Denoising

Basic noise removal assumes that noise takes the form of pixels that are outliers in their immediate neighborhood. We can remove the noise by "smoothing out" or blurring an image slightly.

Uniform Blur

A **uniform blur** does so by setting the color of each pixel in the blurred image to be the average of the pixels in its neighborhood in the original image. The neighborhood takes the form of a $k \times k$ square around the pixel (where k is odd so that the square has a middle). Taking the uniform mean over this square is quite easy if we use numpy arrays. For example, suppose that the image `im` is an $n \times m$ numpy array of real numbers. Then, we can ask for the $k \times k$ sub-array around pixel i, j using the command `im[(i-(k-1)/2):(i+(k-1)/2)+1),(j-(k-1)/2):(j+(k-1)/2)+1)]`. Remember that when we ask for a range `a:b` in an array, the `a` is included in the output, and the `b` is not.

Also, when we use numpy arrays, we can multiply two arrays elementwise. This gives a new array which has elements that are the product of elements at corresponding indices in the original arrays. If we wish to find the mean of the elements in a $k \times k$ array, then we should multiply each element by $1/k^2$ (with k^2 being the number of elements). For example,

```
In:   a=np.array([[1,2],[3,4]])
In:   b=np.array([[1/4,1/4],[1/4,1/4]])
In:   a*b
Out:  array([[ 0.25, 0.5 ],[ 0.75,  1.  ]])
```

To obtain the mean we then need to sum over all these products. Notice the difference between the following two commands!

```
In:   sum(a)
In:   np.sum(a)
```

These are all the basic tools we need to write out denoising function. We can let k, the size of the filter, be a user defined parameter. You can think of this as the coarseness of the sandpaper that we use to smooth our image. The larger k is, the more blurring we get. Then, we create the filter. In the uniform case, this is a square $k \times k$ array where every value is $1/k^2$. For example, when $k = 5$, our filter is

$$\begin{matrix} 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \end{matrix}$$

We use the filter to recalculate the value of every pixel in the image (except for the ones too close to the edge to fit the filter around: we will leave those unchanged) to store in the new, blurred image. One important thing: when we find the new value for a pixel, we use the mean over the colors of its neighborhood pixels in the original image, even if some of its neighbors already have new values

for the blurred image. Therefore, you should store the new values in a new blurred image variable; without making changes in the original image!

Gaussian Blur

The idea behind a **Gaussian blur** is very similar to the uniform blur. The difference is that the weighting of the neighborhood pixels in the calculation of the mean is more sophisticated, and favors pixels that are closer to the current pixel. It achieves this by creating a filter that is weighted proposrtionally to a 2D Gaussian function (a Bell curve), with its maximum in the middle of the $k \times k$ filter. The 2D Gaussian is given by

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}}.$$

We wish to center this around the center of the square, which we achieve by giving each element `filter[i,j]` of the filter a value

$$filter[i, j] = \frac{1}{2\pi\sigma^2} e^{-\frac{\left(\left(i-\frac{k-1}{2}\right)^2+\left(j-\frac{k-1}{2}\right)^2\right)}{2\sigma^2}}.$$

The Gaussian distribution function is normalized, meaning that the total volume underneath it adds up to 1. However, we are only using a part of it that fits into the square, not the whole (infinite) function. This means that we need to renormalize the values in our filter. If their sum is less than 1, we would be making the pixels darker on average, which is undesirable. This is easy to do, by dividing the filter by its total sum at the end. Once we have created the filter, our denoising function works exactly the same way as it did with the uniform filter. Note that it has one more parameter: σ. For example, when $k = 5$, and $\sigma = 1$, our filter is

