Performance Comparison among Three Common Filter Techniques for 2D Image De-Noise

Submitted by: Yu Yang & Yue Zhang
@Portland State University
June 15, 2012

Abstract
This paper is written for the course project for CS510 Computational Photography class. This paper is organized as following four sections. Firstly, we talk about the basic concepts of image filtering for image processing in Section 1. Then we describe the algorithms and implementation details for three filtering technique: naive filter, fast median filter, bilateral filter in Section 2. Next we show the results from our application and compare the performance in terms of timing data for those algorithms.

1. INTRODUCTION

Filtering is one of the most fundamental operation of image processing and computer vision. By using the common mean filters with the idea of computing the average value of the neighboring pixels at each pixel position, it is very suitable for removing the granularity noise from scanned images. The effect of de-noise of this method is effective. However, in the meantime the blurring issue is raised because of the averaging operation. The degree of the blurring of the output images is in proportion to the radius of neighboring.

Median filter is a very useful nonlinear smoothing filter. The fundamental principle of median filters is to replace a certain pixel in a sequence of digits or digital images with the median from its neighborhood. By doing this, the pixels in the neighborhood with a high difference in intensity would be replaced with the value that are approximate to its neighboring pixels. Thus the independent noisy point can be eliminated and it is very good at removing salt and pepper noise in the images.

Median filters is widely used in digital image processing field. It can remove the noise and keep the edges on the images at the same time. Thus we can get a reasonable recovery from a "dirty" input image. Also it is very convenient in practice since we do not need the statistical features of images in computation process.
Our project is focusing on the fast median filtering algorithm since performance is one of the key considerations for processing digital images nowadays. No matter how the computation devices can be developed fastly, to find and apply a fast algorithm to solve the problem is always important in practical use. Since we are interested in verifying the speedup we can get from the fast median algorithm, we also implemented the general version of median algorithm to compare its performance with the fast version. We call it naive median filter in this paper.

So with the same input image and the same size of filter window, we can do the tests for the naive median filter and fast median filter and check the performance speedup we get from the fast version. More details on comparison results are covered in Section 3.3.

The third filtering is called bilateral filtering. Bilateral filter is also an edge-preserving method and de-noise smoothing filter like median filters.The reason that we also implemented bilateral filtering algorithm in our application is that we want to compare the effect after processing with median filtering algorithms and the approximate time it would take to achieve a similar result from the median filtering algorithms. The idea underlying bilateral filtering algorithm is to do in the range of any image what traditional filters do in its domain. Two pixels can be close to each other occupy nearby spatial location, or they can be similar to one another which means they have nearby values in a perceptually meaningful fashion.

2. IMPLEMENTATION

2.1 Naive Median Filter

The basic idea of naive version median filter is we can set up an square area which we call it as filter window in this paper. The size of this filter window can be specified as any two equal odd values. Some typical values could be 3x3 pixels, 5x5 pixels and 9x9 pixels and so on. Then we can run through this window on step each time by a pixel. The value of the pixel at the central position in the filter window is replaced with the median of its neighboring pixels' values. For simplicity purpose, we do not consider the filter window size with odd values in our application.

For those pixels on the four edges of an 2D image, we ignore the boundary issue for such pixels. Since this is a trivial issue as to our test experiments. This means that when the filter window size is relatively big, the edges of the output images will remain the same and we can tell that the display on the four edges are quite different from the processed part in the image. The pseudo code for this naive median filter is represented as below.

```
allocate outputPixelValue[image width][image height]
edgex := (window width / 2) rounded down
edgey := (window height / 2) rounded down
for x from edgex to image width - edgex
    for y from edgey to image height - edgey
        allocate colorArray[window width][window height]
        for fx from 0 to window width
            for fy from 0 to window height
                colorArray[fx][fy] := inputPixelValue[x + fx - edgex][y + fy - edgey]
        sort all entries in colorArray[][]
        outputPixelValue[x][y] := colorArray[window width / 2][window height / 2]
```

Figure 1: Median filter pseudo code for 2D images (refer to [3])

This pseudo is used for processing 2D images. From the pseudo code, we see that first we set the size of the filter window, and get the scope of the pixels we are going to compute new values excluding those pixels near the four edges of the image. Then we use this window to run through the pixels of this scope and store the neighboring pixel values into an 2D array. Next we can simply sort the values in this array at ascending order and pick out the median

from it to replace the value of pixel at the central position of the filter window.

Based on this idea, we can say the algorithm itself is very straightforward but it is not very efficient due to the redundant computations for most of the neighboring pixel values especially when the size of filter window is very big. In order to overcome the inefficiency of this median filter algorithm, we are interested in finding a better solution to solve the same problem. And the method we used is called fast median filter.

2.2 Fast Median Filter

Figure 2 shows the pseudo code for this algorithm.

Input: Image X of size $m \times n$, kernel radius r
Output: Image Y of the same size as X
  Initialize kernel histogram H
  for $i = 1$ to m **do**
    for $j = 1$ to n **do**
      for $k = -r$ to r **do**
        Remove $X_{i+k,j-r-1}$ from H
        Add $X_{i+k,j+r}$ to H
      end for
      $Y_{i,j} \leftarrow \text{median}(H)$
    end for
  end for

Figure 2: pseudo code for Huang's algorithm (refer to [1])

As we see, this algorithm creates the histogram for each pixel value initially and add 1 to the occurrence of each value that resides inside the window. Then, we need to slide the window horizontally and update the histogram for the occurrence of values inside the window. Moreover, we will compute the median value according to the histogram for each time of sliding the window and replace the central pixel values inside the window with that median.

Figure 3 and Figure 4 below are the examples to show which part really needs to be updated when sliding the window. In this case, we can see that when right sliding 1 slot in Figure 3, the blue column is left outside the window and all corresponding values in that column to the histogram should decrement by 1 and the occurrence of all values in red column should increment by 1 since they fall inside the window.

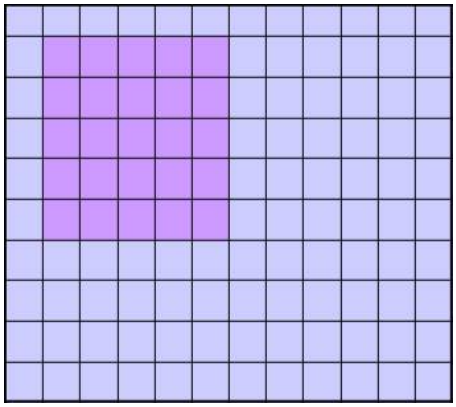


Figure 3

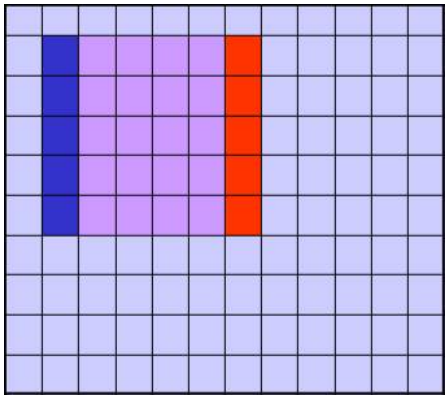


Figure 4

Here is the more concrete example presented in class to give more details about how this algorithm is implemented:

Step 1: compute the center of the window for median th = mn / 2 (m, n are the window width and height)

Step 2: Put the window (m x n) at the beginning of a new row. Construct a histogram **H**, determine the median **med** , **lt_med**=|{(i,j): I(i,j)< med}| (lt_med store the number of values lower than med)

Step 3: For each pixel p in the leftmost column: H[I(p)]=H[I(p)]-1 if I(p)<med then lt_med=lt_med -1

Step 4: Move the window one column right. For each pixel p in the rightmost column: H[I(p)]=H[I(p)]+1 if I(p)<med then lt_med=lt_med +1

Step 5: If (lt_med >**th)** then

```
repeat
 med=med-1;  lt_med = lt_med - H[med];
until   lt_med<=th
else
  while (lt_med+H[med]<=th) do
    lt_med = lt_med + H[med]; med=med+1;
```

Step 6: If the right-hand column of the window is not at the right-hand edge of the image, go to step 3

Note: when the right-hand column hits the right boundary of the image, move the window down and then slide the window from right to left, and when the window hits the left boundary of the image, we move down the window and then slide it from left to right, and so forth.

2.3 Bilateral Filter

Bilateral filter extends the method of Gaussian smoothing by weighting the filter coefficients with their corresponding relative pixel brightness. For those pixels which are quite different in brightness from the central pixel are weighted less even if they might be close to the central pixel value.

This is actually a convolution with a nonlinear Gaussian filter with weights based on pixels brightnesses. Thus we need two Gaussian filters at a localized pixel neighborhood. One for the spatial domain and the other one in the intensity domain. And we call them domain filter and range filter respectively. The details on computation equations are given in course slides.

Since bilateral filter algorithm is not the important implementation of our application, we just implement a relatively simple version to apply on input images.

3. EXPERIMENTAL RESULTS

3.1 Usage of the Application

In this subsection we show the GUI of our application which is developed for both verifying the correctness of the three algorithms and collecting the timing data for performance analysis use. This application, MedianFilter Test, is developed under Microsoft Visual Studio 2010 IDE in C#.

The following figures are the snapshots from the screen which illustrate how to successfully use this application to test any input 2D images in steps.

Step 1: Highlight the old values (3X3 as default set if skip this step) at the ""window size" tag and replace them with the new odd values you like. Make sure that the two new values are equal as we only consider our filter window (kernel) as n by n size. See Figure 5.

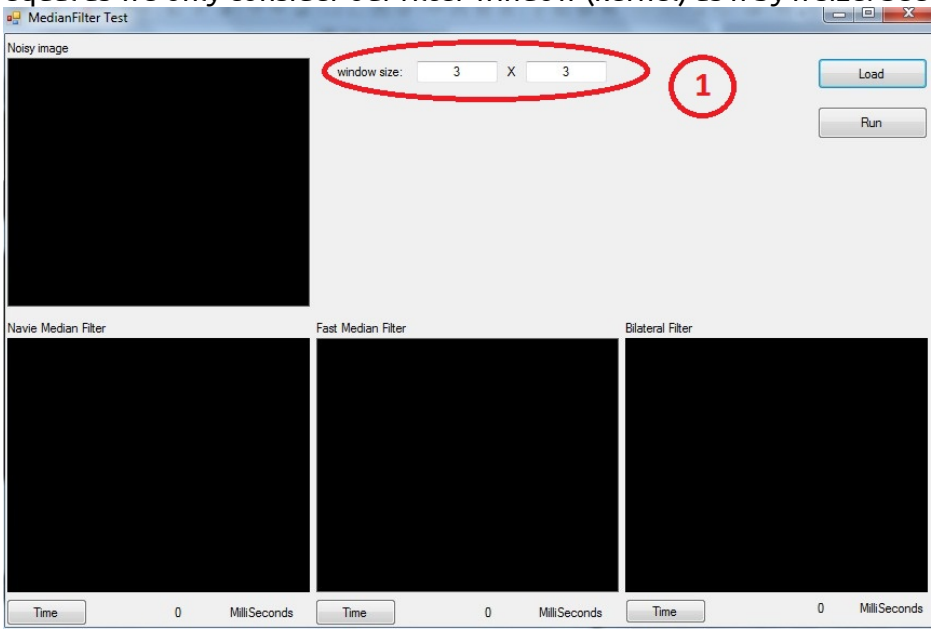


Figure 5: default filter window size 3 X 3

Step 2: Click on the Load button on the upper right corner and open a “noisy” image from disk. After loading the image as input, we can see it is displayed in the picturebox under tag “Noisy image”. See Figure 6.

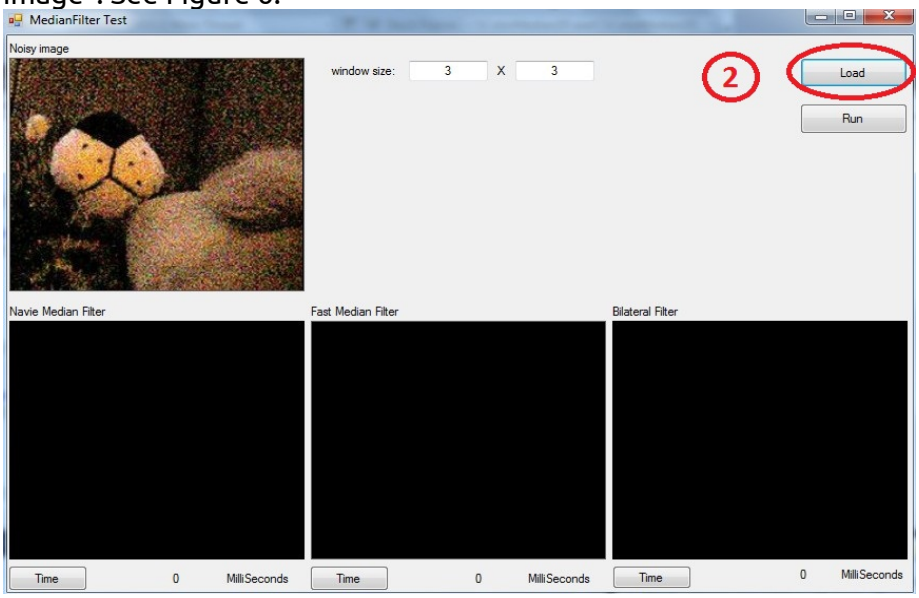


Figure 6: Load an unprocessed 2D image as input data

Step 3: Click on the Run button to execute the three filtering algorithms and output the results shown in the picture boxes at the bottom respectively. It might take a while depends on the parameter we set.

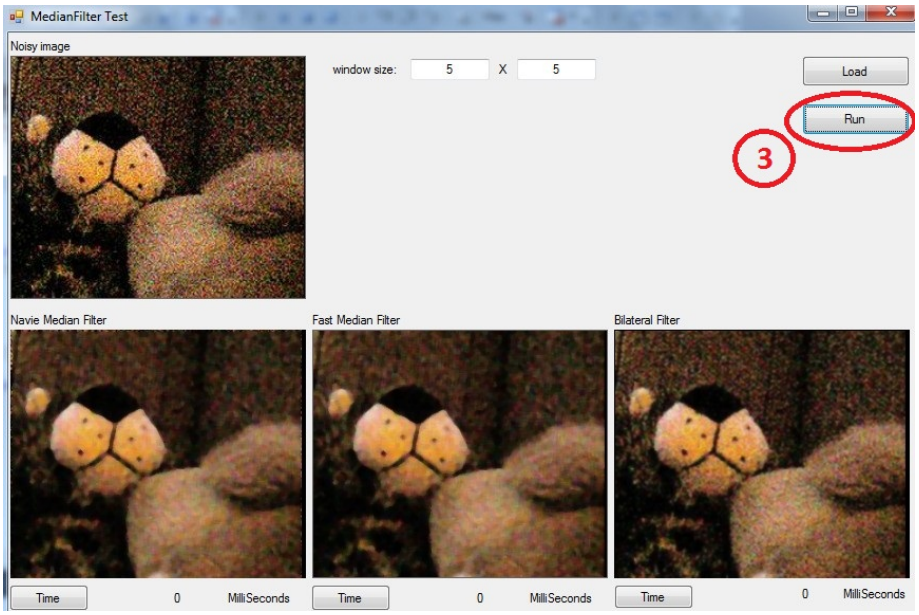


Figure 7: Run the program and have the output images displayed

Step 4: Click on any one of the three Time Buttons if we want to see the eclipsed time for processing the input image by a certain filtering algorithm.

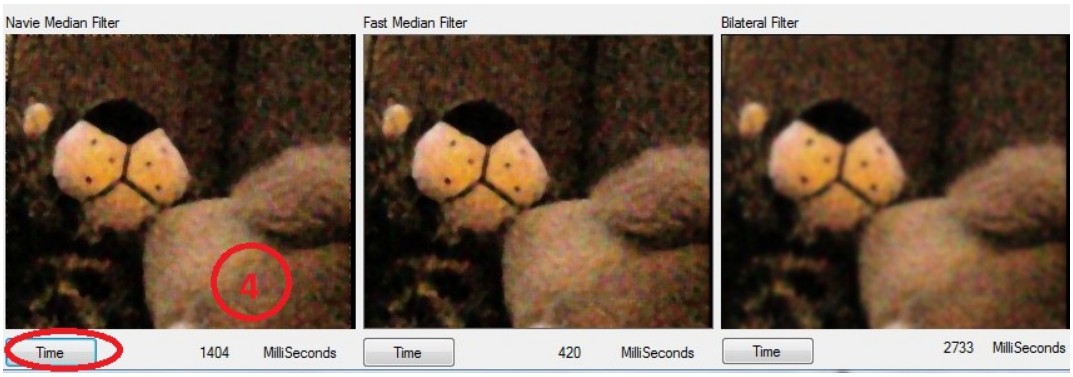


Figure 8: Timing results for executing each algorithm

To start processing a new input image, we need to start over the application to repeat the same steps as listed above.

3.2 Sample Outputs

The output results can be observed directly from the application window. The figure below is an example of what the output would be looked like.

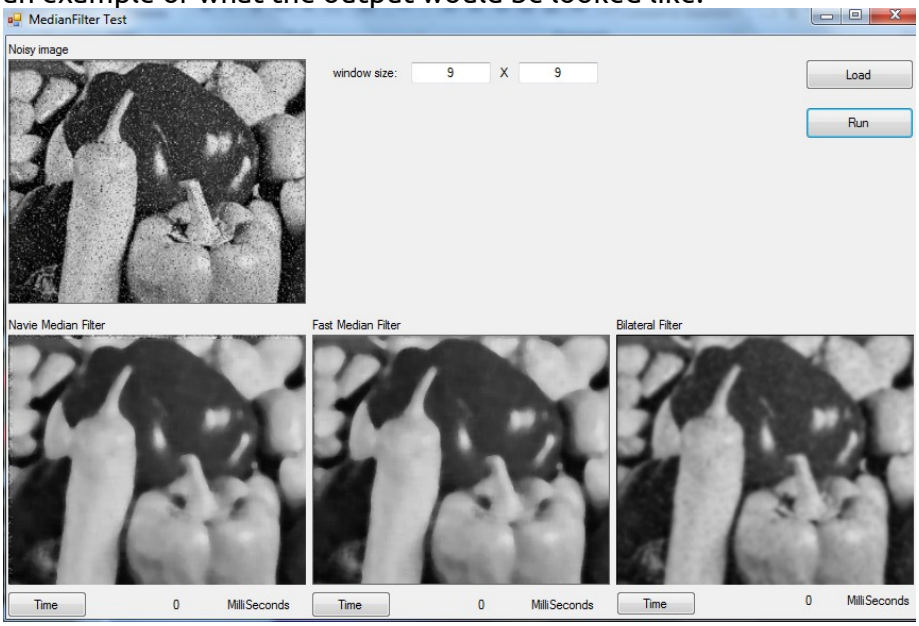


Figure 9: Output results for a 312x312 pixels gray scale image with 9x9 filter window

We have tested a set of different 2D images including both gray scale images and color images. The following figures show that the effects of the output images after applying the filters. The first image is a gray scale image and the other two are color images with different sizes. All of the three different filters were working correctly and we get the expected results on de-noising for those input images.

