Michael Christensen
October 2, 2013
CS 450

Homework 3

1.
I did correlation here:
Signal: [1 3 4 2 1 3 2 1 3 1]
Mask/Kernel: [1 1 1]
Signal padded with 0's: [**0** 1 3 4 2 1 3 2 1 3 1 **0**]
Output = [[0 1 3] · [1 1 1], [1 3 4] · [1 1 1], [3 4 2] · [1 1 1], [4 2 1] · [1 1 1], [2 1 3] · [1 1 1], [1 3 2] · [1 1 1], [3 2 1] · [1 1 1], [2 1 3] · [1 1 1], [1 3 1] · [1 1 1], [3 1 0] · [1 1 1]]
= [4 8 9 7 6 6 6 6 5 4]
I handled the boundaries of this finite-length signal by padding it with zeros on both sides (which effectively just ignored the weight value for elements that go past the beginning or end).

2.
I did correlation here:
Signal: [1 3 4 2 1 3 2 1 3 1]
Mask/Kernel: [1 2 1]
Signal Padded with 0's [**0** 1 3 4 2 1 3 2 1 3 1 **0**]
Output =[[0 1 3] · [1 2 1], [1 3 4] · [1 2 1], [3 4 2] · [1 2 1], [4 2 1] · [1 2 1], [2 1 3] · [1 2 1], [1 3 2] · [1 2 1], [3 2 1] · [1 2 1], [2 1 3] · [1 2 1], [1 3 1] · [1 2 1], [3 1 0] · [1 2 1]]
= [5 11 13 9 7 9 8 7 8 5]

3.
I did convolution here (which I assumed meant spatial filtering with the impulse response/kernel flipped)
Signal: [1 3 4 2 1 3 2 1 3 1]
Impulse Response: [1 2 3]
Signal Padded with 0's: [**0 0** 1 3 4 2 1 3 2 1 3 1 **0 0**]
Output = [[0 0 1] · [3 2 1], [0 1 3] · [3 2 1], [1 3 4] · [3 2 1], [3 4 2] · [3 2 1], [4 2 1] · [3 2 1], [2 1 3] · [3 2 1], [1 3 2] · [3 2 1], [3 2 1] · [3 2 1], [2 1 3] · [3 2 1], [1 3 1] · [3 2 1], [3 1 0] · [3 2 1], [1 0 0] ·[3 2 1]]
= [1 5 13 19 17 11 11 14 11 10 11 3]

4.
Image: [1 3 4 4 1]
[1 2 5 6 3]
[1 3 2 3 2]
[2 2 1 2 1]
[1 3 2 1 1]
Mask: [1 1 1]
[1 2 1]
[1 1 1]
Correlation: (I handle the edges by duplicating the edge values once with 0's in the corner spaces, so the resulting image is:)
[**0 1 3 4 4 1 0**]
1 [**1** 1 3 4 4 1 **1**]
2 [**1** 1 2 5 6 3 **3**]
3 [**1** 1 3 2 3 2 **2**]

4 [**2** 2 2 1 2 1 **2**]
5 [**1** 1 3 2 1 1 **1**]
[**0 1 3 2 1 1 0**]

Row 1: [[0 1 3] · [1 1 1] + [1 1 3] · [1 2 1] + [1 1 2] · [1 1 1],
[1 3 4] · [1 1 1] + [1 3 4] · [1 2 1] + [1 2 5] · [1 1 1],
[3 4 4] · [1 1 1] + [3 4 4] · [1 2 1] + [2 5 6] · [1 1 1],
[4 4 1] · [1 1 1] + [4 4 1] · [1 2 1] + [5 6 3] · [1 1 1],
[4 1 0] · [1 1 1] + [4 1 1] · [1 2 1] + [6 3 3] · [1 1 1]] = [14 27 39 36 24]

Row 2: [[1 1 3] · [1 1 1] + [1 1 2] · [1 2 1] + [1 1 3] · [1 1 1],
[1 3 4] · [1 1 1] + [1 2 5] · [1 2 1] + [1 3 2] · [1 1 1],
[3 4 4] · [1 1 1] + [2 5 6] · [1 2 1] + [3 2 3] · [1 1 1],
[4 4 1] · [1 1 1] + [5 6 3] · [1 2 1] + [2 3 2] · [1 1 1],
[4 1 1] · [1 1 1] + [6 3 3] · [1 2 1] + [3 2 2] · [1 1 1]] = [15 24 37 36 28]

Row 3: [[1 1 2] · [1 1 1] + [1 1 3] · [1 2 1] + [2 2 2] · [1 1 1],
[1 2 5] · [1 1 1] + [1 3 2] · [1 2 1] + [2 2 1] · [1 1 1],
[2 5 6] · [1 1 1] + [3 2 3] · [1 2 1] + [2 1 2] · [1 1 1],
[5 6 3] · [1 1 1] + [2 3 2] · [1 2 1] + [1 2 1] · [1 1 1],
[6 3 3] · [1 1 1] + [3 2 2] · [1 2 1] + [2 1 2] · [1 1 1]] = [16 23 28 28 26]

Row 4: [[1 1 3] · [1 1 1] + [2 2 2] · [1 2 1] + [1 1 3] · [1 1 1],
[1 3 2] · [1 1 1] + [2 2 1] · [1 2 1] + [1 3 2] · [1 1 1],
[3 2 3] · [1 1 1] + [2 1 2] · [1 2 1] + [3 2 1] · [1 1 1],
[2 3 2] · [1 1 1] + [1 2 1] · [1 2 1] + [2 1 1] · [1 1 1],
[3 2 2] · [1 1 1] + [2 1 2] · [1 2 1] + [1 1 1] · [1 1 1]] = [18 19 20 17 16]

Row 5: [[2 2 2] · [1 1 1] + [1 1 3] · [1 2 1] + [0 1 3] · [1 1 1],
[2 2 1] · [1 1 1] + [1 3 2] · [1 2 1] + [1 3 2] · [1 1 1],
[2 1 2] · [1 1 1] + [3 2 1] · [1 2 1] + [3 2 1] · [1 1 1],
[1 2 1] · [1 1 1] + [2 1 1] · [1 2 1] + [2 1 1] · [1 1 1],
[2 1 2] · [1 1 1] + [1 1 1] · [1 2 1] + [1 1 0] · [1 1 1]] = [16 20 19 13 11]

Output:
[14 27 39 36 24]
[15 24 37 36 28]
[16 23 28 28 26]
[18 19 20 17 16]
[16 20 19 13 11]

5. G&W 3.19b

Example image to illustrate:

1 2 3 4 5
1 [1 3 4 4 1]
2 [1 2 5 6 3]
3 [1 3 2 3 2]
4 [2 2 1 2 1]
5 [1 3 2 1 1]

Median of (2, 2) with a 3x3 neighborhood = median[1 1 1 2 **2** 3 3 4 5] = 2

Median of (2, 3) = median[2 2 3 3 **3** 4 4 5 6] = 3

Going along row 2 from column 2 → 3, 3 values in the set change. If we start with an ordered set *A* of 9 numbers from the neighborhood at (2, 2) and move to (2, 3), the subset [1 1 1] is eliminated and the subset [4 6 3] is added. Since the set *A* – [1 1 1] is ordered still, adding [4 6 3] to *A* – [1 1 1] is a linear operation *O(n)*, which is must better than selecting (O(*n*)) and sorting (O(*n log n*) best case). The best way to know which numbers to remove from the neighborhood set as you slide the median-filtering window across the image horizontally (vertically) is to maintain each column (row) of *n* digits you can iterate over in the ordered list you maintain (made up those *n* columns (rows)), removing the three digits in *A* that match the ones in the column (row) you are removing. This is also a O(*n*) operation, so removing the old column(row) and adding the new one to the ordered set of *n* x *n* neighbors is a O(2*n*) = O(*n*) operation still. The key in this approach is to maintain a list of ordered neighbors, removing the ones from the left-most column and adding the new column as you go from left-to-right (or rows, equally, if going from top to bottom).

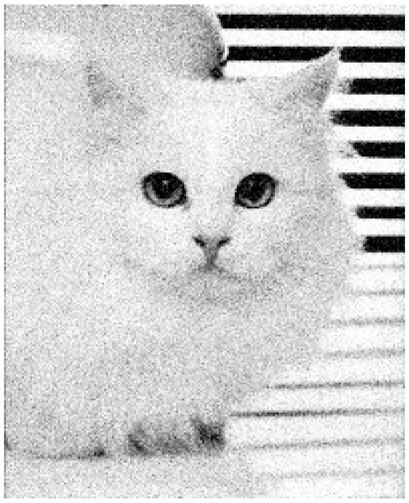
I turned the written part of the assignment in via Learning Suite on 10/2 before midnight.

Programming Exercises:

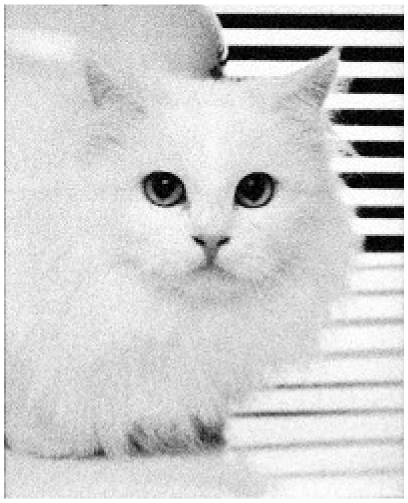
A. Arithmetic Operations:

2.

