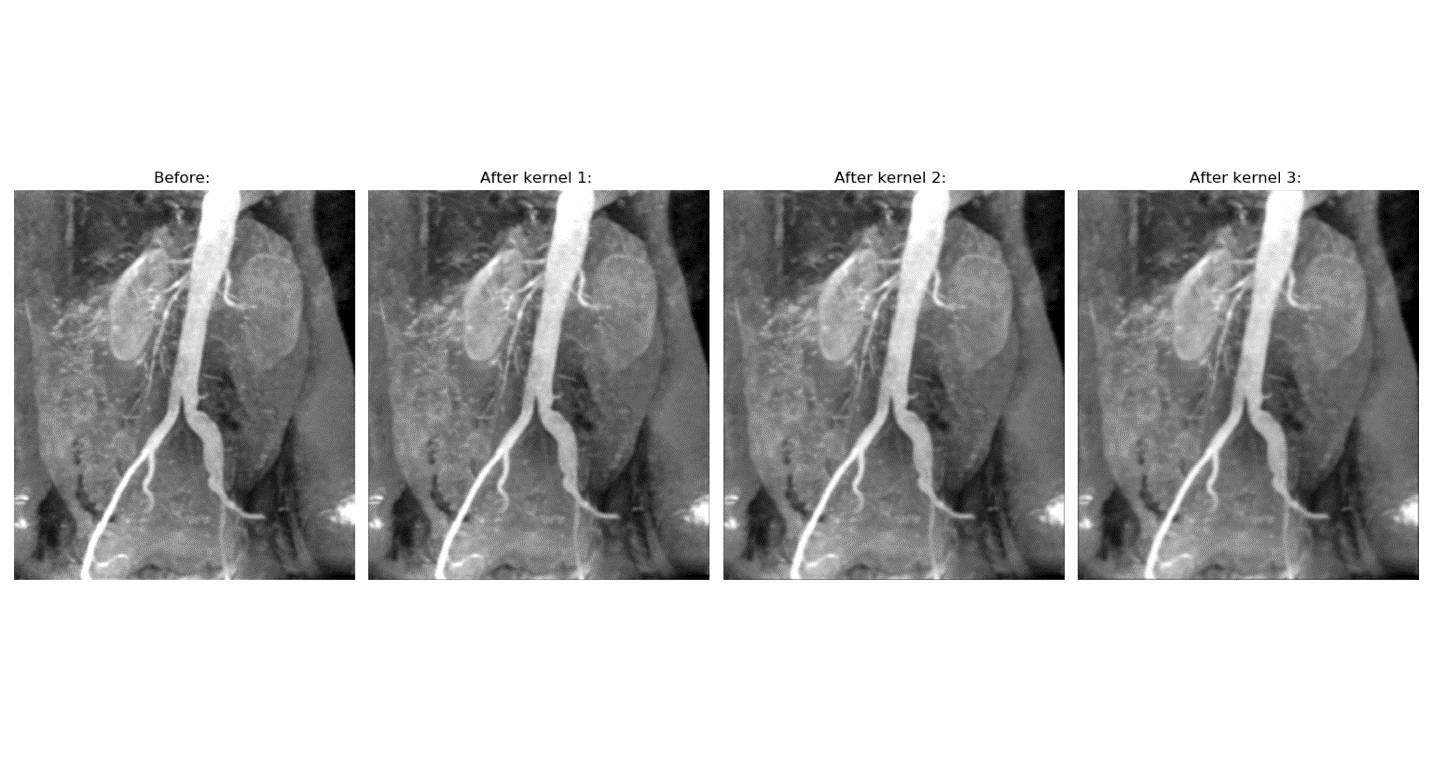


Figure : Gaussian blur at various kernel sizes on image 312

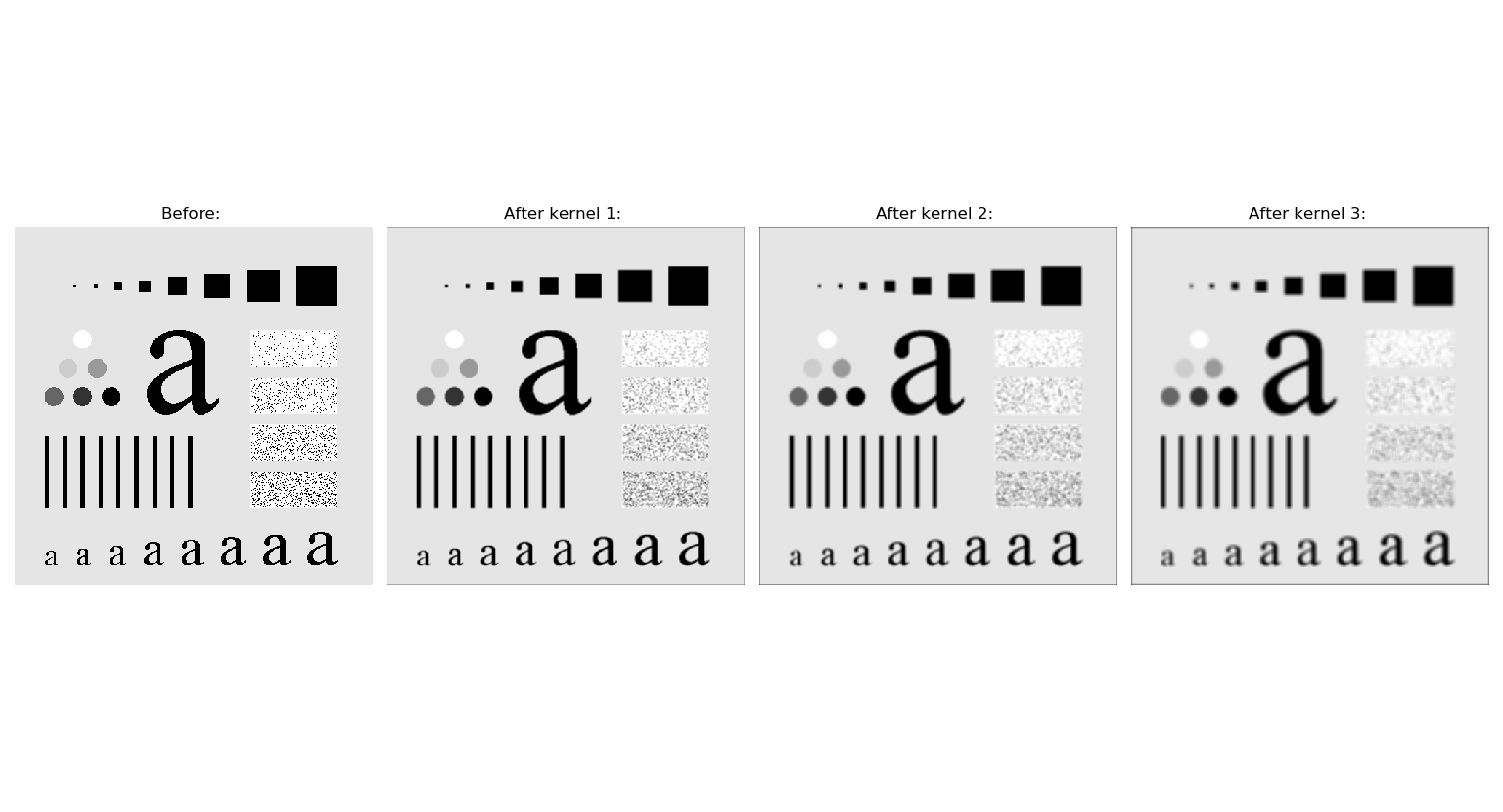


Figure : Gaussian blur at various kernel sizes on image 333

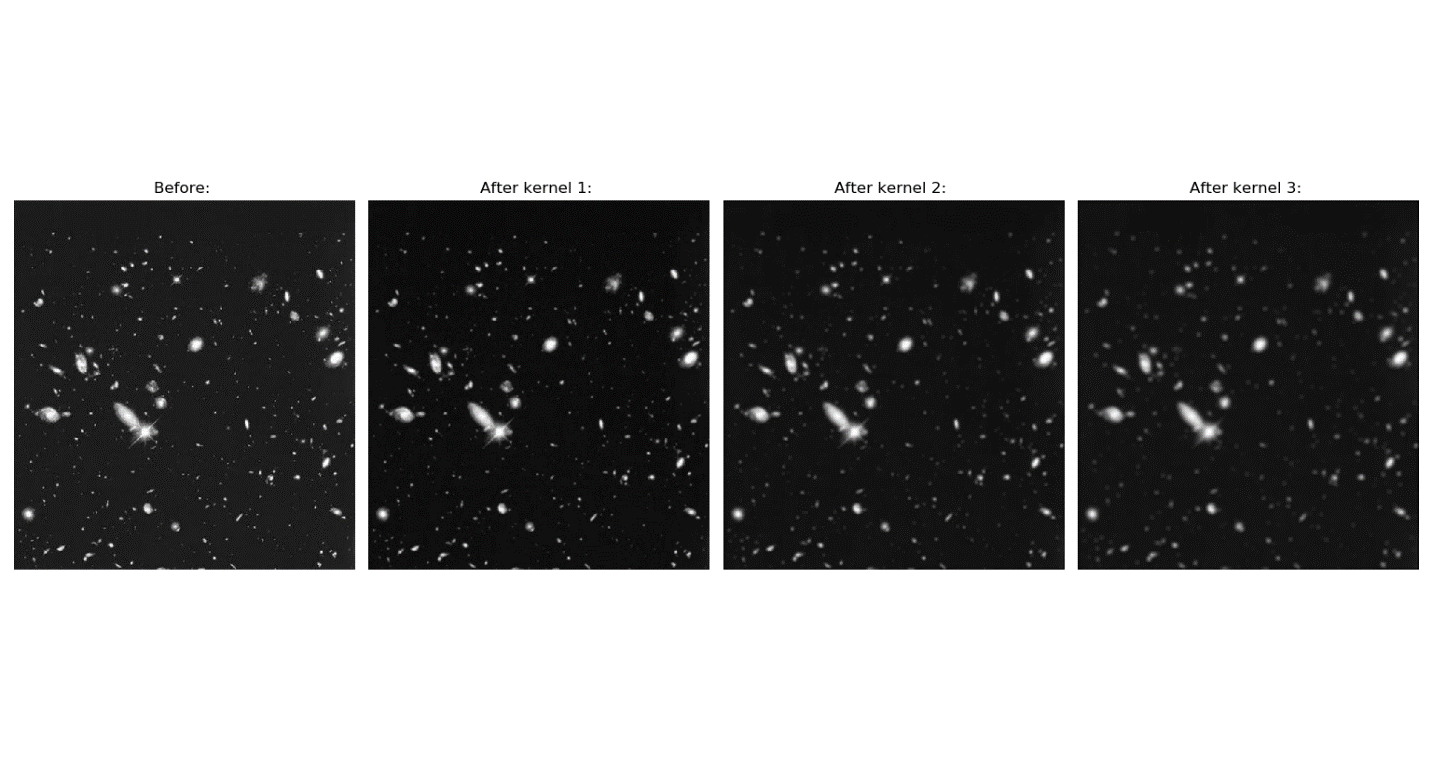


Figure : Gaussian blur at various kernel sizes on image 334

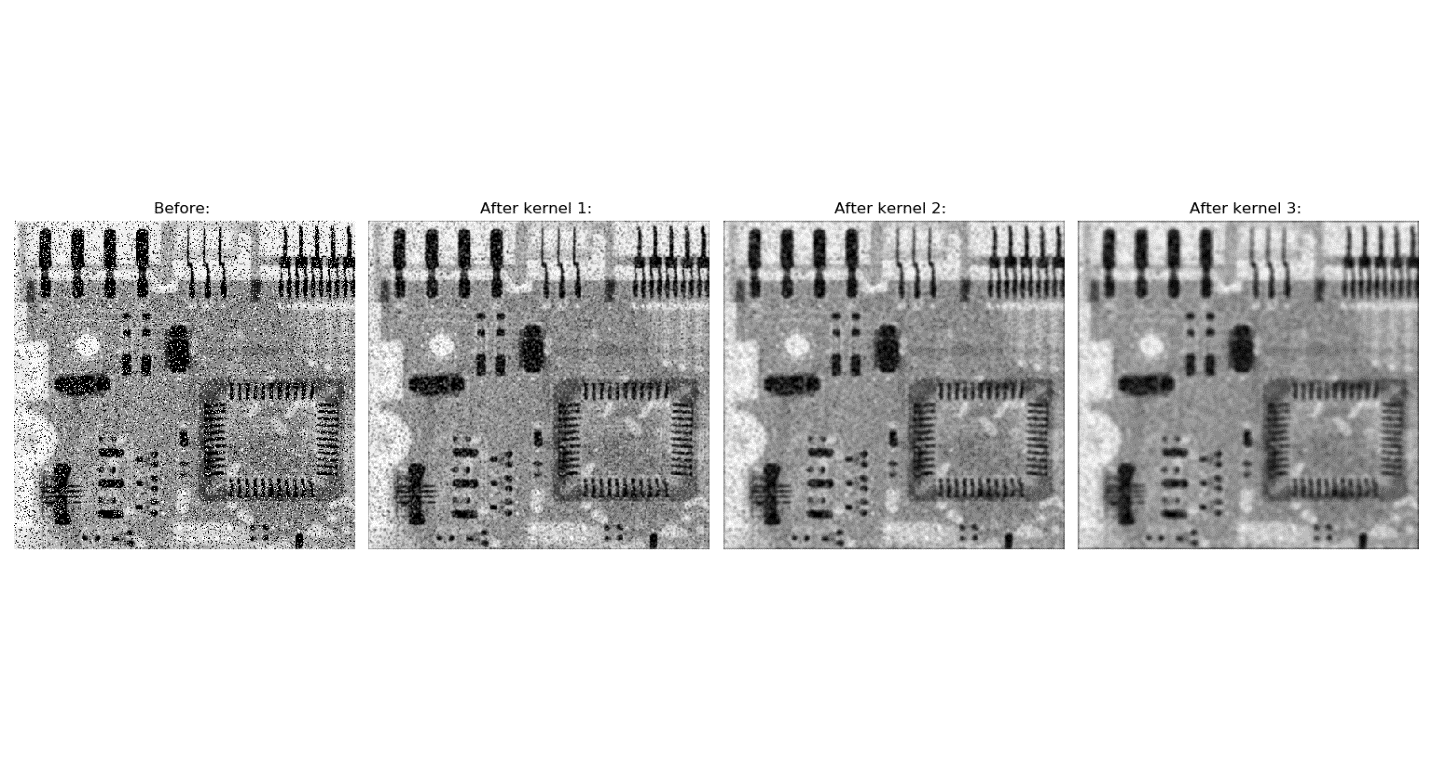


Figure : Gaussian blur at various kernel sizes on image 335

For part 2 of the assignment, a Laplacian sharpening algorithm was run on two different