PROJECT REPORT

On

Wavelet and PDE based approach for Image and Video Scaling and De-noising

Supervised By:

RAJEEV SRIVASTAVA
Associate Professor
IT-BHU, Varanasi

Submitted By:

RAHUL JAIN
07020007
IDD PART – IV

SABYA SACHI
07020004
IDD PART – IV

SAKET JALAN
07020006
IDD PART - IV

DEPARTMENT OF COMPUTER ENGINEERING

IT-BHU, VARANASI

DECEMBER 2010

DEPARTMENT OF COMPUTER ENGINEERING
INSTITUTE OF TECHNOLOGY
BANARAS HINDU UNIVERSITY
VARANASI-221005, (U.P.)
INDIA

Date:

CERTIFICATE

This is to certify that **Rahul Jain, Sabya Sachi and Saket Jalan,** students of Department of Engineering, Institute of Technology BHU, Varanasi have worked on the topic **"Wavelet and PDE based approach for Image and Video Scaling and De-noising"** under my direct supervision and guidance for their B.Tech Major Project, the findings of which have been incorporated in this report. They have worked diligently, meticulously and methodically on the project. The report submitted embodies the original work done by them on the project and is approved for submission.

RAJEEV SRIVASTAVA

Associate Professor
IT-BHU, Varanasi

ACKNOWLEDGEMENT

We would like to convey our deepest gratitude to **Mr. Rajeev Srivastava**, Associate Professor, Department of Computer Engineering, Institute of Technology BHU, who guided us through this project. His keen awe-inspiring personality, superb guidance, and constant encouragement are the motive force behind this project work.

We would also like to thank **Prof. A.K. Agrawal**, Head of the Department, Department of Computer Engineering, Institute of Technology BHU, for allowing us to avail all the facilities of the Department necessary for this project.

We are very thankful to all the technical and non-technical staffs of the Department of Computer Engineering for their assistance and co-operation.

RAHUL JAIN	**SABYA SACHI**	**SAKET JALAN**
07020007	07020004	07020006
IDD PART – IV	IDD PART – IV	IDD PART - IV

Department of Computer Engineering

Institute of Technology

Banaras Hindu University, Varanasi - 221005

ABSTRACT

Scaling is the process of resizing the input signal, video or image. Video files consist of multiple frames that are displayed one by one in transition. Scaling of video files is done by scaling all of its frames to the desired resolution.

There are various methods of scaling image files like bilinear, bicubic, nearest-neighbor etc. which are used in scaling image/video files. During scaling they add some noise (unwanted data having no meaning) in the resultant image, which becomes more visible as the scaling factor is increased which reduces its quality.

In this project we study different scaling algorithms like bilinear, nearest-neighbour, bicubic, and lanczos (sinc 3) with both static as well as dynamic (from video) images for different resolutions. We also apply Wavelet and PDE based methods to de-noise the resultant images and compare various results using different parameters.

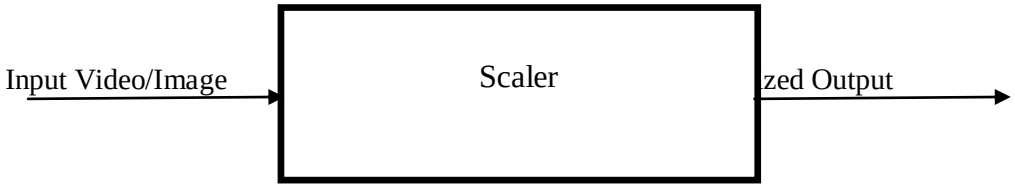
Various results have been compared using different parameters such as Peak Signal to Noise Ration (PSNR), Mean Square Error (MSE), Correlation Parameter (CP) and Mean Structure Similarity Index Map (MSSIM) between ideal output image and the resulting output with these methods.

TABLE OF CONTENTS

1. Introduction

1.1) Video and Image Scaling

Video/Image Scaling is the process of converting the image or video signals from one resolution or size to another. Applying scaling to a video involves capturing frames from the video, one by one, resizing the images and putting them back together. The method focuses on just scaling the video, not preserving the audio.



The input image must be provided in raster scan format (left to right and top to bottom). The valid outputs will also be given in this order.
Video scalars are often combined with other video processing devices or algorithms to create a video processor that improves the apparent definition of video signals. These other devices may include the ability for:

- Aspect ratio control
- Frame rate conversion
- Color point conversion
- Detail enhancement
- Edge enhancement
- brightness/contrast/hue/saturation/sharpness/gamma adjustments
- motion compensation
- color space conversion

1.2) Purpose

Video Scaling is used in large number of products nowadays: various music players, DVD players, computer monitors, video editing and broadcasting equipments etc. There are two main criteria for any scaling algorithm:

a. Efficiency of algorithm
b. Quality of the scaled result

Quality of the image is the measure of amount of noise, smoothness and sharpness of the resulting image. As the scaling factor increases the quality of the scaled image decreases as the pixels of the original image become more and more visible.
Decreasing the time a method takes to scale an image also decreases the quality of the scaled image. So, an algorithm has to consider a tradeoff between the efficiency and the quality of the algorithm.
Some errors associated with the scaling process are:

a. **Posterization**: Posterization of an image entails conversion of a continuous gradation of tone to several regions of fewer tones, with abrupt changes from one tone to another.

b. **Ringing:** Ringing artifacts are artifacts that appear as spurious signals ("rings") near sharp transitions in a signal. Visually, they appear as "rings" near edges; audibly, they appear as "echos" near transients.

c. **Aliasing:** Aliasing is the effect in which the image appears jagged at places.

1.3) Scope

Different scaling algorithms produce different quality of image. Scaled images contain errors like noise which is more visible for large scaling factors. To further analyze the scaled image de-noising algorithms can be used to remove the noise from them and further improve the quality of image. There are various de-noising techniques like partial differentiation based method or wavelet based method that can be employed on the scaled images.

1.4) Overview

This report in further sections describes the various things involved with this project strating from the introduction of video and image files. It then describes scaling algorithms bilinear, bicubic, nearest-neighbour and lanczos (sinc3) and the denoising methods using partial differentiation and wavelets along with their algorithms.
At the end we compare various methods using different parameters like PSNR (Peak Signal to Noise Ratio), CP (Correlation Parameter), MSE (Mean Square Error) and MSSIM (Mean Structured Similarity Index Map). The comparison has been done between different algorithms, for different scaling factors.

2. Literature Survey

2.1) Video Files

Video files are the sequence of still images representing scenes in motion. In the context of the video, these images are called frames. The term video commonly refers to several storage formats for moving pictures: digital video formats, including Blu-ray Disc, DVD, QuickTime, and MPEG-4; and analog videotapes, including VHS and Betamax. Video can be recorded and transmitted in various physical media: in magnetic tape when recorded as PAL or NTSC electric signals by video cameras, or in MPEG-4 or DV digital media when recorded by digital cameras.

Characteristics of video files[5]

- **Frame Rate:** It is defined as the number of still frames per second, ranges from six or eight frames per second (*frame/s*) for old mechanical cameras to 120 or more frames per second for new professional cameras. ranges from six or eight frames per second (*frame/s*) for old mechanical cameras to 120 or more frames per second for new professional cameras.

- **Interlacing:** Interlacing was invented as a way to achieve good visual quality within the limitations of a narrow bandwidth. The horizontal scan lines of each interlaced frame are numbered consecutively and partitioned into two fields: the odd field (upper field) consisting of the odd-numbered lines and the even field (lower field) consisting of the even-numbered lines. In *progressive scan* systems, each refresh period updates all of the scan lines. The result is a higher spatial resolution and a lack of various artifacts that can make parts of a stationary picture appear to be moving or flashing.