$$\begin{array}{ccccc}
1.1 \cdot 10^{-4} & 2.1 \cdot 10^{-3} & 5.8 \cdot 10^{-3} & 2.1 \cdot 10^{-3} & 1.1 \cdot 10^{-4} \\
2.1 \cdot 10^{-3} & 4.3 \cdot 10^{-2} & 1.2 \cdot 10^{-1} & 4.3 \cdot 10^{-2} & 2.1 \cdot 10^{-3} \\
5.8 \cdot 10^{-3} & 1.2 \cdot 10^{-1} & 3.2 \cdot 10^{-1} & 1.2 \cdot 10^{-1} & 5.8 \cdot 10^{-3} \\
2.1 \cdot 10^{-3} & 4.3 \cdot 10^{-2} & 1.2 \cdot 10^{-1} & 4.3 \cdot 10^{-2} & 2.1 \cdot 10^{-3} \\
1.1 \cdot 10^{-4} & 2.1 \cdot 10^{-3} & 5.8 \cdot 10^{-3} & 2.1 \cdot 10^{-3} & 1.1 \cdot 10^{-4}
\end{array}$$

By the way, this way of running the filter over our image: placing the filter on top of each pixel, and taking the sum of products of the overlaying elements, is mathematically called a **convolution**. You may have seen this before in other classes, but if not, don't worry. We simply write this as a double loop over all pixels (except for the edge). If you are interested in reading more about different types of blurring, there is a nice tutorial here `jhlabs.com/ip/blurring.html`. This is written for Java programmers, but the explanations are very nice.

Exercises

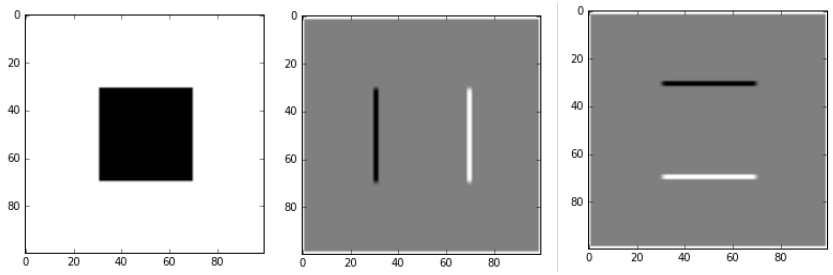
- Once you have written the functions for both uniform and Gaussian blurring, find or create images that show the difference in performance of the two methods. You may need to play around with the parameters, and with different levels of noise in your image.

Edge Detection

Detecting edges in image can be useful for image recognition, classification, retouching, etc.. Now that you have already written the denoising functions from the previous section, it will not be hard to write an edge detection function that uses the Sobel filter. When we detect edges, we are not interested in finding a kind of average of a set of pixels; we are interested in the difference! For example, a pixel is on a vertical edge when pixels to its left have a different color from pixels on its right. We can measure this difference in much the same way as we did before, by multiplying a pixel's neighborhood with a filter, and summing over the products. As a tiny example, if I want to measure the difference `b-a` between two elements of a small array `[a,b]`, I multiply elementwise by the array `[-1,1]`, and sum the result. To measure a vertical edge on a pixel, we look at the difference between a vertical strip of pixels to its left and to its right, weighting the immediate neighbors more heavily. The Sobel vertical and horizontal filters look, respectively, like,

$$\begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix} \qquad \begin{matrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{matrix}$$

For each pixel `i,j`, multiplying the 3×3 subarray around it with the (horizontal or vertical) Sobel filter yields a score (S_H or S_V) between -4 and 4. We can rescale this score to always be between 0 and 1 by taking $\frac{S_H+4}{8}$ or $\frac{S_V+4}{8}$. Now, gray (.5) indicates no edge, black indicates a negative edge, and white indicates a positive edge. This makes sense if we want to visualize the edges that we found in a grayscale image. When we ask for vertical or horizontal edge detection, we will see the difference between negative and postive edges (the color going from high to low or low to high, respectively). For example, the following image gives us the following output for vertical and horizontal edge detection, respectively. Notice the difference between positive and negative detected edges.



We may also ask for edge detection in general directions. We do this by combining the horizontal and vertical scores as Euclidean distances: $S = \sqrt{S_V^2 + S_H^2}$. Now, there is not more direction of edges, only strength. We can normalize this score so that black indicates no edge, and white indicates a strongest edge. This should look like this: