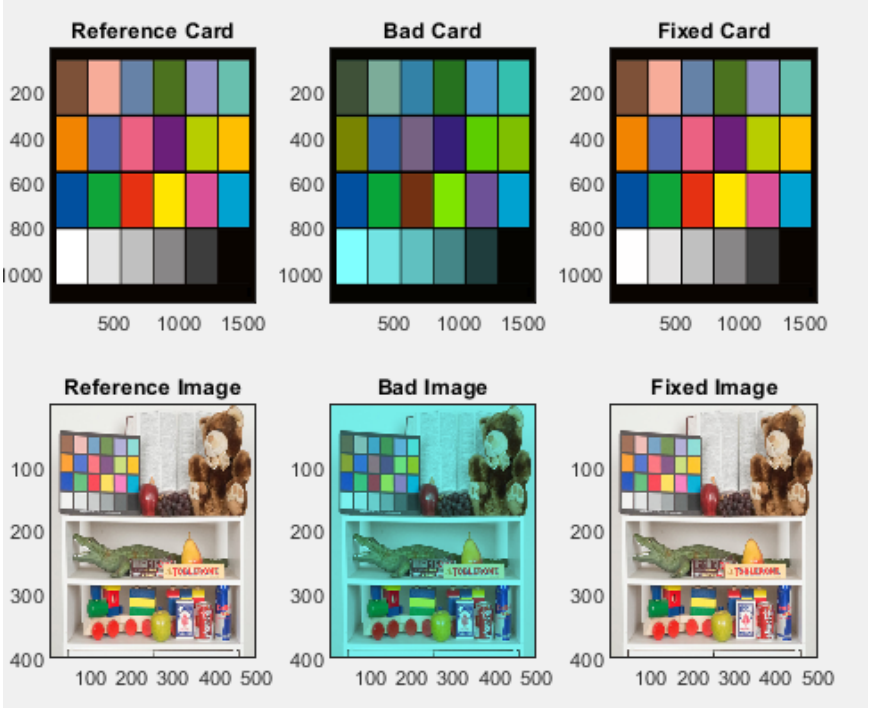
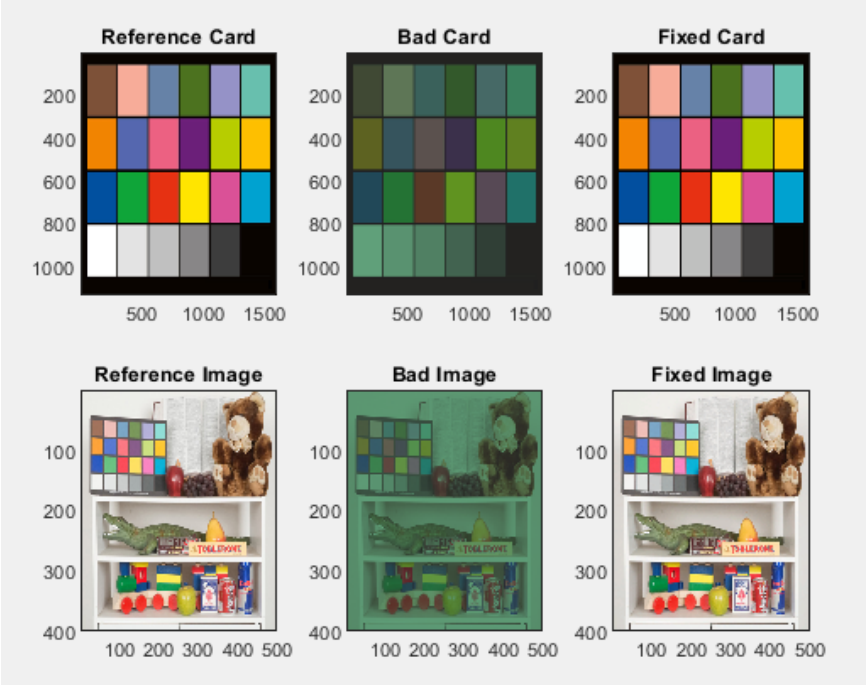
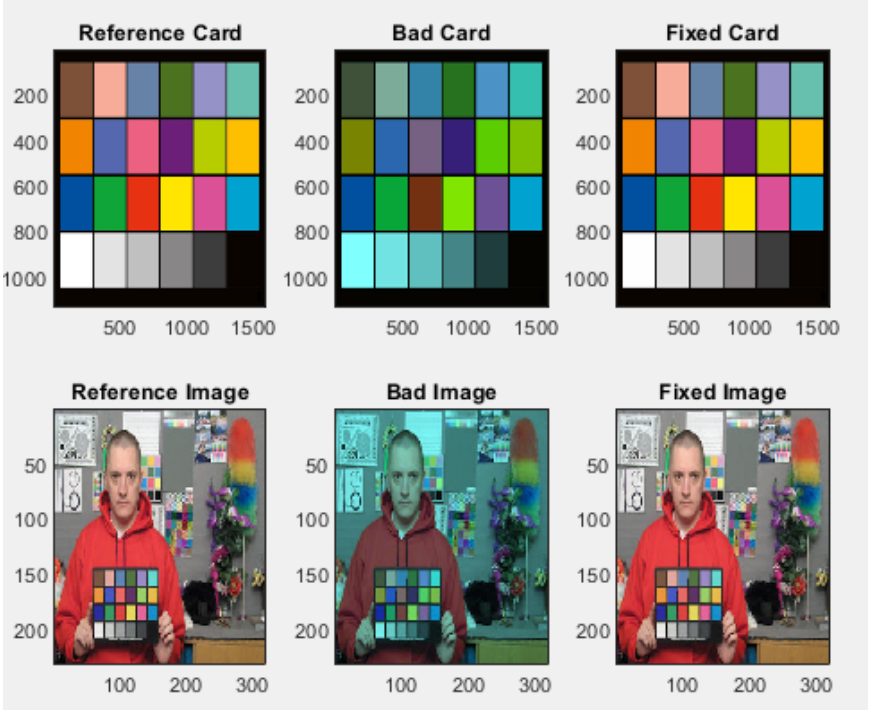
Assignment 1

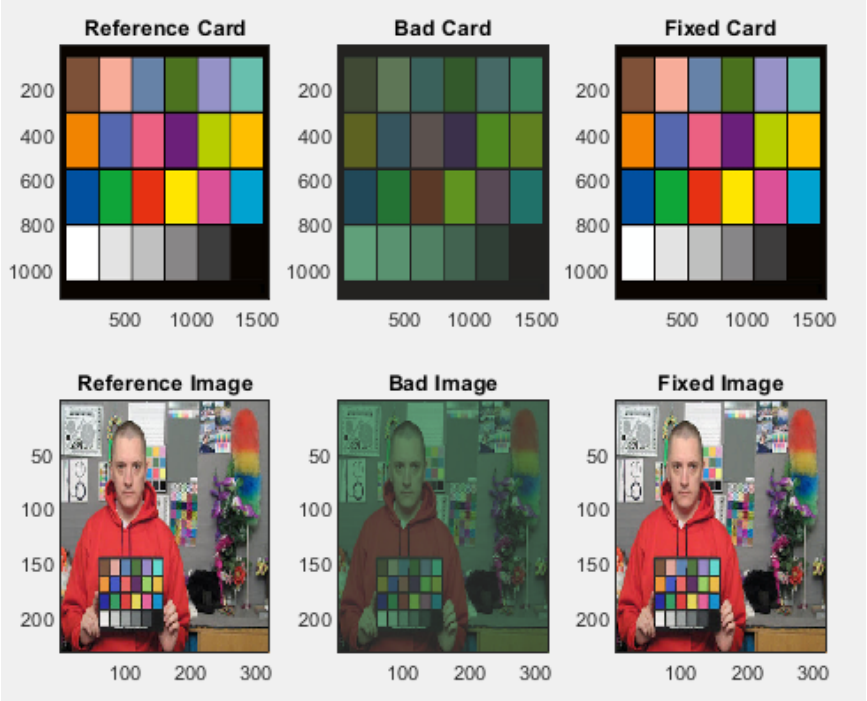
Exercise 1A

Results









Comments

The three functions that were created were successful in doing their intended purposes. The entire process successfully color corrected a bad image based on reference cards.

get_chart_values() took 0.002s. This can be attributed to greatly reducing the number of samples taken from M*N to 24. This was done by taking 24 fixed points in the image, knowing that the card always has 4 rows, with 6 columns that are equally spaced. This reduces running time from a time complexity of O(M*N) to O(1).

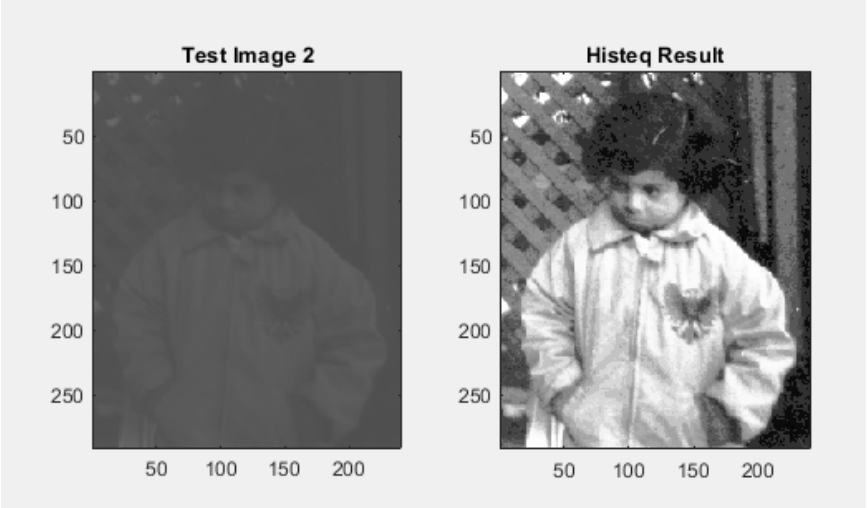
chart_correction() took 0.007s. Polyfit took 59.2% of the running time and polyval took 28.4%. The rest were spent on other lines.

Surprisingly, apply_rgb_map() took the longest. When running chart_test(), apply_rgb_map() took 0.770s or 64.0% of the entire running time. Looking further into the profiler, the line that took the longest was replacing the values of the pixels with the corresponding value in the lookup table. This entire process has a time complexity of O(M*N*3), as each pixel of each color needs to be individually mapped to the correct values.

Overall, the entire process has a linear time complexity due to mapping of the correct values of the pixels individually. Removing that function, the rest of the process has a constant time complexity.

Exercise 1B

Results



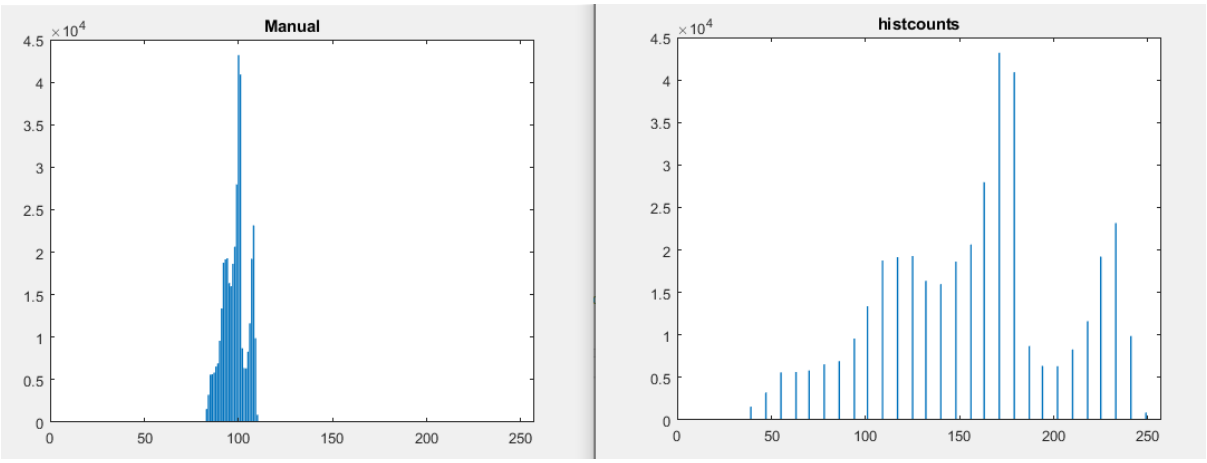
Comments

The filter was successful in improving the contrast of the images. As seen in the second and fourth image, even if the contrast was very bad, the filter was able to correct it and bring out the different elements in the image that was washed out due to very low contrast.

Running the filter on one image took 0.026s. The majority of the running time was taken by generating the histogram. As a side note, this process was done manually, as using the builtin function, histcounts(), led to results that were too dark. histcounts() somehow stretched the histogram to fill the most of the bins, even if the values of the image were multiplied by 255 beforehand. The image following the comments shows a bar chart of the histogram and the difference in the generated vector using histcounts and manual counting of the values.

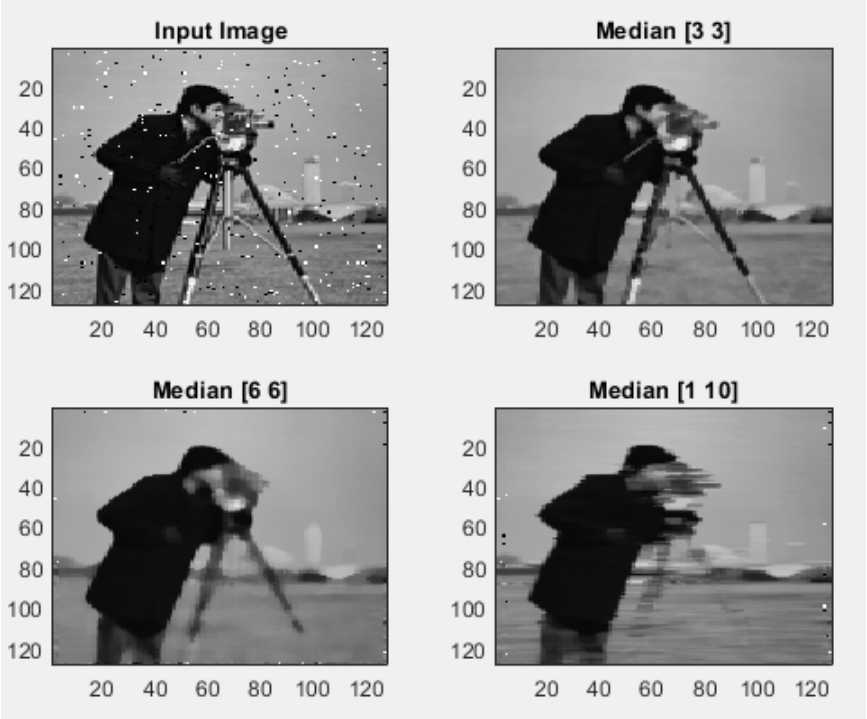
Going back to the running time, generating the histogram and applying the lookup table took the most time as each pixel had to be iterated through. Histogram generation took 0.012s or 46.15% of the running time. Applying the lookup table took 0.011s or 42.31% of the running time.

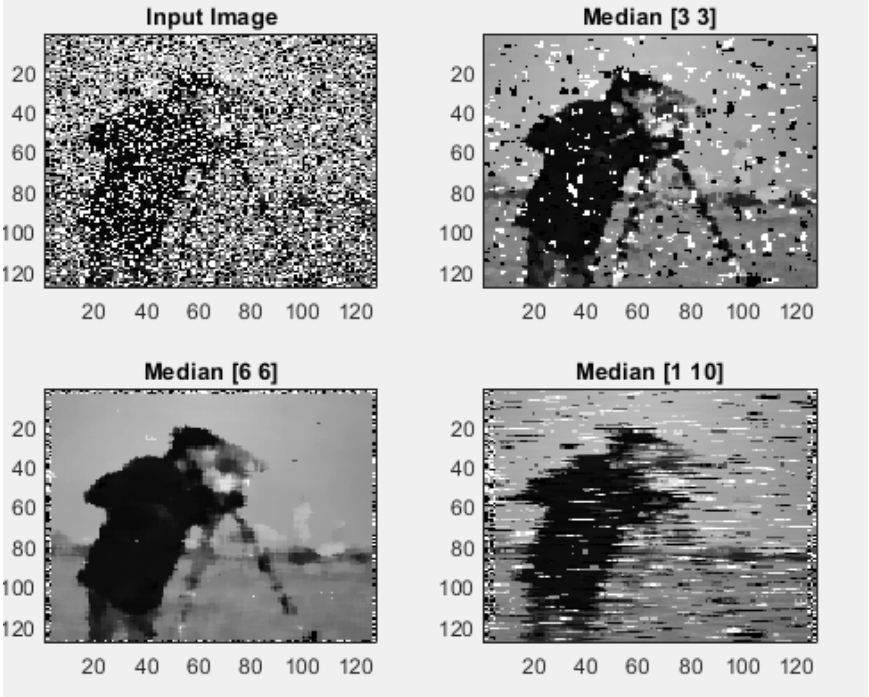
The process has a time complexity of O(M*N) or simply, it takes linear time for this function to finish as each pixel needs to be accessed and manipulated.

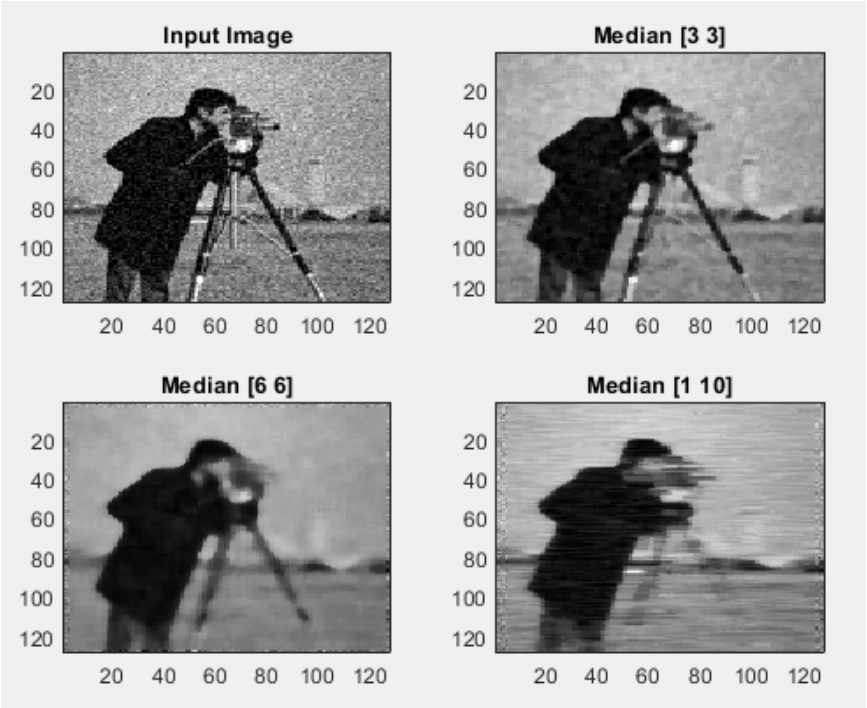


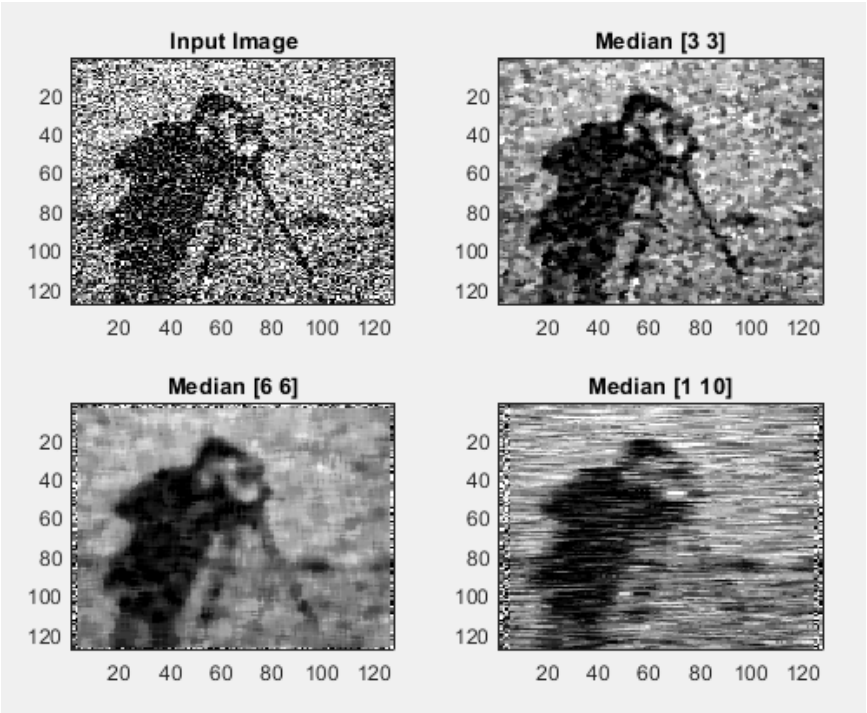
Exercise 1C

Results









Comments

The function created was successful in removing salt and pepper noise in the given image. In the second example with heavy salt and pepper noise in the image, the subject in the image was recovered using a median filter of 6x6. It is still a bit blocky, a known property of median filters, but it is still a very good result, taking into account the input image

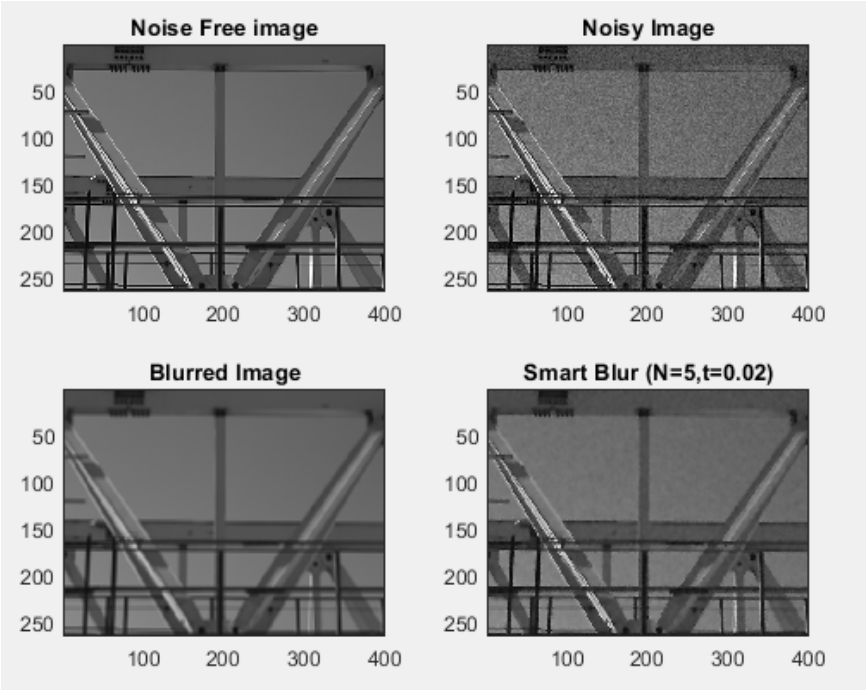
The function did not do a good job in removing gaussian noise, as evidenced by the third and fourth images. Artifacts of the blur remained, even if the noise levels of the gaussian noise was not that high. **Therefore, a median filter fails when the noise in the image is a gaussian noise.**

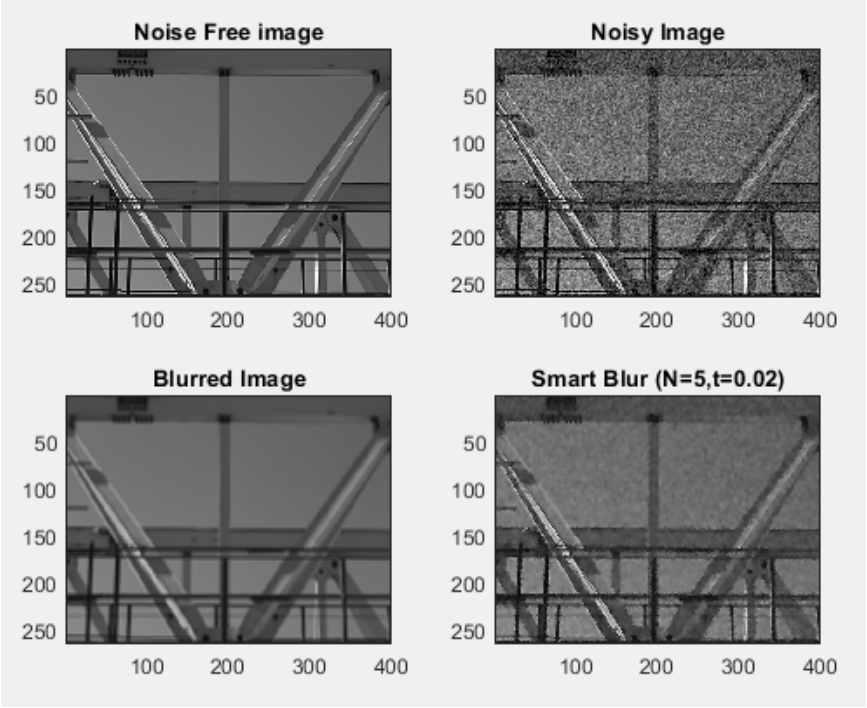
Running the median_test() function, median_filter() took up just 0.130 or 10.2% of the running time. Taking into account that 3 images were passed, a single run of the median_filter() has an average time of 0.043s. Running a single function call to median_filter() with a 6x6 filter took 0.055s. Looking into the function deeply, Getting the neighboring pixels and sorting them took the longest, with getting the neighboring pixels taking 0.014s or 25.45% of the running time and sorting taking 0.021 or 38.18% of running time.

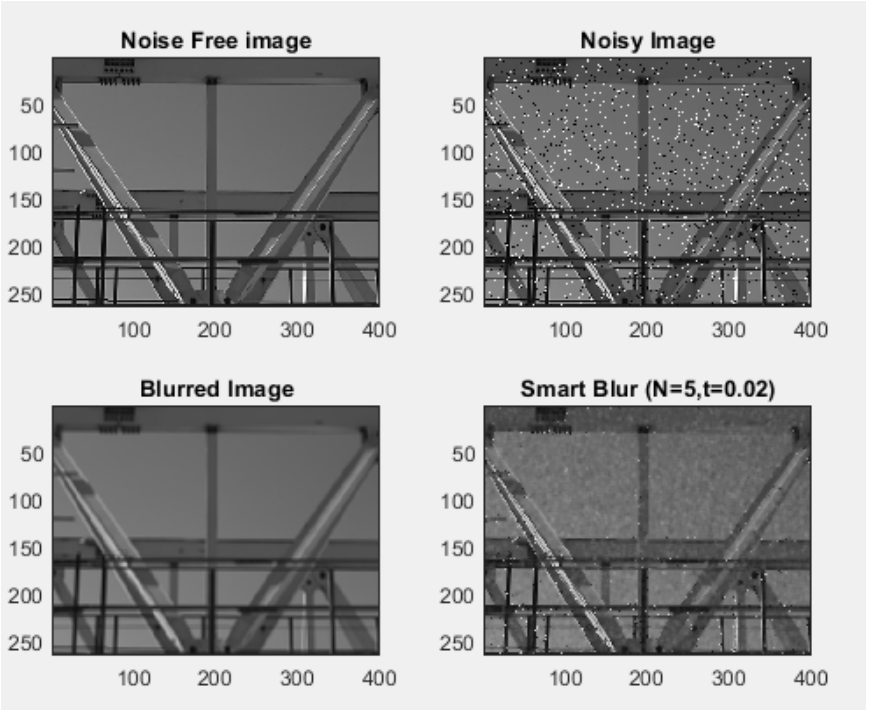
The function has a time complexity of O(M*N) since the function takes a look at each pixel in the image. The edges are not included in this, but this still does not affect the time complexity - it is still linear.