- **Display Resolution:** The size of a video image is measured in pixels for digital video, or horizontal scan lines and vertical lines of resolution for analog video. The display resolution is represented as a x b where a is the number of pixels in a horizontal line in digital video or the number of vertical lines in analog video and b is the number of pixels in a vertical column in digital video or number of horizontal lines in analog video.

- **Aspect ratio:** Aspect ratio describes the dimensions of video screens and video picture elements. All popular video formats are rectilinear, and so can be described by a ratio between width and height. The screen aspect ratio of a traditional television screen is 4:3, or about 1.33:1. High definition televisions use an aspect ratio of 16:9, or about 1.78:1. The aspect ratio of a full 35 mm film frame with soundtrack (also known as the Academy ratio) is 1.375:1.

- **Color Space and Bits per pixel:** The number of distinct bits required to represent the pixel intensity is the number of bits per pixel. It determines the number of distinct colors that a pixel can represent.

- **Video Quality:** Video quality can be measured with formal metrics like PSNR or with subjective video quality using expert observation. Many subjective video quality methods are described in the ITU-T recommendation BT.500. One of the standardized method is the *Double* Stimulus Impairment Scale.

- **Video Compression Methods:** A wide variety of methods are used to compress video streams. Video data contains spatial and temporal redundancy, making uncompressed video streams extremely inefficient. Broadly speaking, spatial redundancy is reduced by registering differences between parts of a single frame; this task is known as intra-frame compression and is closely related to image compression. Likewise, temporal redundancy can be reduced by registering differences between frames; this task is known as inter-frame compression. Only Digital videos can be compressed.

- **Bit Rate:** Bit rate is a measure of the rate of information content in a video stream. It is quantified using the bit per second (bit/s or bps) unit or Megabits per second (Mbit/s). A higher bit rate allows better video quality.

- **Stereoscopic:** *Stereoscopic video* can be created using several different methods:
  - Two channels — a right channel for the right eye and a left channel for the left eye.
  - One channel with two overlaid color coded layers.
  - One channel with alternating left/right frames for each eye.

2.2) Images

Digital image is representation of a two dimensional images using ones or zeroes. Depending on whether or not the image resolution is fixed, it may be of vector or raster type. Vector images store the lines, slopes and contours of the images as mathematical formula. So, their resolution isn't fixed.
Raster image, on the other hand have a finite set of digital values, called *picture elements* or pixels. The digital image contains a fixed number of rows and columns of pixels. Pixels are the smallest individual element in an image, holding quantized values that represent the brightness of a given color at any specific point.
Each pixel of a raster image is typically associated to a specific position in some 2D region, and has a value consisting of one or more quantities. Based on these quantities, raster images can be binary, grayscale, color etc.
Binary and the grayscale images are both black-and-white images. Both store a single value for a pixel but the difference between them is that the binary images can only two values for a position signifying either black or white but the grayscale images have their pixels colored in the shades of gray with black as the weakest intensity level and white as the strongest intensity level.

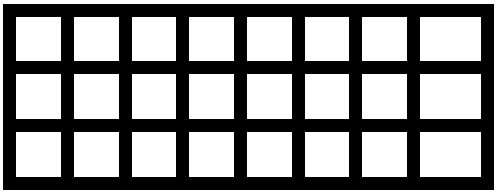
23	34	234	23	67
45	0	255	0	24
34	67	45	89	34
87	4	5	33	23
45	23	34	45	67

Intensity matrix of a grayscale image

Color images include color values for the pixels. For the visually acceptable results, at least three intensity values are needed for each pixel. Color images acan be RGB(Red, Green, Blue), CMYK(Cyan, Magenta, Yellow, Black) etc. In RGB images, the pixels have three intensity values for each pixel in the image, each value denoting a single primary color of Red, Green or Blue. Usually the intensity values for each pixel in a RGB image is represented

by 24 bit with 8 bits representing red, 8 bits representing Green and 8 Bits representing blue. Thus there will be 256 shades for each of the colors which when mixed give 256 x 256 x 256 total number of colors available for each pixel.