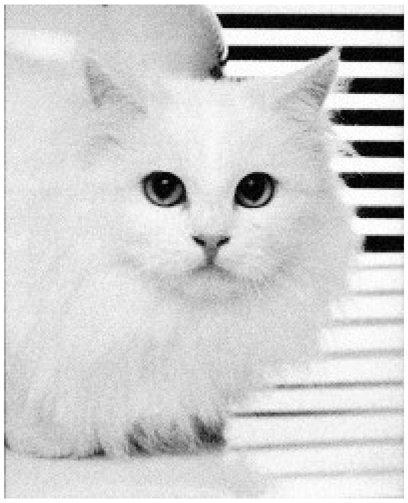
Average 2:



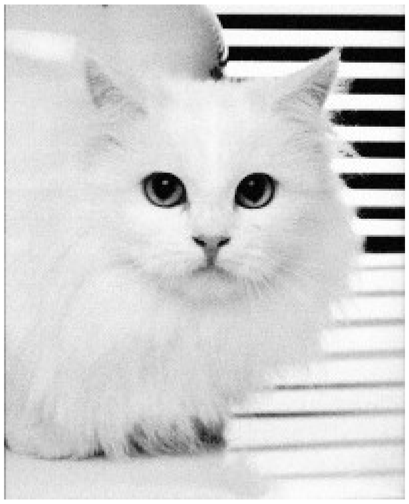
Average 5:



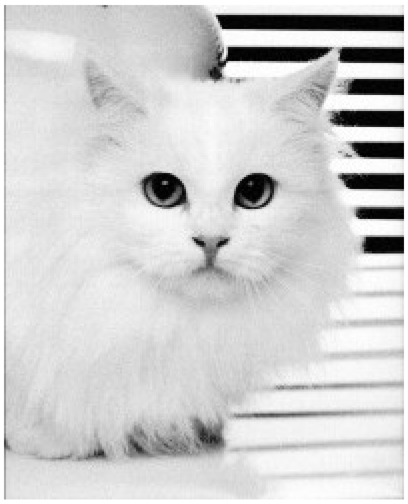
Average 10:



Average 20:

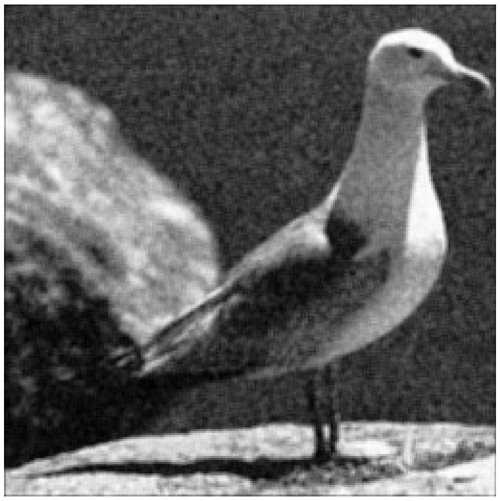
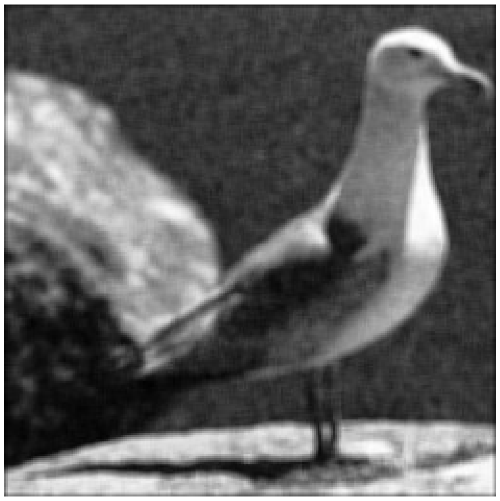
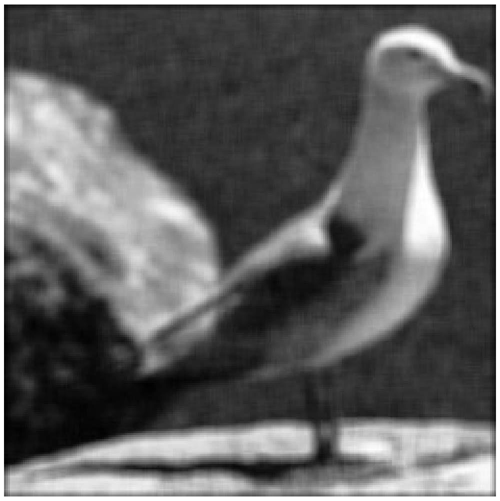
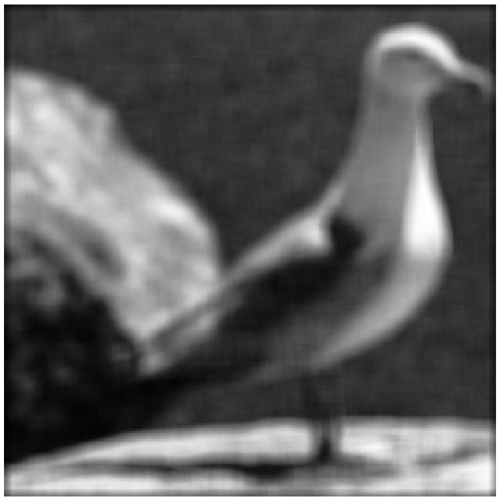


Average 40:



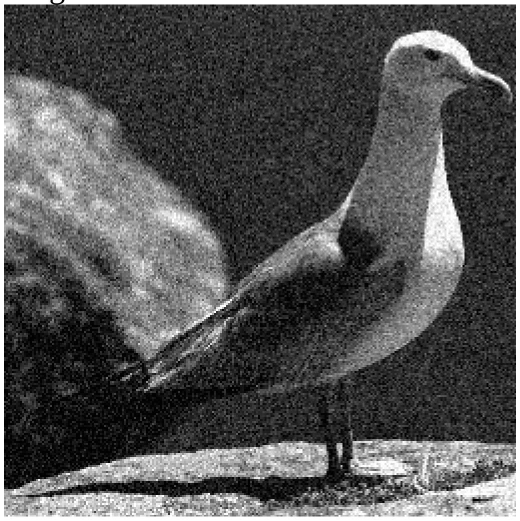
This works because each image is identical, roughly, except for the small random variations produced by noise. We therefore have 2-40 copies of each pixel (depending on the number of the images we're using) and since the probability of each pixel being drastically different from the actual value of the object at that pixel is low, we get a clean idea of what the pixel is. The number of images that are enough is a subjective measurement, but here with this image I would say that around 20 images is enough. 10 is the point where you can see significant improvements in cleanliness, but 20 images is where the image looks nearly flawless, especially on the lighter parts of the cat's fur.

B. Spatial Filtering:
1A.
Uniform averaging on *noisygull.png*:

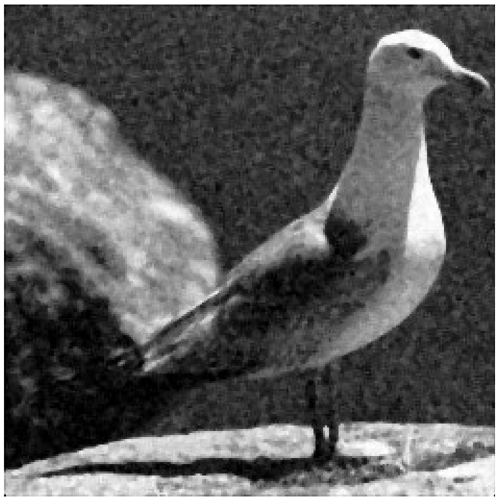
3x3:

5x5:

7x7:

9x9:


As expected, the box filter/uniform averaging here blurs the image, becoming more blurry the larger the kernel size because more surrounding neighbors contribute to each pixel's new value. Edges are blurred, but a possible positive outcome is that noise in the background is diminished, which looked very noisy to begin with. Overall, this method of filtering doesn't seem very suited to this image as it removes a great deal of the details (the original image is on the next page). Something noticeable is that although the edges are maintained using uniform averaging, they become blurred into a mix of the colors on each side of the edge, creating a wide gray band, for example, along the bird's neck.

