Filtering Noisy Images
CSE 400: Image and Video Processing
Geri Madanguit

In this assignment, we were give two distinct pictures below, one that was slightly noisy on the left (will be referred to as Noisy 1 in the text) and the other the same but added extra 'salt and pepper' noise on the right (will be referred to as Noisy 2 on the right). This report will detail the observations and results of filtering noise form images found from using three different methods: Average Smoothing by a Mean Filter, Gaussian Smoothing, and Median Filtering.

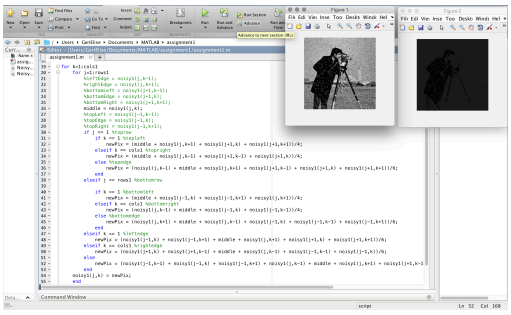
Noisy 1

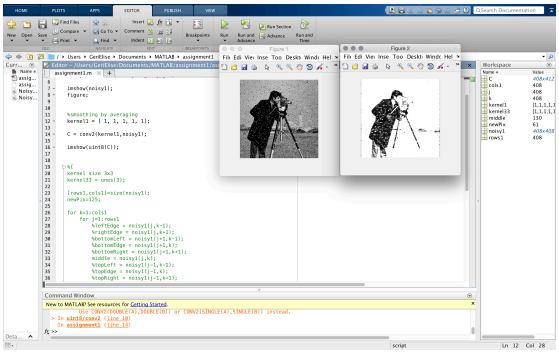
Noisy 2

Average Smoothing by a Mean Filter

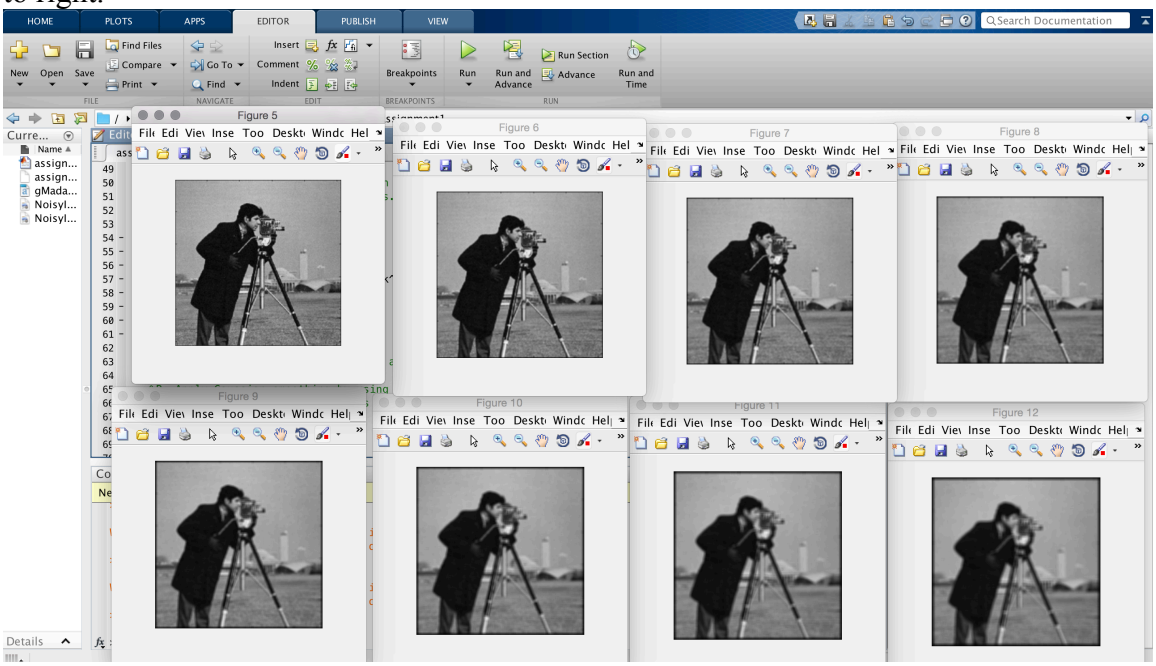
The simplest smoothing kernel is the mean filter, which replaces a pixel value with the mean of its neighborhood. For example with an mxm kernel, where m = 3, you would divide the sum of the neighbors and the pixel itself by 9. The mean filter can be regarded as an approximate low pass filter.