A pixel of a 24-bit image can be depicted as:



Within each cell there is a binary value either 0 or 1.

Characteristics of images

- **Number of channels:** It defines the number of intensity values a pixel contains. A grayscale image has only one channel while a RGB image has 3 channels.
- **Depth:** It is the number of bits used to define an intensity value of a channel. It can vary from being 2-3 bits to even 10 bits. More the depth of image, better the image can be represented.
- **Color Model:** Color model describes the color representation, eg RGB, CMYK etc.
- **Size(Resolution):** Image size is represented by number of pixels in horizontal and a vertical line and is represented as a x b.

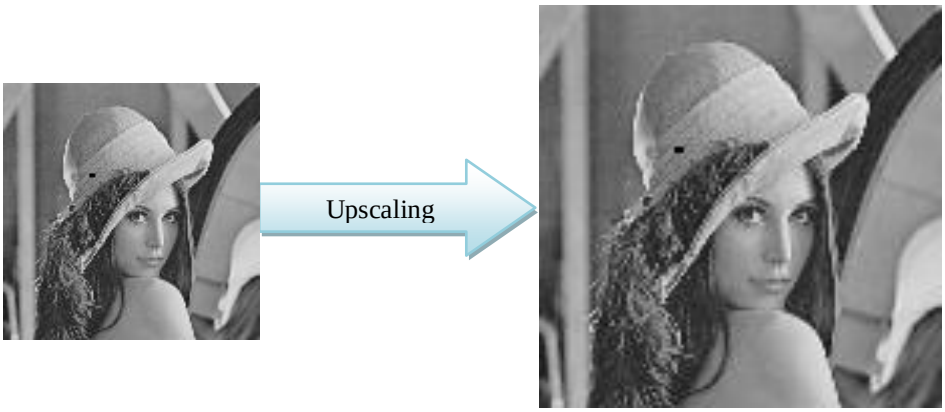
2.3) Scaling

Video Scaling is the process of converting the video signals from one resolution or size to another. Video scaling can be performed as:

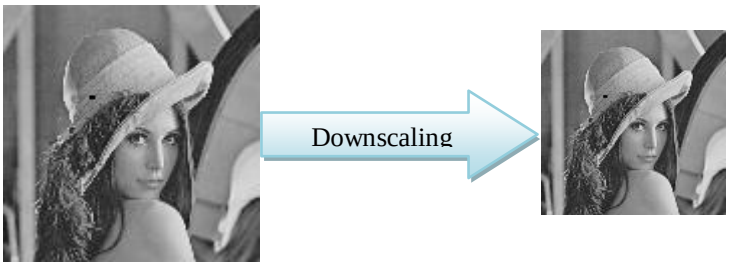
```
While(there is a frame remaining for scale)
        Capture the frame;
        Resize it using appropriate algorithm;
        Integrate it into a new video file;
End while
```

Scaling can be of two types:

- **Upscaling:** Upscaling is the scaling type in which the image(video) is scaled from a lower resolution to higher resolution



- **Downscaling:** Downscaling is the scaling type in which the image(video) is scaled from a higher resolution to lower resolution



2.4) Scaling Methods

There are various methods available for scaling. Some of the most important once are

- **Nearest Neighbour:** Nearest neighbor technique also known as point sampling algorithm finds the pixel of the original image which is nearest to the pixel for which the intensity value is to be calculated and then assigns the former's value to the later. To do this, it first matches the original images to the zoomed one and then checks which pixel is the nearest one; in case there is a conflict, topmost or leftmost pixel is chosen. This method yields the fastest result but also the quality of result is very bad when the scaling factor increases.

- **Bilinear Method:** Bilinear method is the extension of linear interpolation on two variables and is commonly used to interpolate functions of two variables. An image can also be considered as a function 'intensity' of two variables 'width' and 'height'. The idea is to perform the linear interpolation first in horizontal direction and then in vertical direction or vice-versa.