images. Found in the sharpening module, *lpl\_sharpen()* takes in an image to be sharpened and a mask which can be found in the *masks.py* module. As seen previously, the image is run through the preprocessing algorithm. However, an additional copy is also made in order to later display both the Laplacian sharpening output, as well as the final image with the output added back in e.g.:

*im\_final = im.copy()*

*im, im\_lpl = prepare\_image(im, padding, 'repeat')*

The image is then convolved with the specified mask, which in my case is [[-1,-1,-1],[-1,8,-1],[-1,-1,-1]] :

*im\_lpl = scipy.signal.convolve2d(im, array, 'valid')*

The type is then changed for upcoming normalization:

*im\_lpl = im\_lpl.astype('float64')*

Next the Laplacian sharpening output is normalized:

*im\_lpl -= np.min(im\_lpl)*

*im\_lpl /= np.max(im\_lpl)*

*im\_lpl \*= 255.0*

This will now be the image used as the sharpened output display. Next, the final image is created by adding the sharpened output back into the image, and then normalizing it:

*im\_final = im\_final + im\_lpl*

*im\_final -= np.min(im\_final)*

*im\_final /= np.max(im\_final)*

*im\_final \*= 255.0*

Finally, both the Laplacian output and the final image are postprocessed and returned.

Note: For display, a fourth image was created in addition to the original, the Laplacian sharpened image, and the original+sharpen image. This fourth image has been contrast stretched in order to fix “graying” present in the final image with comparison to the original.