Before I understood the function of conv2 in Matlab, I attempted hardcoding a 3x3 matrix kernel containing all ones. My average smoothing failed miserably because the result I received back was a dark image with a subtle darker image to replace as the body of the person. Once, I understood more Matlab functions, I was able to implement the conv2 function. When implementing at first, I forgot to consider the division in the number of neighbors and resulted in the image below. But after correct implementation, I was able to produce a for-loop to run average smoothing with conv2 and a kernel m x m, incrementing m by 1 from 1 to 10. This helped me understand that as the size of the kernel increased the blurrier the image got, which decreased the image quality in terms of detail and fine lines.





In my observations, I noted that 3x3 was the ‘best’ kernel for Noisy 1 and 5x5 was the ‘best’ kernel for Noisy 2. When I mention ‘best’ it doesn’t determine that the noise for the image is completely removed with the kernel, it just determines the best output achieved without disintegrating the details and lines of the image. It is important to keep the lines and detail of the image fine to translate the image for later use and functionality. For example, in edge detection, it is harder to detect edges in a blurry image.
The image below represents the progression of Noisy1 as the kernel size increases. As you can see the image gets blurrier and that there is also an edge effect of the image, almost serving as a black frame, increasing in thickness as the kernel increases from left to right.



In conclusion, when averaging, signal frequencies shared with noise are lost, as well as sharp signal variations, resulting in a blurred image. Impulsive noise (‘salt and pepper’ a.k.a. Noisy 2) isn’t completely removed and only slightly diffused. In addition, it is good to note that the secondary lobes in the Fourier Transform of the mean filter let noise into the filtered image.

Gaussian Smoothing

Gaussian smoothing is a particular case of averaging but better because the Fourier transform is still Gaussian, and therefore has no secondary lobes. This makes the Gaussian kernel a better low-pass filter than the mean filter.

There is more to consider in Gaussian smoothing than the mean filter. In addition to the kernel, there were sigma values (the standard deviation) to play around to optimize the best-filtered image.

In this part of the assignment I implemented two different ways, both resulting the same, as expected. Convolving an image with 2D Gaussian kernel is the same as convolving

$$I_G = I * G =$$

$$= \sum_{h=-\frac{m}{2}}^{\frac{m}{2}} \sum_{k=-\frac{m}{2}}^{\frac{m}{2}} G(h,k) I(i-h, j-k) =$$

$$= \sum_{h=-\frac{m}{2}}^{\frac{m}{2}} \sum_{k=-\frac{m}{2}}^{\frac{m}{2}} e^{-\frac{h^2+k^2}{2\sigma^2}} I(i-h, j-k) =$$

$$= \sum_{h=-\frac{m}{2}}^{\frac{m}{2}} e^{-\frac{h^2}{2\sigma^2}} \sum_{k=-\frac{m}{2}}^{\frac{m}{2}} e^{-\frac{k^2}{2\sigma^2}} I(i-h, j-k)$$

first all rows, then all columns with 1D Gaussian having same sigma value. This is mathematically proven in the simplification equation process to the left. As you can see in the equation, the last part shows the 1D Gaussian being convolved with the image incrementing in the k values (be it rows or columns), and then being convolved with the resulting image incrementing in the h values.

For convolving an image with 2D Gaussian, I implemented the INT_GAUSS_KER algorithm referred in the book. For the convolving a 1D Gaussian, I implemented the SEPAR_FILTER algorithm, which is also referred in the book.