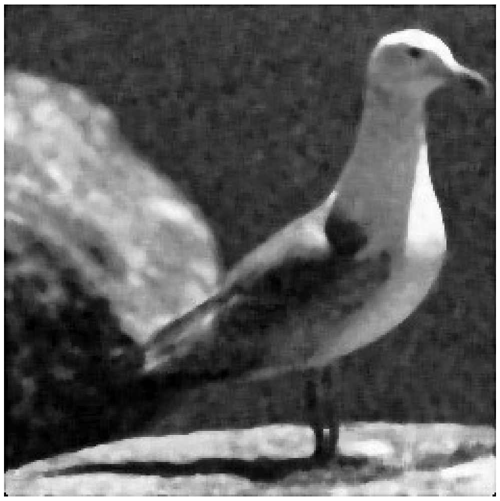
1B.
Median filtering on *noisygull.png*:
Original:



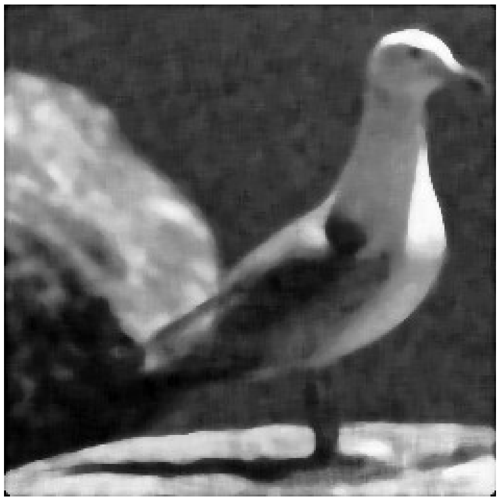
3x3:



5x5:

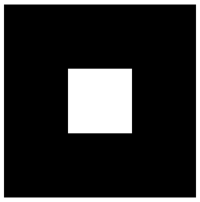


7x7:



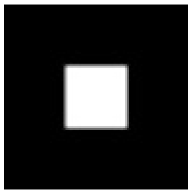
Median filtering, like averaging, increases blurriness as the kernel's dimensions increase. However, the edges are maintained very well (compare 7x7's neck-to-background edge to averaging 7x7's respective part). Unlike averaging, there is no wide gray band along the edges of the bird and background, and details like eye or separation of the legs is maintained up to and including the 7x7 image.

2. Original *whitebox.png*:

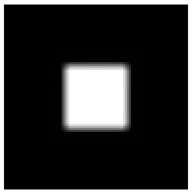


2A. Uniform Averaging:

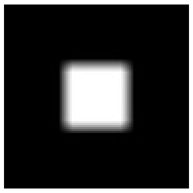
3x3:



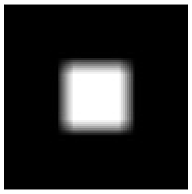
5x5:



7x7:

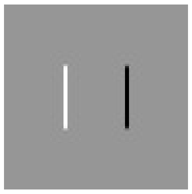


9x9:

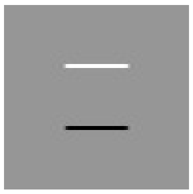


2B. Sobel Kernels:

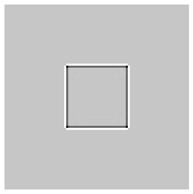
Derivative in x:



Derivative in y:



2C. Laplacian Kernel:

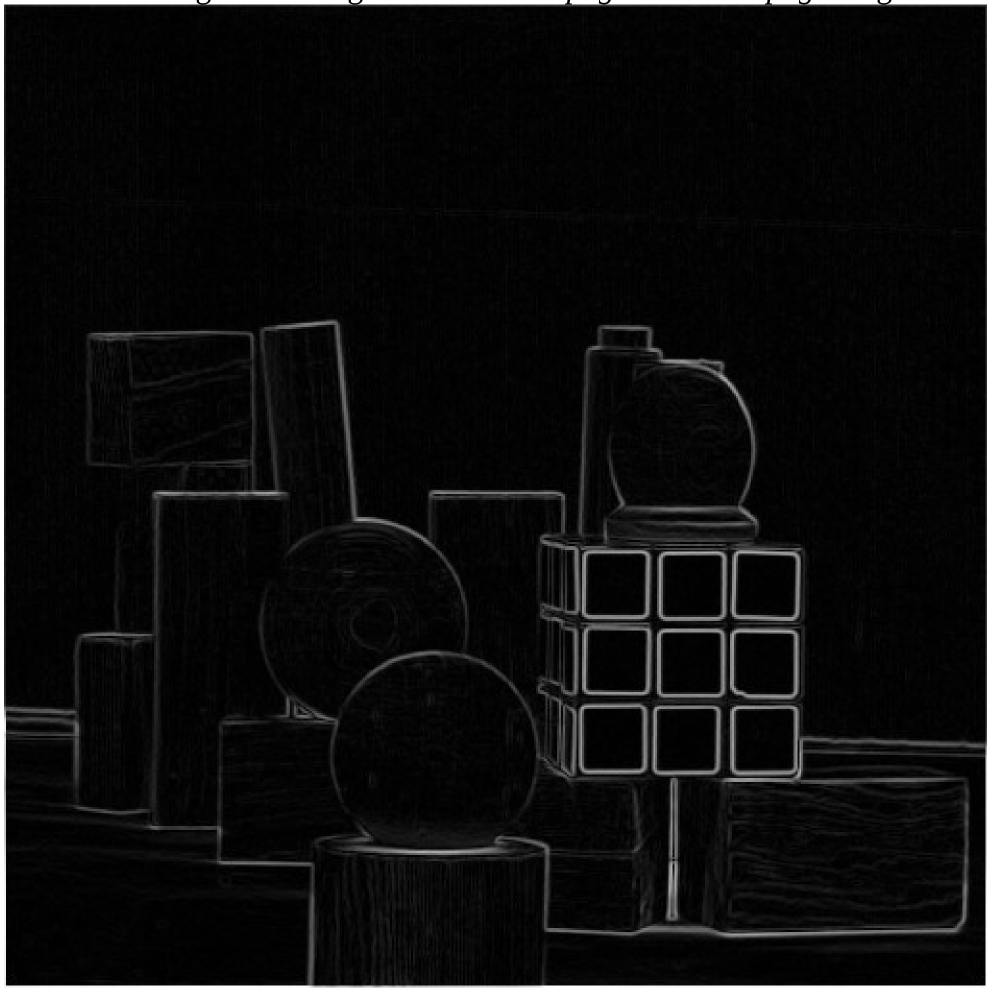


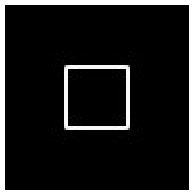
Uniform averaging in 2A blurred the image's edges as the number of dimensions in the box filters increased, similar to the noisy gull's edges. As expected, the rest of the image remained identical (the non-edge parts) since each pixel was equal to the average of its neighbors.

The Sobel kernel is effective at showing edges in one particular direction. In the first image, there is a vertical white bar on the left because the kernel takes the first derivative (in the x-direction here), and white(1.0) + black(0.0) equals 2*1.0 +(-2)*0.0 = 1.0 (white), normalized. Similarly, on the right there is a black bar because black(0.0) * 2 +white(1.0)*-2 = -2.0 = 0.0 (black, since -2 is below 0).

The Laplacian shows both edges because it takes the second derivatives in both dimensions, showing where the image goes from one color to another sharply.

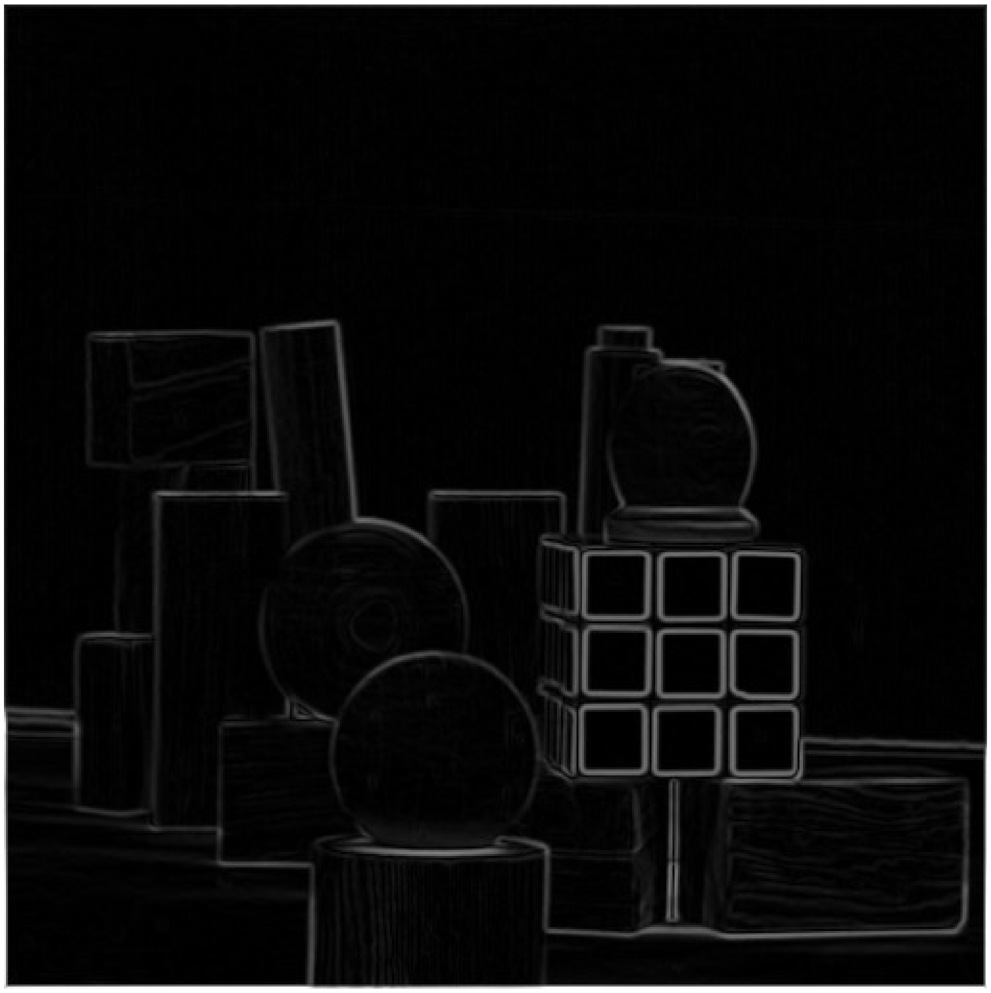
3. Gradient magnitude images for *whitebox.png* and *blocks.png* using Sobel kernels:



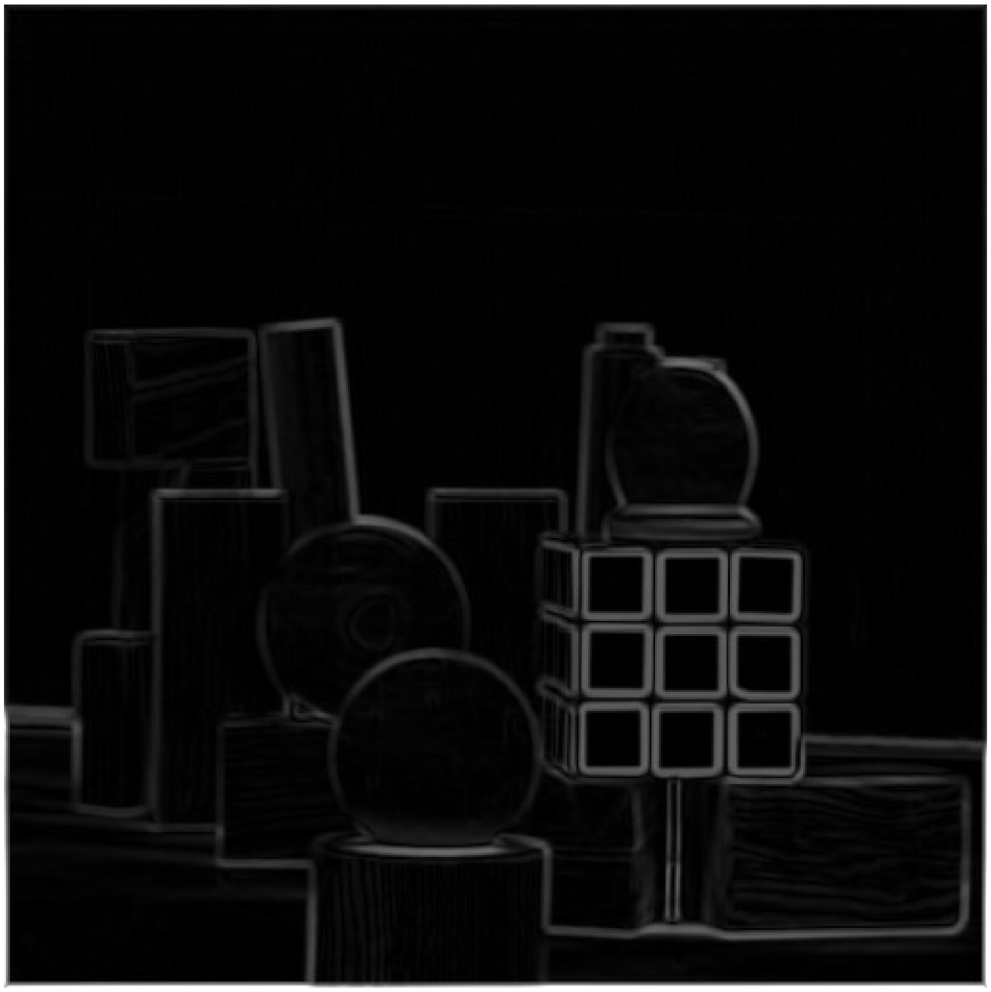


4. Gradient magnitude images for *blocks.png*, blurring the image first by a uniform kernel of various sizes:

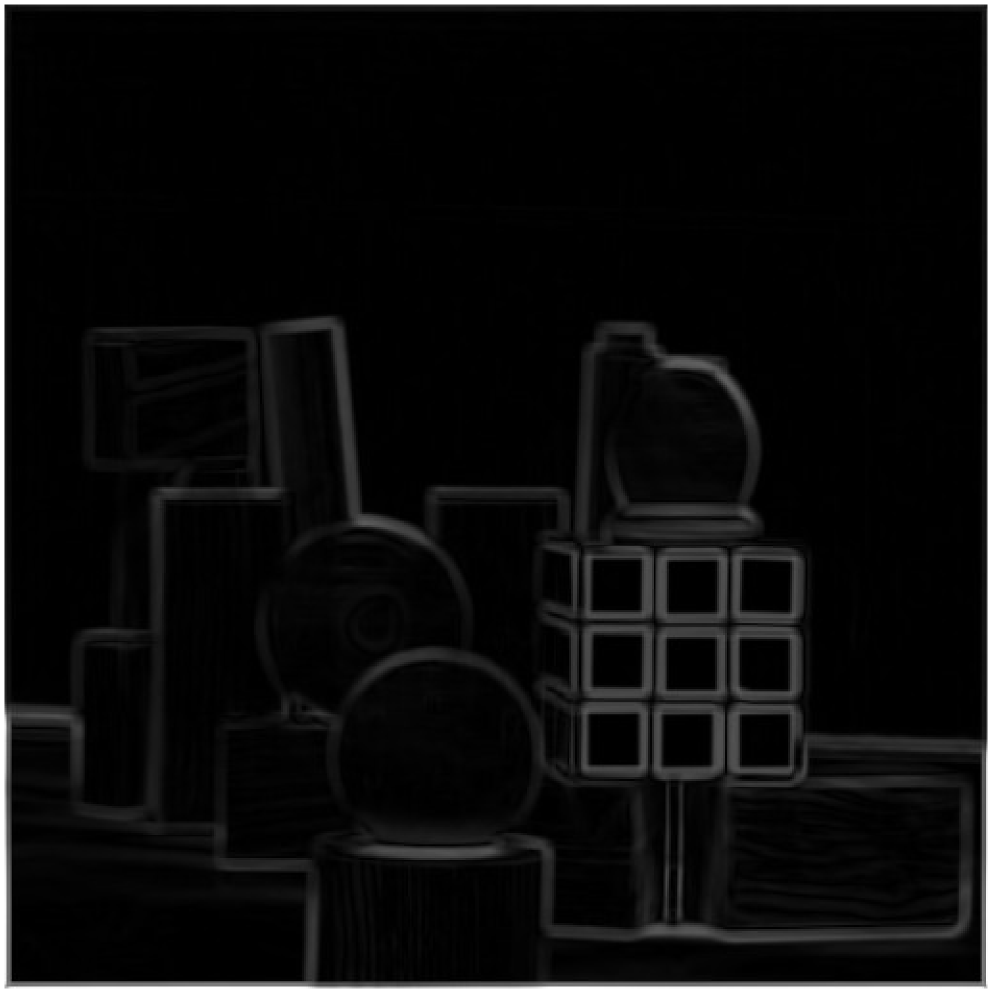
3x3:



5x5:

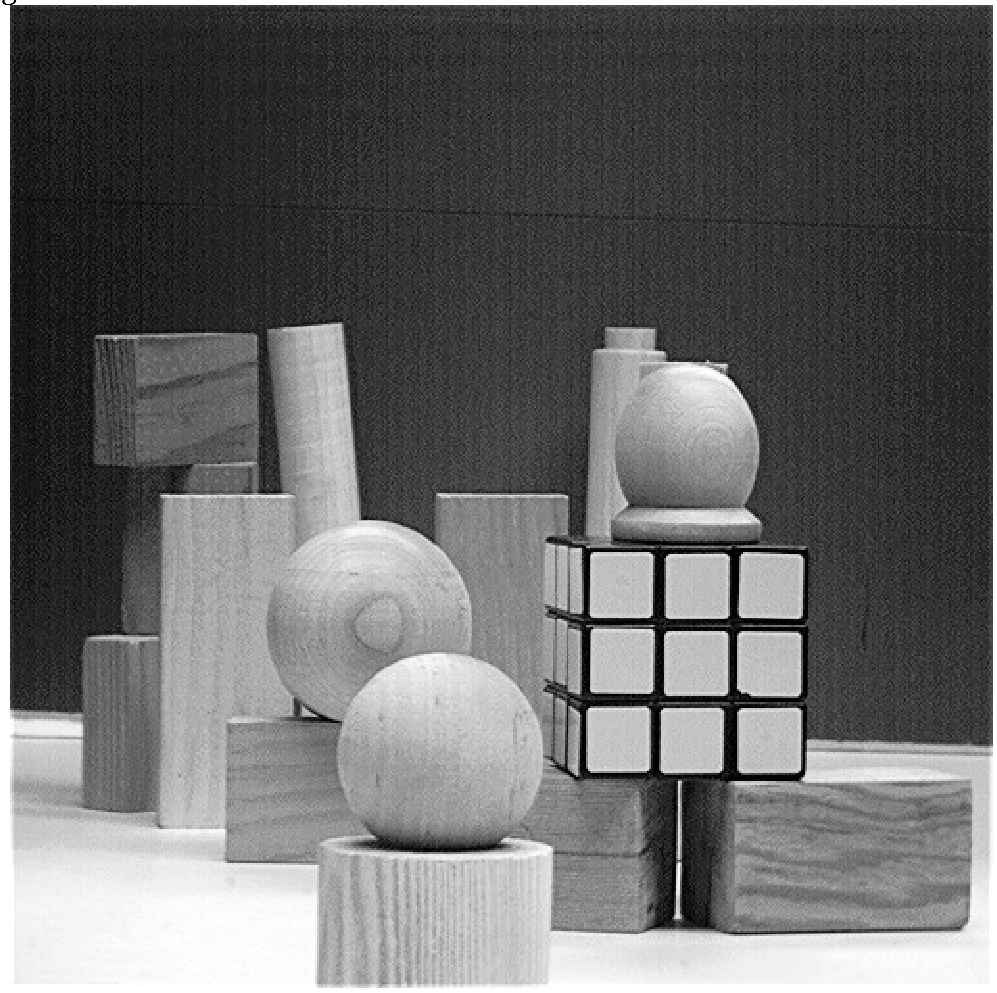


7x7:



The images get progressively blurred by the use of larger averaging kernels, but the edge detection ability isn't affected (the same edges in the crisp picture still maintained in the more blurry 7x7 picture), apart from how the edges are wider in the more blurrier images.

5. Unsharp masking to sharpen *blocks.png*:
The biggest trend I noticed was that larger radius sizes had more effect in making the image more grainy when the strength was small. For example the image with strength = 4 and radius = 3 was roughly equivalent to the image with strength strength = 6 and radius = 5. In other words, the greater the strength value, the less effect the increased kernel radius had in making the image harsher. However, this harshness was beneficial in that it made the edges “pop”/distinguish themselves more. The best image that balanced sharpening and maintaining the quality of the original image was the image with strength of 6 and radius 5 (or possibly 3). That image is below, and all the unsharp-ed masking images are found at the end of this document.



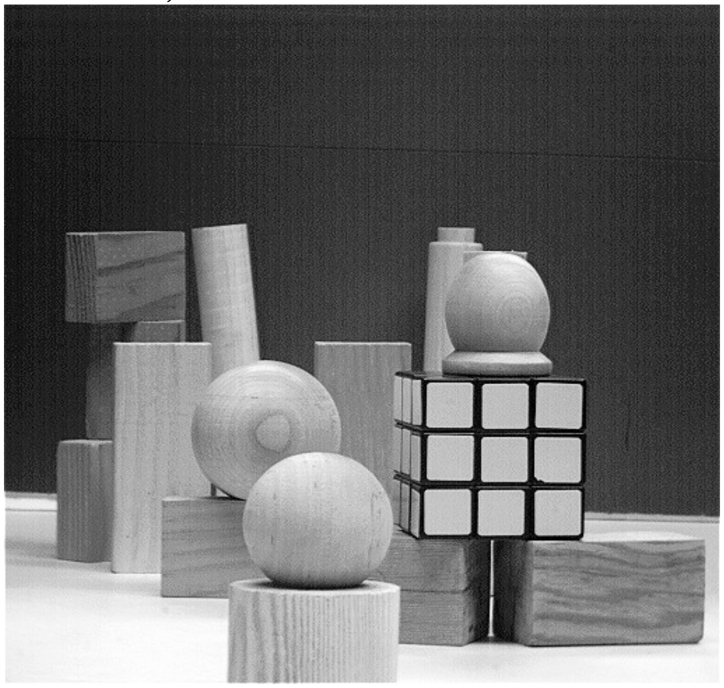
Programming Exercise Write-up:
The code is found starting on the next page. The only major challenge I encountered was at the start of the project, before implementing any of the filtering functions. As you can see in the commented out code, when I was reading in the images, either they would appear in non-gray-scale or they would be flipped. I discovered that using the pylab and python imaging libraries was creating the majority of this problem, since they treat images differently, but after studying more about the matplotlib.pyplot library, I was able to do everything I needed to do with that. The written part took about 2 hours, and the programming took about 8 hours.

Code:
```
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

def run_arithmetic_operations():
      # For some reason, when I used what I normally would
      # i.e.      img = pl.imread("Frames/Cat0.pgm")
    #       pl.imshow(img)
    #       plt.show()
    # the image had all its colors inverted, so this
    # fixes it but is much more confusing...
    #
    # img = pili.open("Frames/Cat0.pgm")
    # img.load()
    # back = pili.new("RGB", img.size, (255, 255, 255))
    # back.paste(img)
    # pl.imshow(np.asarray(back)) # PIL and matplotlib don't use same conventions,
else would flip
    # plt.show()

    # results in image flipped, but as long as I'm consistent
    shape = plt.imread("Frames/Cat0.pgm").shape
    reduce_noise(2, shape)
    reduce_noise(5, shape)
    reduce_noise(10, shape)
    reduce_noise(20, shape)
    reduce_noise(40, shape)


def reduce_noise(num_images, shape):
    new_img = np.zeros(shape)
    for i in range(1, num_images):
        next_img_name = "Frames/Cat" + str(i) + ".pgm"
        next_img = plt.imread(next_img_name)
        new_img += next_img

    new_img = new_img * (1.0/num_images)

    # pl.imshow(np.asarray(new_img), cmap=cm.Greys_r)
    new_img_name = "Cat_with_" + str(num_images) + ".png"
    plt.imsave(new_img_name, new_img, cmap=cm.Greys_r)


def run_spatial_filtering():
    # img_name = "noisygull"
    # img = np.asarray(plt.imread(img_name + ".png")[:,:,0])
    # uniform_averaging(img, img_name, 3)
    # uniform_averaging(img, img_name, 5)
    # uniform_averaging(img, img_name, 7)
    # uniform_averaging(img, img_name, 9)

    # median_filtering(img, img_name, 3)
    # median_filtering(img, img_name, 5)
    # median_filtering(img, img_name, 7)

    # img_name = "whitebox"
    # img = np.asarray(plt.imread(img_name + ".png")[:,:,0])
    # uniform_averaging(img, img_name, 3)
    # uniform_averaging(img, img_name, 5)
    # uniform_averaging(img, img_name, 7)
    # uniform_averaging(img, img_name, 9)

    #sobel_kernel(img, img_name) # does both x and y
    #laplacian_kernel(img, img_name)

    # gradient_magnitude_image(img, img_name)

    img_name = "blocks"
    img = np.asarray(plt.imread(img_name + ".png")[:,:,0])
    # gradient_magnitude_image(img, img_name)
    # blurred_img_3x3 = uniform_averaging(img, img_name, 3)
    # gradient_magnitude_image(blurred_img_3x3, img_name + "_blurred_3x3")

    # blurred_img_5x5 = uniform_averaging(img, img_name, 5)
    # gradient_magnitude_image(blurred_img_5x5, img_name + "_blurred_5x5")

    # blurred_img_7x7 = uniform_averaging(img, img_name, 7)
    # gradient_magnitude_image(blurred_img_7x7, img_name + "_blurred_7x7")

    # blurred_img_9x9 = uniform_averaging(img, img_name, 9)
    # gradient_magnitude_image(blurred_img_9x9, img_name + "_blurred_9x9")

    # Add to A(original_strength) the amount 1's, which depends on the radius
    unsharp_masking(img, img_name, orig_strength=2, radius=3)
    unsharp_masking(img, img_name, orig_strength=3, radius=3)
    unsharp_masking(img, img_name, orig_strength=4, radius=3)
    unsharp_masking(img, img_name, orig_strength=5, radius=3)
    unsharp_masking(img, img_name, orig_strength=6, radius=3)
    # unsharp_masking(img, img_name, orig_strength=2, radius=5)
    # unsharp_masking(img, img_name, orig_strength=3, radius=5)
    # unsharp_masking(img, img_name, orig_strength=4, radius=5)
    # unsharp_masking(img, img_name, orig_strength=5, radius=5)
    # unsharp_masking(img, img_name, orig_strength=6, radius=5)
    unsharp_masking(img, img_name, orig_strength=2, radius=7)
    unsharp_masking(img, img_name, orig_strength=3, radius=7)
    unsharp_masking(img, img_name, orig_strength=4, radius=7)
    unsharp_masking(img, img_name, orig_strength=5, radius=7)
    unsharp_masking(img, img_name, orig_strength=6, radius=7)


def spatial_filtering(img, kernel, spat_func, normalization=None):
    new_img = np.zeros((img.shape[0], img.shape[1]), img.dtype) # I am only caring
about one 2 dimensions here, to make life easier
    for r in range(0, img.shape[0]):
        for c in range(0, img.shape[1]):
                new_img[r, c] = spat_func(img, kernel, r, c, normalization)

    return new_img


def unsharp_masking(img, img_name, orig_strength=1, radius=3):
    kernel = fill_unsharp_mask_kernel(orig_strength, radius)
    new_img = spatial_filtering(img, kernel, calculate_normal, 1.0)
    new_img = trim_extremes(new_img)
    new_img_name = img_name + "_unsharp_masking_s(" + str(orig_strength) + ")-r("
+ str(radius) +").png"
    plt.imsave(new_img_name, new_img, cmap=cm.Greys_r)
    return new_img


def trim_extremes(img):
    for r in range(0, img.shape[0]):
        for c in range(0, img.shape[1]):
            if (img[r, c] < 0.0):
                img[r, c] = 0.0
            elif (img[r, c] > 1.0):
                img[r, c] = 1.0
    return img


def fill_unsharp_mask_kernel(original_strength, radius):
    kernel = np.zeros((radius, radius))
    col_row_with_ones = (radius - 1)/2
    kernel[:,col_row_with_ones] = -1
    kernel[col_row_with_ones,:] = -1
    kernel[col_row_with_ones, col_row_with_ones] = original_strength +
(col_row_with_ones * 4)
    kernel = kernel * (1.0/original_strength)
    return kernel


def gradient_magnitude_image(img, img_name):
    new_img = np.zeros(img.shape, img.dtype)
    sobel_x, sobel_y = sobel_kernel(img, img_name)
    new_img_name = img_name + "_gradient_magnitude.png"
    new_img = np.sqrt(sobel_x ** 2 + sobel_y ** 2)
    plt.imsave(new_img_name, new_img, cmap=cm.Greys_r)
    return new_img


def uniform_averaging(img, img_name, dim):
    kernel = np.ones((dim, dim))
    new_img_name = img_name + "_" + str(dim) + "x" + str(dim) + ".png"
    normalizing_factor = kernel.shape[0] * kernel.shape[1]
    new_img = spatial_filtering(img, kernel, calculate_normal, normalizing_factor)
    plt.imsave(new_img_name, new_img, cmap=cm.Greys_r)
    return new_img


def median_filtering(img, img_name, dim):
    kernel = np.ones((dim, dim))
    new_img_name = img_name + "_" + str(dim) + "x" + str(dim) + ".png"
    new_img = spatial_filtering(img, kernel, calculate_median)
    plt.imsave(new_img_name, new_img, cmap=cm.Greys_r)
    return new_img

def sobel_kernel(img, img_name):
    x_kernel = np.zeros((3, 3))
    y_kernel = np.zeros((3, 3))

    x_kernel[0, 0] = -1
    x_kernel[0, 2] = 1
    x_kernel[1, 0] = -2
    x_kernel[1, 2] = 2
    x_kernel[2, 0] = -1
    x_kernel[2, 2] = 1

    y_kernel = x_kernel.transpose()

    x_new_img_name = img_name + "_sobel_kernel_x.png"
    y_new_img_name = img_name + "_sobel_kernel_y.png"
    normalizing_factor = 8.0
    x_kernel_img = spatial_filtering(img, x_kernel, calculate_normal,
normalizing_factor)
    y_kernel_img = spatial_filtering(img, y_kernel, calculate_normal,
normalizing_factor)

    plt.imsave(x_new_img_name, x_kernel_img, cmap=cm.Greys_r)
    plt.imsave(y_new_img_name, y_kernel_img, cmap=cm.Greys_r)

    return x_kernel_img, y_kernel_img


def laplacian_kernel(img, img_name):
    laplacian_kernel = np.asarray([[0, 1, 0], [1, -4, 1], [0, 1, 0]])
    new_img_name = img_name + "_laplacian.png"
    new_img = spatial_filtering(img, laplacian_kernel, calculate_normal, 1.0)
    plt.imsave(new_img_name, new_img, cmap=cm.Greys_r)
    return new_img


def calculate_normal(img, kernel, row, col, normalizing_factor):
    avg = 0
    a = int((kernel.shape[0] - 1) / 2)
    b = int((kernel.shape[1] - 1) / 2)
    for s in range(-a, a+1):
        for t in range(-b, b+1):
            if (row+s < 0) or (row+s >= img.shape[0]) \
                or (col+t < 0) or (col+t >= img.shape[1]):
                avg += 0
            else:
                avg += img[row + s, col + t] * kernel[a + s, b + t]
    return avg / float(normalizing_factor)


def calculate_median(img, kernel, row, col, normalizing_factor=None):
    arr = np.ndarray((1,1), img.dtype)
    a = int((kernel.shape[0] - 1) / 2)
    b = int((kernel.shape[1] - 1) / 2)
    for s in range(-a, a+1):
        for t in range(-b, b+1):
            if (row+s < 0) or (row+s >= img.shape[0]) \
                or (col+t < 0) or (col+t >= img.shape[1]):
                arr = np.append(arr, 0)
            else:
                arr = np.append(arr, img[row + s, col + t] * kernel[a + s, b + t])

    return np.median(arr)
```