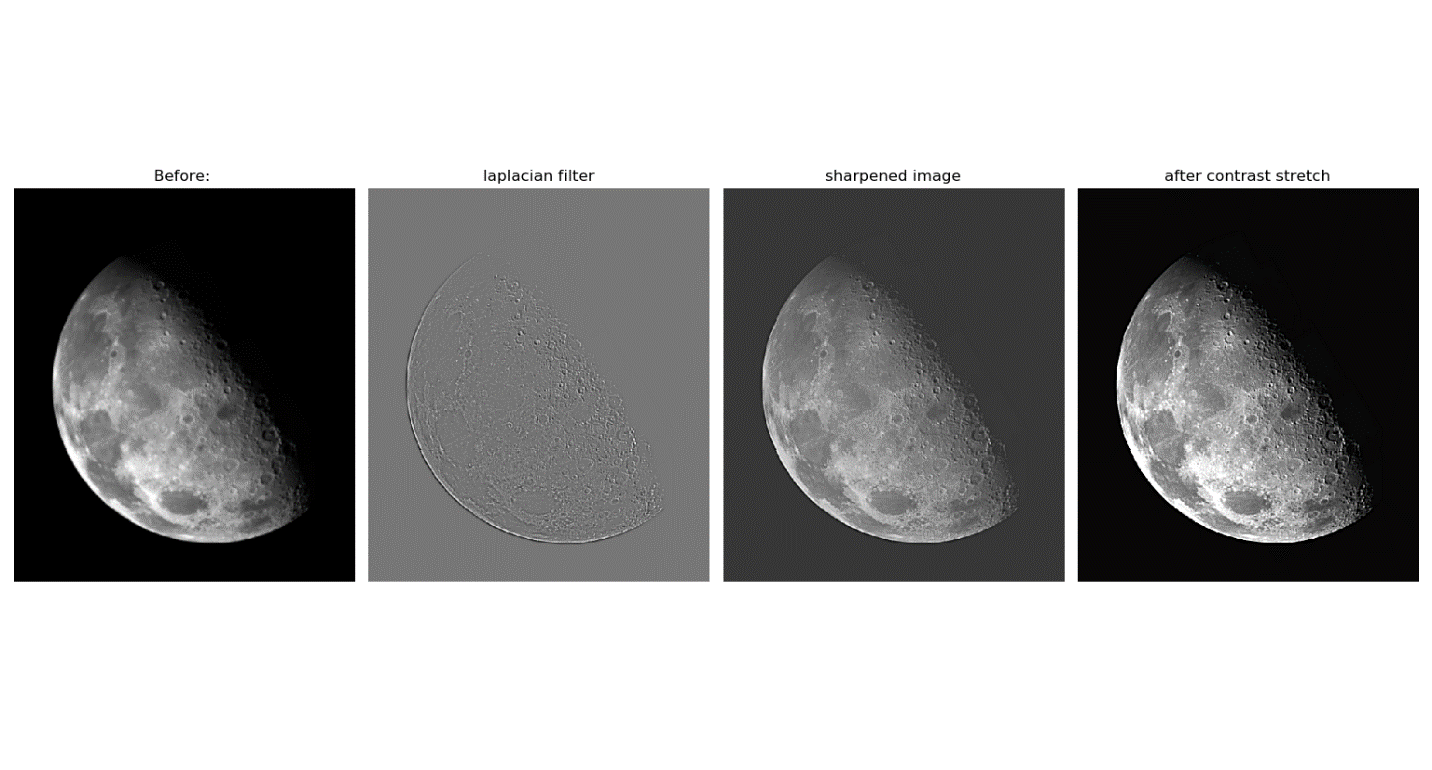


Figure : Steps of laplacian sharpen on image 338

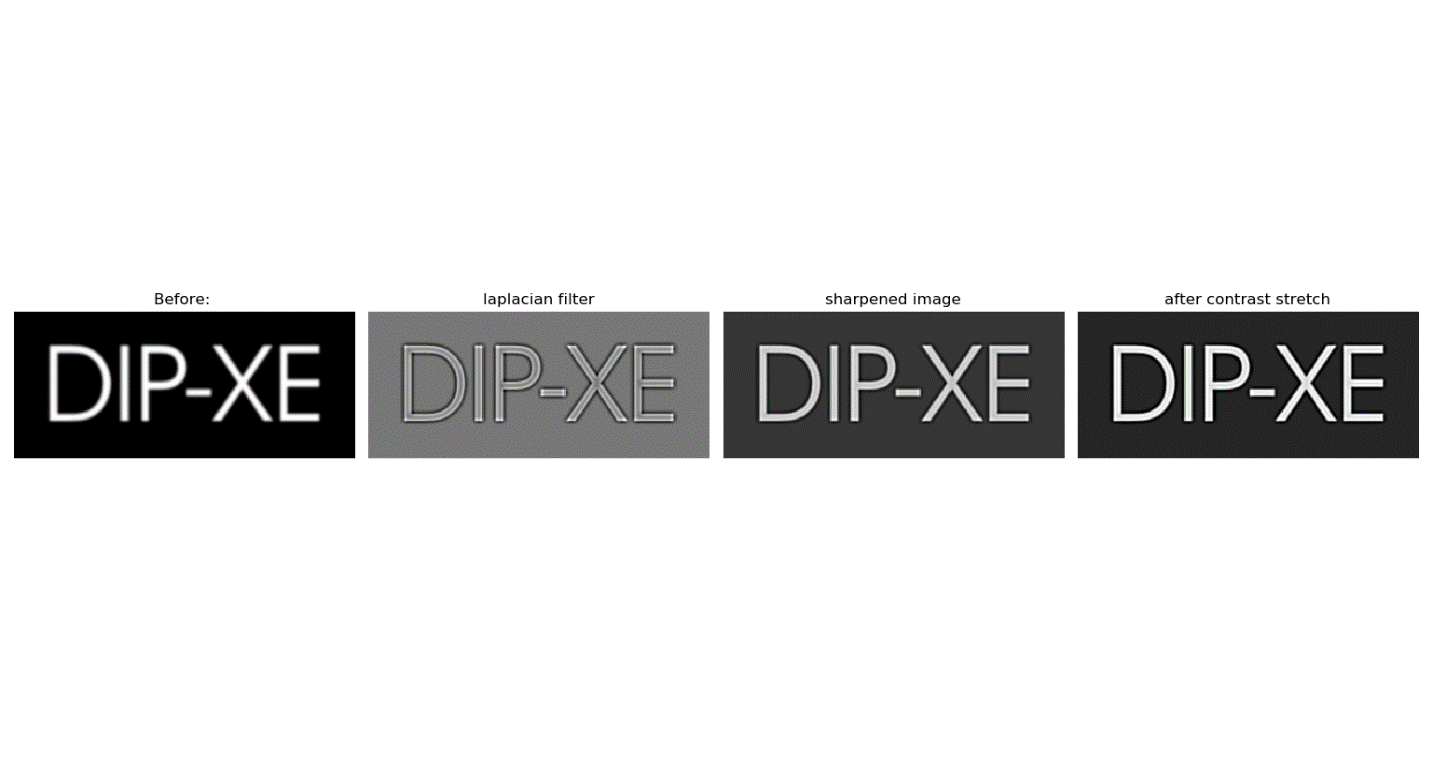


Figure : Steps of laplacian sharpen on image 340

For the final part of the assignment, a gaussian pyramid must be constructed and displayed. This consists of three main functions: *guass\_pyramid\_resize(), get\_gaussian\_pyramid(),* and *display\_guass\_pyramid().* To start with, the *get\_gaussian\_pyramid()* function takes in an image and a specified number of levels. Working recursively, it calls *guass\_pyramid\_resize()* according to the given number of levels and returns all the processed images and the original image as a list. The code for this function can be seen here:

*def get\_gaussian\_pyramid(im: np.ndarray, levels: int) -> List[np.ndarray]:*

*if levels > 7:*

*levels = 7*

*if levels <= 1: # hit end of recursive or no levels*

*return [im, ]*

*x = get\_gaussian\_pyramid(im, levels-1)*

*x.append(guass\_pyramid\_resize(x[-1]))*

*return x*

Next, the *guass\_pyramid\_resize()* function resizes a given image for use in a gaussian pyramid. To do so, it first runs the given image through a gaussian filter of kernel size one e.g.:

*im = smoothing.guass\_filter(im, 1, False)*

It then creates a new image with half the dimensions size of the input image:

*dimensions = im.shape*

*im\_ds = np.zeros((dimensions[0]//2, dimensions[1]//2))*

Next, it loops through the new image and samples every other pixel from the smoothed input image:

*for i in range(0, dimensions[0]):*

*for j in range(0, dimensions[1]):*

*im\_ds[i, j] = im[2 \* i, 2 \* j]*

Finally, it postprocesses the down sampled image and returns it.

The *display\_guass\_pyramid()* function focuses on formatting and displaying the gaussian pyramid. It takes in a list of images as input, which consists of the original image and its down sampled copies. It then loops through this list and places the images in accordance with the example given. Additionally, the skimage.transform library is utilized to change all the images to the same size for placement at the top of the figure in order to make it match the example perfectly. The final result can be seen below:



Figure : Gaussian pyramid on "lenna" image

Overall, I found my results to be quite satisfactory. There were a few notable issues I ran into, however. One of these is that my median smoothing algorithm is quite inefficient due to running in O(n4) time. This is obviously less than ideal, however in the time I had I was unable to create a faster working example. I would still like to improve upon it, and I have several possible ideas for how to do so. An additional issue I found was that the final image from my sharpening algorithm induced a gray cast on the black background of the moon picture. I corrected this by stretching the contrast using the *exposure.rescale\_intensity()* function from skimage. One final problem I ran into was python’s dynamic typing, which at times led to me mixing types accidentally. This required me to keep closer track of return and initialization types as well as led to the type-setting in the pre and post processing functions.