5x5 sigma = 1

9x9 sigma = 2

15x15 sigma = 4

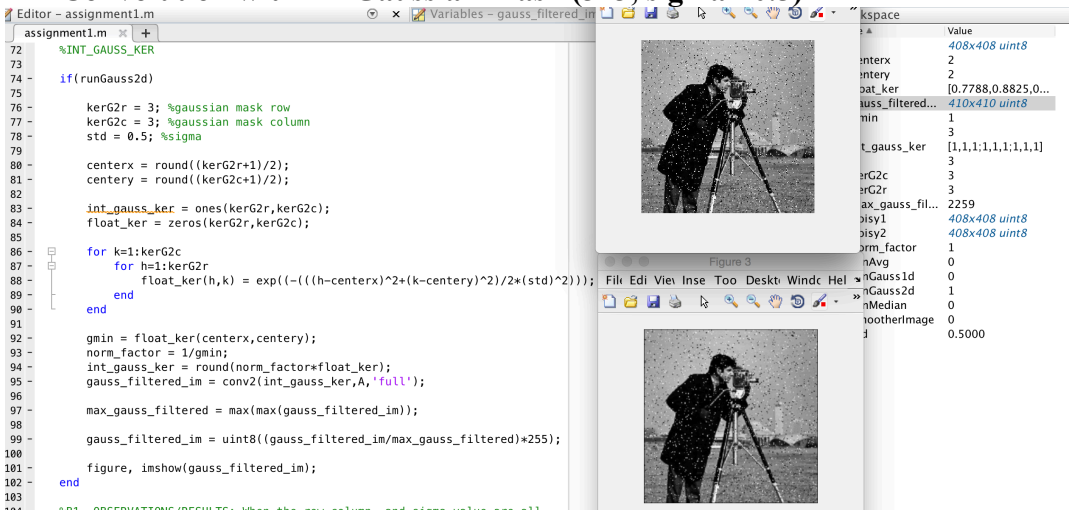
The Gaussian filter method thins out the higher frequencies more than the lower frequencies and exhibits no oscillations in frequency response. The space of the response curve is a Gaussian. Although slightly better than the mean filter, this method still has consequences for edge detection. In the Gaussian-altered Noisy 2 images above, there is a physical trend in the image as the kernel and sigma increase.

When building a kernel its good to keep in mind that the w should always be less than or equal to 5 * sigma, or half-width equal to 3 * sigma. These are good rule of thumbs to optimize image filtering with Gaussian.

What I noted in my observations, was when the Gaussian mask's row and column, and sigma were all of the same value, there was little to none effect on the image. Like the mean filter, there was an edge effect, 'a black frame', as the kernel increased in size. I also observed that for Noisy 1, the 'best' results were with a 3x3 kernel with sigma = 1. For Noisy 2, the 'best' results were with a 5x5 kernel with sigma = 1.

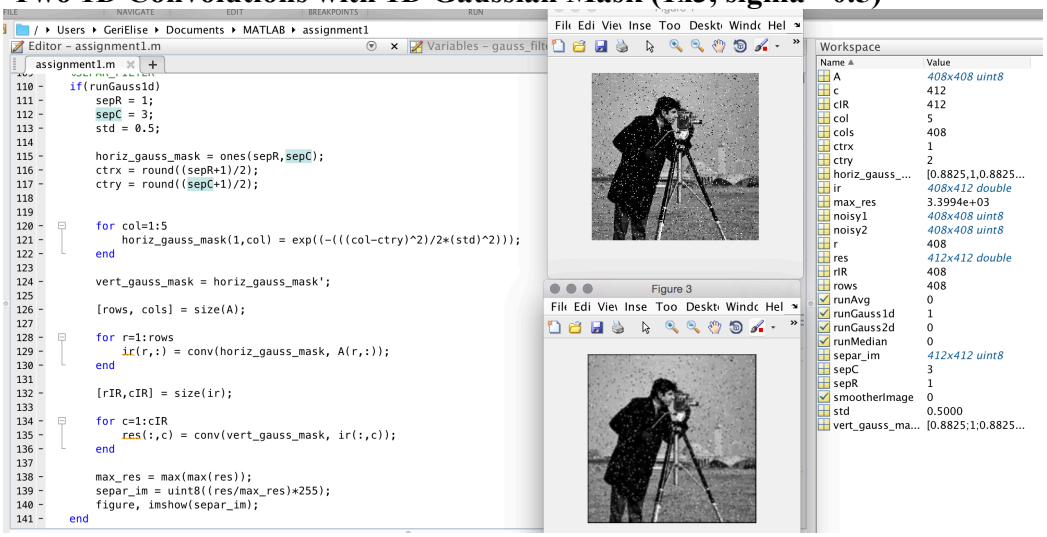
The images below show code and results between two different implementations of Gaussian.