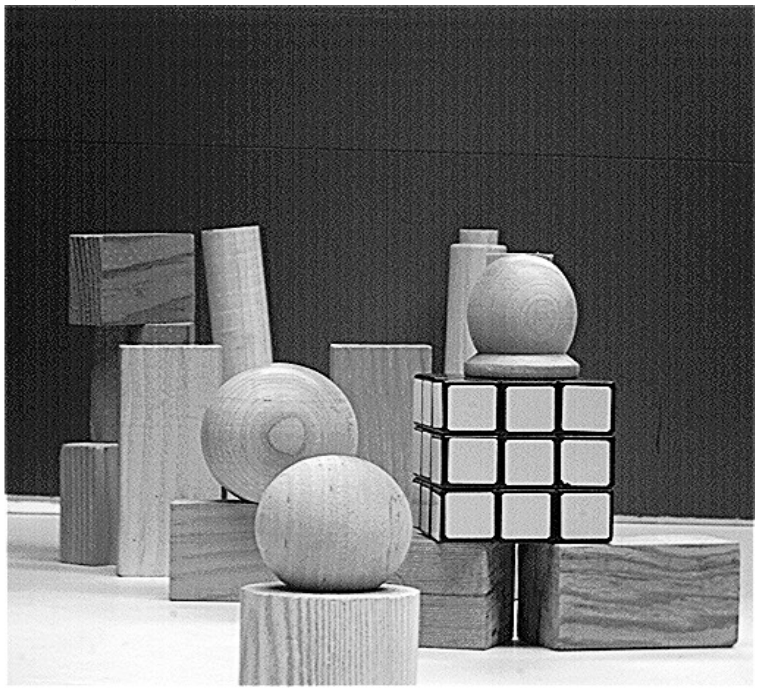
```
def main():
    #run_arithmetic_operations()
    run_spatial_filtering()


# boiler-plate code
if __name__ == "__main__":
    main()
```

More unsharp masking to sharpen *blocks.png*(s = strength, r = radius):

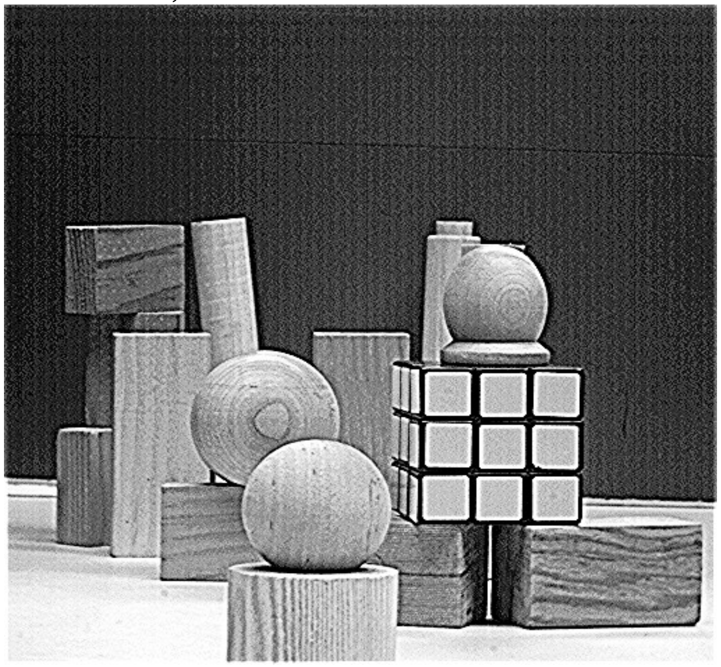
s=2, r = 3



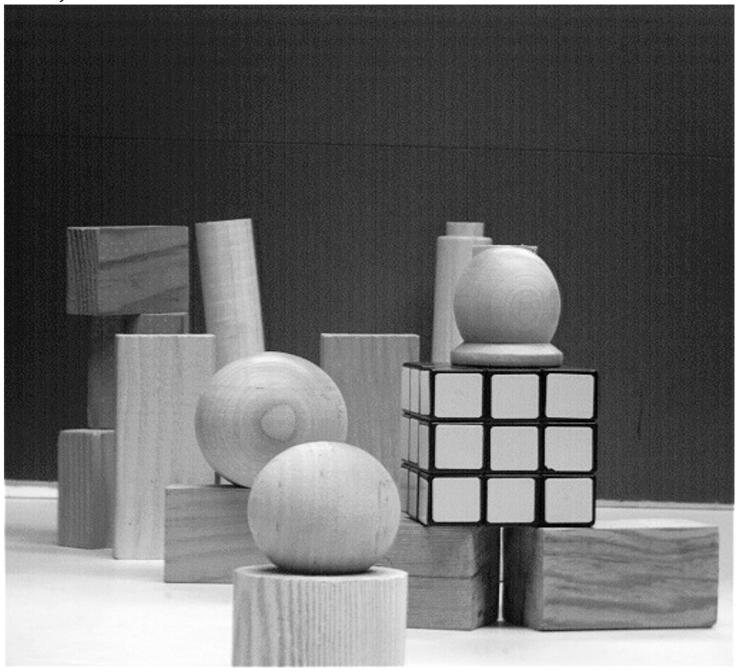
s=2, r= 5



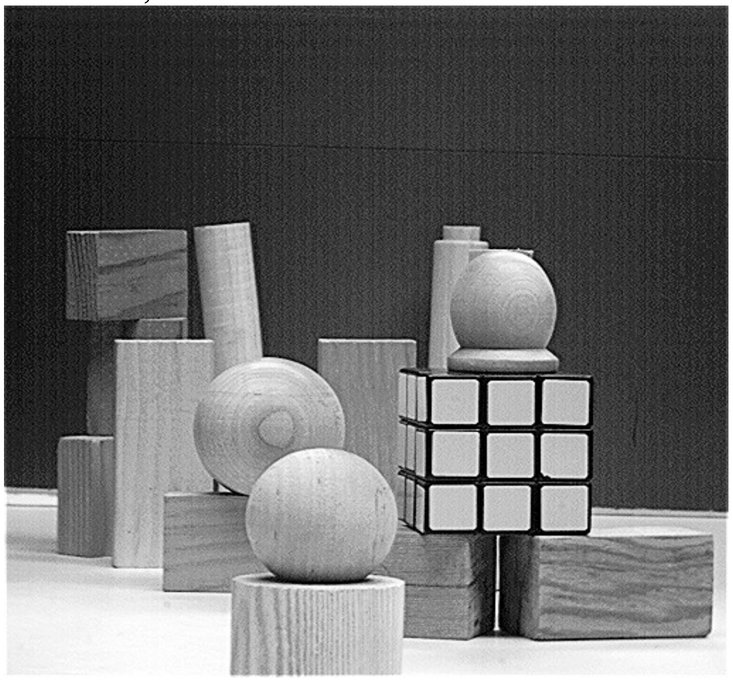
s=2, r=7



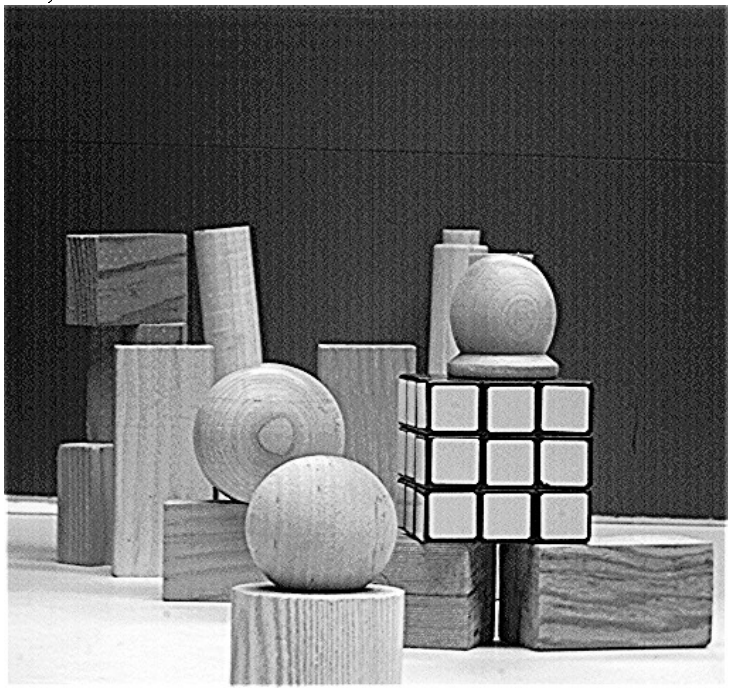
s=3, r=3



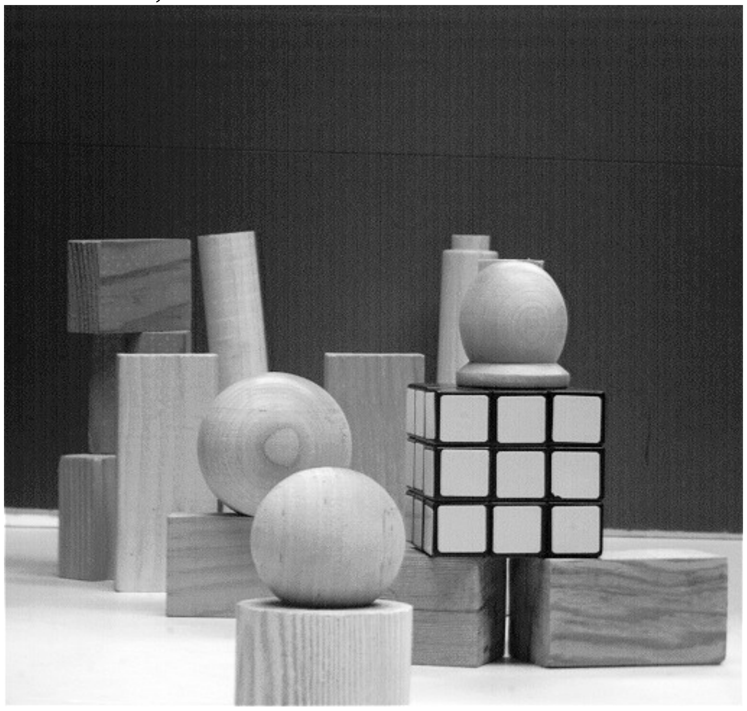
s=3, r=5



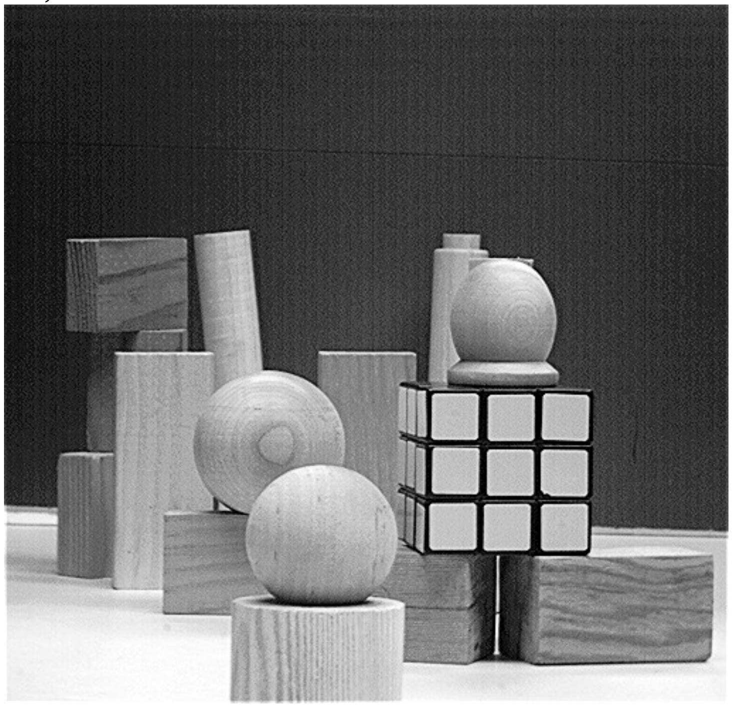
s=3, r=7



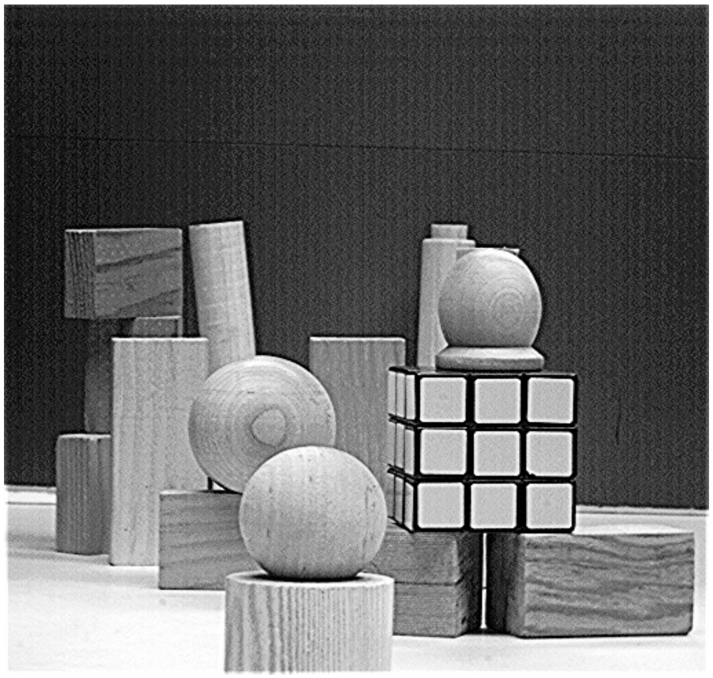
s=4, r=3



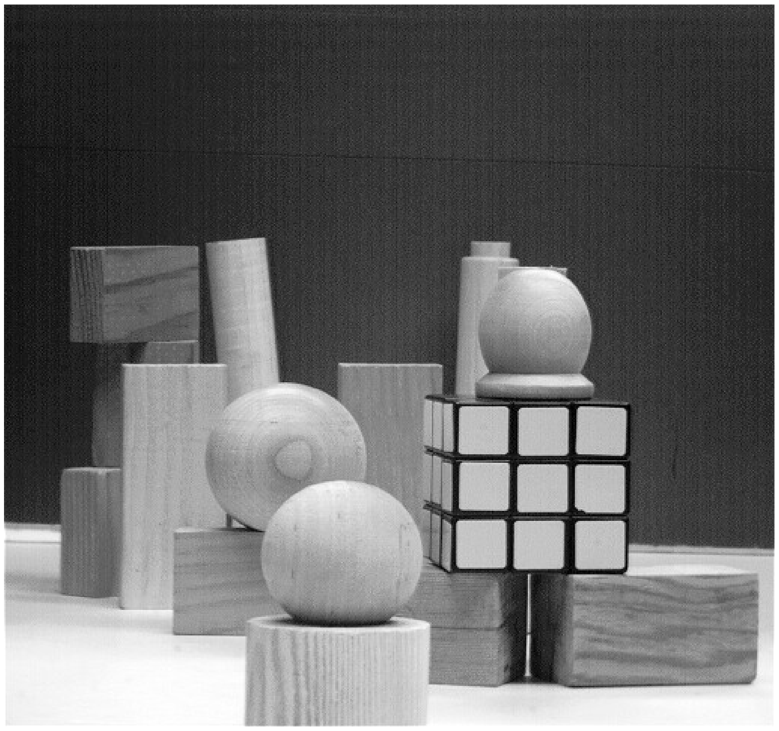
s=4, r=5



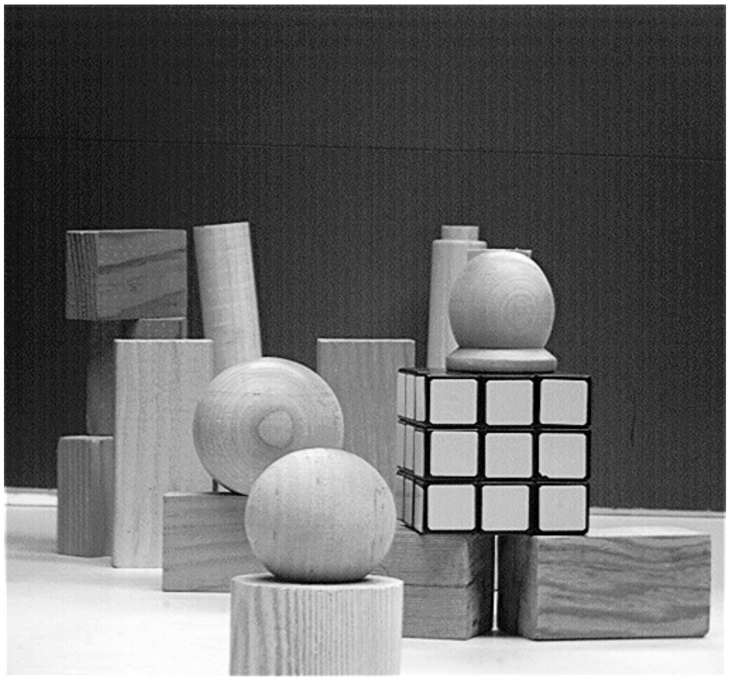
s=4 r=7



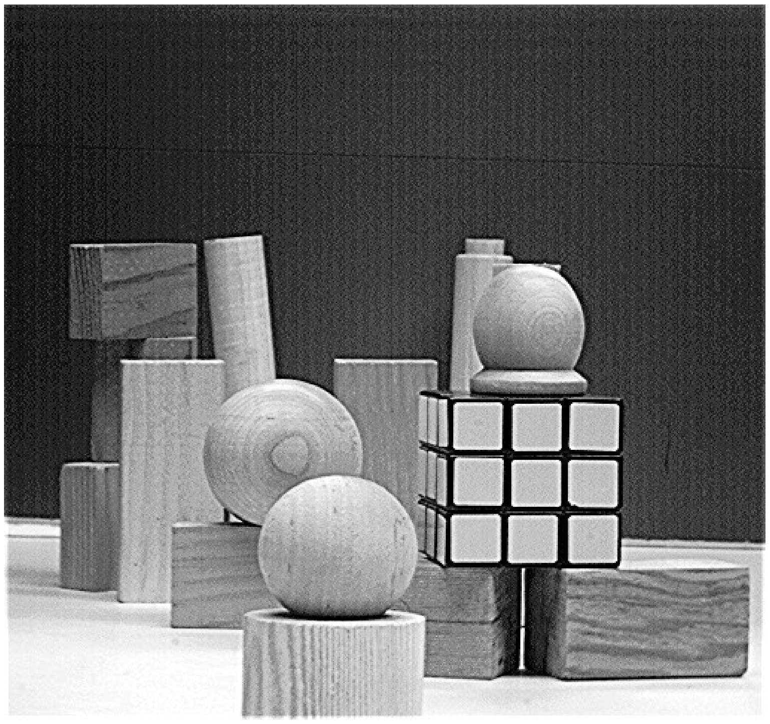
s=5 r=3



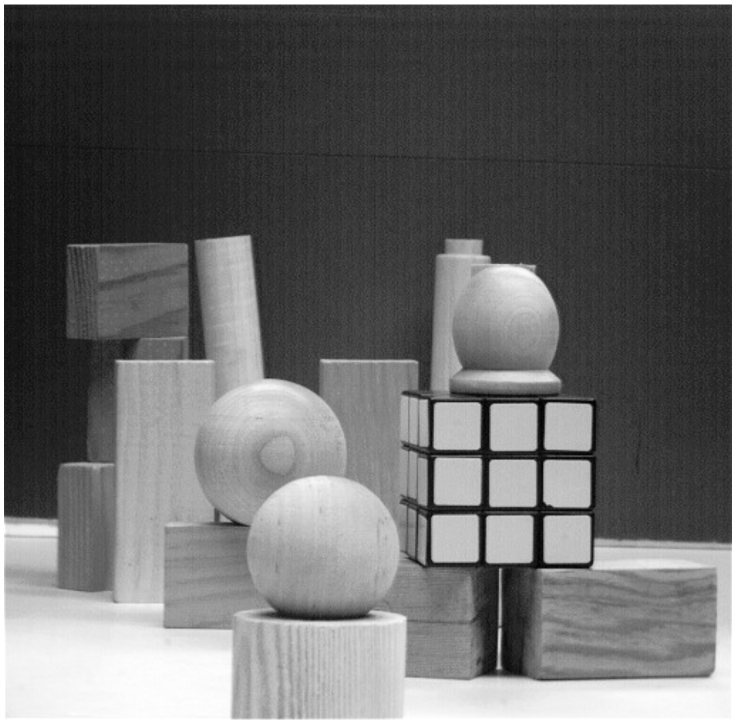
s=5 r=5



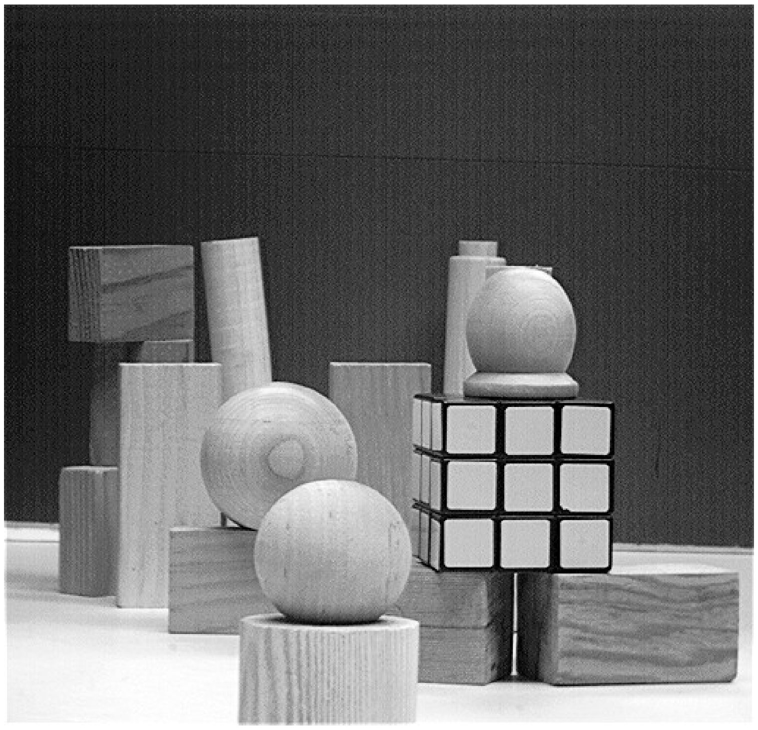
s=5 r=7



s=6 r=3



s=6 r=5



s=6 r=7