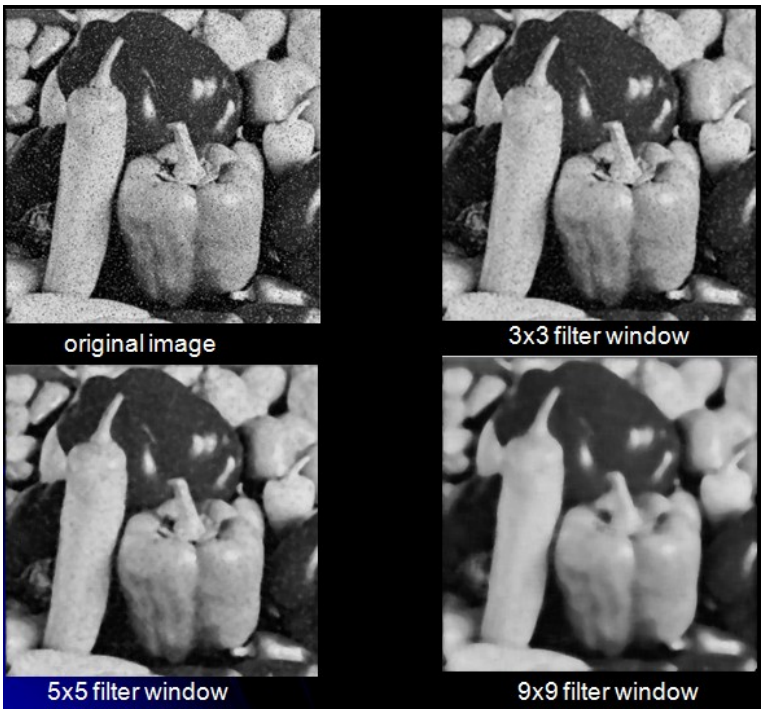


Figure 10: De-Noising results with different window sizes for a 312x312 grayscale image

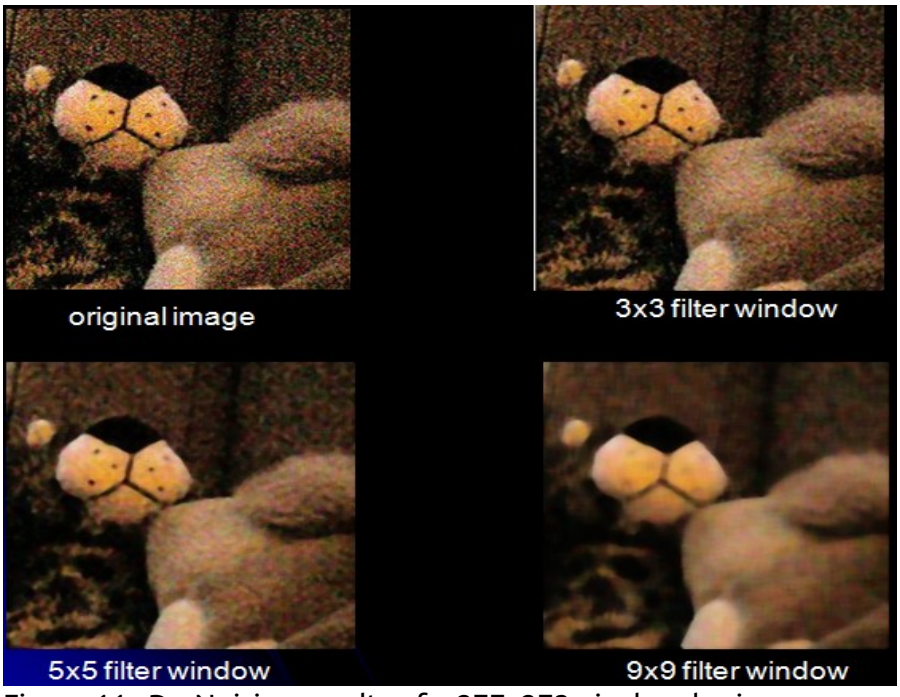
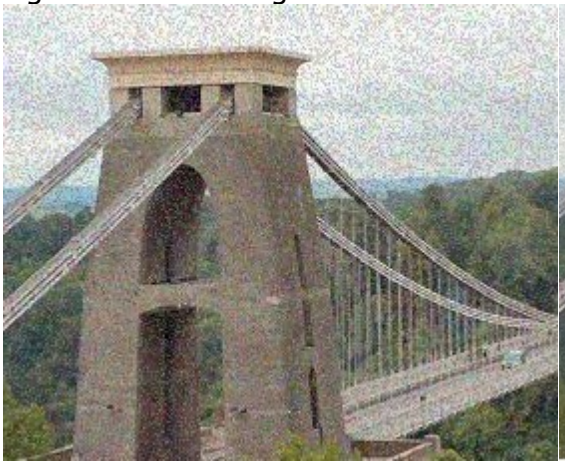


Figure 11: De-Noising results of a 277x273 pixels color image

Figure 12: De-Noising results of a 460x350 pixels color imageswith different window sizes


Original image


SigmaS=6 SigmaR = 3


SigmaS=6 SigmaR = 6


SigmaS=6 SigmaR = 12

Figure 13: The output images by applying bilateral filter algorithm

3.3 Timing Results and Complexity Analysis

For input image as shown in Figure 11, we get the timing data for naive version and fast version median filter algorithms. The following tables show the comparison data for them. All time unit in these tables use milliseconds.

Filter Window Size	Naive Median Filter	Fast Median Filter
3 x 3	279	138
5 x 5	1890	369

For input image as shown in Figure 10. we get the timing data for naive version and fast version median filter algorithms.

Filter Window Size	Naive Median Filter	Fast Median Filter
5x5	1275	264
9x9	1755	440

The naive version of median filter simply builds a list of the pixel values in the filter window and then sorts these values. The median is the value located at the center of the list. For most general cases, the upper bound of the time complexity is O(r^2*logr) where r is the radius of the filter window.

As discussed in fast median filter algorithm, we do not need to process mostly neighboring pixels as we did for naive version. This is the key observation to reduce the complexity in practice. The fast median algorithm we implemented in our application is similar to Huang's median filtering. Thus we can expect a complexity of O(n) where n is the number of pixels in

an image. The improvement from the fast median algorithm is much better after using this algorithm and we are assured that the algorithm is implemented correctly in the end.

4. CONCLUSION

In this paper, the implementations of three common filter techniques integrated in our application have been described and tested by several gray and color noise images, we can conclude the result as follows:

a. As the window size becomes larger and larger, the result images after de-noise will become more and more blurry and time-consuming.

b. Compared with the naive median filter, fast median filter proposed by Huang is roughly 3 times faster.

c. Bilateral filter is the most time-consuming algorithm among these three implementations. Moreover, as the spatial distance and intensity distance values vary, it can be worse.

REFERENCE

[1]. **Median Filtering in Constant Time**. Perreault, S. and Hebert, P. IEEE Transactions on Image Processing. 2007

[2]. **Fast Median and Bilateral Filtering**. Ben Weiss. ACM Transactions of Graphics (Proceedings of the ACM SIGGRAPH'06 Conference). 2006

[3]. Median Filter Wiki page: http://en.wikipedia.org/wiki/Median_filter

[4]. Maheswari, D., and Dr. V. Radha. "**NOISE REMOVAL IN COMPOUND IMAGE USING MEDIAN FILTER.**" International Journal on Computer Science and Engineering 2.4 (2010): 1359-362.

[5]. **Bilateral Filtering for Gray and Color Images**. C. Tomasi and R.Manduchi. IEEE ICCV 1998

[6]. T. Huang, G. Yang, and G. Tang, "**A Fast Two-Dimensional Median Filtering Algorithm**," IEEE Trans. Acoust., Speech, Signal Processing, vol. 27, no. 1, pp. 13–18, 1979.

[7]. DURAND, F. AND DORSEY, J. 2002. **Fast Bilateral Filtering for the Display of High-Dynamic-Range Images**. ACM SIGGRAPH 2002.

[8]. KABIR, I. 1996. **High Performance Computer Imaging**. Greenwich, CT. Manning Publications. pp. 181-192.

[9]. WEISS, B. 2006. **Method and Apparatus for Processing Image Data**. US Patent 7,010,163.