2D Convolution with 2D Gaussian Mask (3x3, sigma= 0.5)



```
%INT_GAUSS_KER

if(runGauss2d)

    kerG2r = 3; %gaussian mask row
    kerG2c = 3; %gaussian mask column
    std = 0.5; %sigma

    centerx = round((kerG2r+1)/2);
    centery = round((kerG2c+1)/2);

    int_gauss_ker = ones(kerG2r,kerG2c);
    float_ker = zeros(kerG2r,kerG2c);

    for k=1:kerG2c
        for h=1:kerG2r
            float_ker(h,k) = exp((-(((h-centerx)^2+(k-centery)^2)/2*(std)^2)));
        end
    end

    gmin = float_ker(centerx,centery);
    norm_factor = 1/gmin;
    int_gauss_ker = round(norm_factor*float_ker);
    gauss_filtered_im = conv2(int_gauss_ker,A,'full');

    max_gauss_filtered = max(max(gauss_filtered_im));

    gauss_filtered_im = uint8((gauss_filtered_im/max_gauss_filtered)*255);

    figure, imshow(gauss_filtered_im);
end
```

Two 1D Convolutions with 1D Gaussian Mask (1x3, sigma= 0.5)



```
if(runGauss1d)
    sepR = 1;
    sepC = 3;
    std = 0.5;

    horiz_gauss_mask = ones(sepR,sepC);
    ctrx = round((sepR+1)/2);
    ctry = round((sepC+1)/2);


    for col=1:5
        horiz_gauss_mask(1,col) = exp((-(((col-ctry)^2)/2*(std)^2)));
    end

    vert_gauss_mask = horiz_gauss_mask';

    [rows, cols] = size(A);

    for r=1:rows
        ir(r,:) = conv(horiz_gauss_mask, A(r,:));
    end

    [rIR,cIR] = size(ir);

    for c=1:cIR
        res(:,c) = conv(vert_gauss_mask, ir(:,c));
    end

    max_res = max(max(res));
    separ_im = uint8((res/max_res)*255);
    figure, imshow(separ_im);
end
```

Median Filtering

Unlike the two methods previously described, Median filtering is non-linear and therefore cannot be modeled by convolution. In Median Filtering, each pixel is replaced by the median value of the neighborhood.

```
%{
for j=1:rM
  for k=1:cM
    %leftEdge = noisy1(j,k-1);
    %rightEdge = noisy1(j,k+1);
    %bottomLeft = noisy1(j+1,k-1);
    %bottomEdge = noisy1(j+1,k);
    %bottomRight = noisy1(j+1,k+1);
    middle = noisy2(j,k);
    %topLeft = noisy1(j-1,k-1);
    %topEdge = noisy1(j-1,k);
    %topRight = noisy1(j-1,k+1);
    if j == 1 %toprow
      if k == 1 %topleft
        A = [middle, noisy2(j,k+1), noisy2(j+1,k), noisy2(j+1,k+1)];
      elseif k == cM %topright
        A = [middle, noisy2(j,k-1), noisy2(j+1,k-1), noisy2(j+1,k)];
      else %topedge
        A = [noisy2(j,k-1), middle, noisy2(j,k+1), noisy2(j+1,k-1), noisy2(j+1,k), noisy2(j+1,k+1)];
      end
    elseif j == rM %bottomrow
      if k == 1 %bottomleft
        A = [middle, noisy2(j-1,k), noisy2(j-1,k+1), noisy2(j,k+1)];
      elseif k == cM %bottomright
        A = [noisy2(j,k-1), middle, noisy2(j-1,k), noisy2(j-1,k-1)];
      else %bottomedge
        A = [noisy2(j,k+1), middle, noisy2(j,k-1), noisy2(j-1,k), noisy2(j-1,k-1), noisy2(j-1,k+1)];
      end
    elseif k == 1 %leftedge
      A = [noisy2(j-1,k), noisy2(j-1,k+1), middle, noisy2(j,k+1), noisy2(j+1,k), noisy2(j+1,k+1)];
    elseif k == cM %rightedge
      A = [noisy2(j+1,k), noisy2(j+1,k-1), middle, noisy2(j,k-1), noisy2(j-1,k-1), noisy2(j-1,k)];
    else
      A = [noisy2(j-1,k-1), noisy2(j-1,k), noisy2(j-1,k+1), noisy2(j,k-1), middle, noisy2(j,k+1), noisy2(j+1,k-1), noisy2(j+1,k), noisy2(j+1,k+1)];
    end
```

When first tackling this filtering method, I attempted to hardcode it as you can see on the left. I hard-coded the nested for-loop to get the median values in a 3x3 kernel. The results were successful, however I knew the code could be more optimized and adaptable to easily transition between different kernel sizes.

It took me a while to figure out a more optimal solution, but I finally found Matlab functions called im2col and col2im to solve Median Filtering.

```
if(runMedian)
    mfR = 3; %median kernel row
    mfC = 3; %median kernel column

    column_to_median = im2col(A, [mfR mfC]);
    median_to_im = median(column_to_median);
    med_filtered_im = col2im(median_to_im,[mfR mfC],size(A));
    figure, imshow(med_filtered_im);
end
```

As you can see this code cut down about 30 lines of code, while easily being adaptable to test out different kernel sizes when manipulating it. Median was very successful for the 'salt and pepper noise' Noisy 2 image. However there was little to no difference when filtering out the noise in Noisy 1. When set to a 2x2 kernel, there was practical no physical difference and when set to a 4x4 kernel, the smoothing, blurred out the picture too much and lost quality in detail. So I would say the 'best' kernel for Noisy1 is a 3x3. The same thinking process went into deciding the 'best' kernel for Noisy2 also. 3x3 had still too much 'salt and pepper' noise, while 5x5 was too blurred, and so the happy median was a 4x4 kernel, with a few 'salt and pepper' noise still left over, but too blurred out.

This image is the 3x3 kernel for Noisy1. It is very smoothed out, with little to none noise. But much of the detail has been taken out and it shows that the previous methods are also adequate enough, or even better to handle the noise in Noisy 1.

In observation, the bigger the kernel (the neighborhood), the smoother the results.
In conclusion, Median Filtering is the best for 'salt and pepper' noise.

Averaging by Smoothing vs. Gaussian Filtering vs. Median

	3x3	5x5	7x7
NOISY2. Averaging by Smoothing			
NOISY2. Gaussian (sigma = 0.5)			
NOISY2. Median			
NOISY1. Averaging by Smoothing			

NOISY1. Gaussian (sigma = 1)			
NOISY1. Median			

From these results, the best filtering method for Noisy1 would be the Gaussian Smoothing and the best filtering method for Noisy2 would be the Median Filtering

The Average for Smoothing worked ok for Noisy 1 but Gaussian worked better because it is a particular case of averaging but as a better low-pass filter than the mean filter for the noise.

'Salt and Pepper' noise is often modeled as multiplicative noise. Traditional low-pass or other frequency domain approaches aren't very effective. A median filter is a non-linear filter and is more effective against this type of noise. This is clearly true based on the physical evidence shown above.