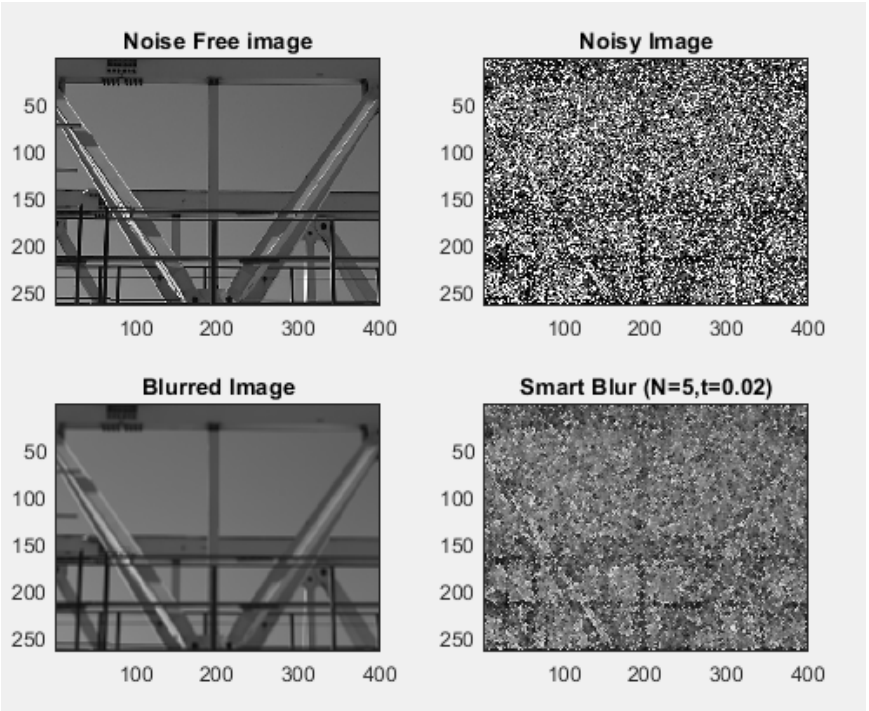
Exercise 1D

Results









Comments

The function created was successful in removing gaussian noise in the image, while retaining the edges, as shown in the first and second images. There are still artifacts from the noise, but overall the edges were retained in the image.

However, the function has less than ideal results when salt and pepper noises were introduced in the image, as shown in the third and fourh image. A lot of artifacts were in the image even if the noise level was minimal. In addition, The image was destroyed when the noise level in the image was great. One reason for this is that the sobel filter sees each dot in the salt and pepper noise as an edge. This means that less weight is given in these areas which results in more of the original image showing through.

In terms of run time, a single call to the function took 0.044s. Computing the gradient, weights, and creating the final image took the longest. Computing the gradient took 0.009s or 20.45% of the running time. Computing the weight took 0.021s or 47.73% of the running time. Finally, creating the final image took 0.009s or 20.45% of the running time. These three actions all iterate over the entire image, scanning each pixel. This results in a linear time complexity.

Exercise 1E

Item 1

1. $2m = 32px(10m)\frac{2}{256px}tan(\frac{f}{2}\cdot\frac{\pi}{180})$

   $0.2 = 32px\frac{2}{256px}tan(\frac{f}{2}\cdot\frac{\pi}{180})$

   $\frac{0.2}{0.25} = tan(\frac{f}{2}\cdot\frac{\pi}{180})625$

   $arctan(\frac{0.2}{0.25}) = \frac{f}{2}\cdot\frac{\pi}{180}$

   $f = 77.32rads$

2. $1.75m = 8pxd\frac{2}{256px}tan(\frac{77.32rad}{2}\cdot\frac{\pi}{180})$

   $1.75m = 0.0625dtan(\frac{77.32rad}{2}\cdot\frac{\pi}{180})$

   $\frac{1.75m}{0.0625} = dtan(\frac{77.32rad}{2}\cdot\frac{\pi}{180})$

   $$\frac{\frac{1.75m}{0.0625}}{tan(\frac{77.32rad}{2}\cdot\frac{\pi}{180})} = d$$

   $d = 35m$

Item 2

3. A median filter takes the most common value in a specified area around a pixel as the value of that pixel. As an example, a 1x9 median filter sees an area around a pixel with these values

[0 0 1 1 1 1 1 1 1]. The filter choses the most common value, which is one, as the value of the pixel currently being examined

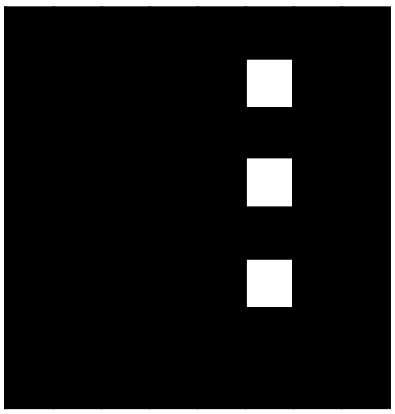
This type of filter is best used in removing images with salt and pepper noise. The filter can also remove some white noise. This filter can also improve images with gaussian noise, but results in some blockiness.

On the other hand, an alpha trimmed filter takes the pixel values of an area around a pixel, it removes a number of minimum and maximum values in these values, then calculates the average of the remaining values. The average is then used as the value of that pixel.

As an example, a 1x9 alpha trimmed with d = 3 sees an area around a pixel with these values [0 0 0 0 1 1 1 1 1]. The filter removes d items from the start and end, resulting in these values: [0 1 1]. These values are then averaged. Ther result, 0.66 is now the value of the pixel being examined.

This type of filter is best used in removing gaussian noise. This filter can also remove salt and pepper noise, since this type of noise is usually the minimum and maximum values of an image.

4. 3x3 Median



5x5 Median

3x3 Alpha Trimmed Mean with d = 3