- **Bicubic Method:** Bicubic method is the extension of cubic interpolation to two variables. The zoomed surface is smoother and has less interpolation effects than corresponding surfaces obtained by bilinear method or nearest-neighbor method but it is also slower. So, it is favoured over the other when the speed of the process doesn't matter. As shown below, the value an unknown pixel is found from the surrounding sixteen pixels.

- **Lanczos method:** The Lanczos filter is a windowed form of the sinc filter, a theoretically optimal "brick-wall" low-pass filter. The sinc function is infinite in extent, and thus not directly usable in practice. Instead, one uses approximations, called windowed forms of the filter. The Lanczos kernel indicates which samples in the original data, and in what proportion, make up each sample of the final data.

Examples:

Original Image

Zoomed in by Nearest Neigbour

Zoomed in by Bilinear

Zoomed in by bicubic

Zoomed in by Lanczos

2.5) Noise

Noise is any type of unwanted brightness or color information in images. It is generally formed when an image is upscaled from a lower resolution to higher resolution. Various types of noises that can form when an image is upscaled are:

- **Aliasing:** When a digital image of very low resolution is interpolated to a high resolution, then the reconstructed image will differ from the original image, and an alias is seen. An example of **spatial aliasing** is the Moiré pattern.[9]

An Aliased image

- **Posterization:** Posterization of an image entails conversion of a continuous gradation of tone to several regions of fewer tones, with abrupt changes from one tone to another.[7]

Posterized image

- **Ringing artifact:** Ringing artifacts are artifacts that appear as spurious signals ("rings") near sharp transitions in a signal. Visually they appear as rings in images and hence get the name.[8]

A ringing artifact in image

2.6) De-noising using PDE (Partial Differential Equation):[2][4]

A generic zooming algorithm takes as input an RGB picture and provides as output a picture of greater size preserving the information content of the original as much as possible. Unfortunately, the normal methods mentioned above ,though preserve the low frequency content of an image, but do not well to preserve the high frequencies so that the final image is of as good a quality as the original one. Especially, when the image is zoomed by a large factor, the zoomed image looks very often blocky.

In recent years, PDE based methods have made great strides in the field by introducing the concept of diffusion. Diffusion is a process that equilibrates concentration differences without creating or destroying mass. In image processing, intensity can be considered equivalent to mass.

There are mainly two types of diffusion equations- isotropic and Anisotropic. Isotropic diffusion method though reduces noise and smoothes the structure, is unable to preserve the edges. The non linear PDE equations implementing anisotropic diffusion, yield intra-region smoothing, not inter-region smoothing, by impeding diffusion at the image edges. Thus the edges are preserved. Various orders of differential equation have been suggested to apply anisotropic diffusion on noisy images.

2.7) De-noising using Wavelets:

2.7.1) Wavelets: [18]

A wavelet is a wave like oscillation that starts at zero, increases and then again decreases back to zero. They are a mathematical tool which divides data into different frequency components and then analyse them based upon its scale. They are used to extract important information from the audio signals and images. It consists of a set of non-linear bases. When projecting a function in terms of wavelets, basis functions are chosen according to the function (unlike in fourier analysis where we use sine and cosine) being approximated to represent it in most efficient way.

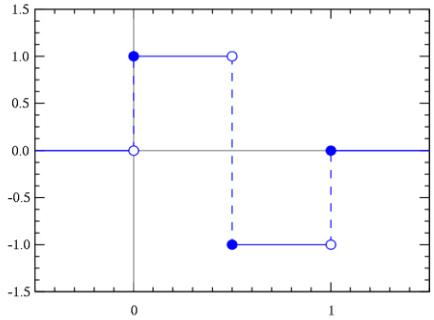
2.7.2) Wavelet Basis Functions:

In Fourier transform we use sine and cosine functions to transform data. In wavelet analysis we use basis functions to represent our data. These functions are usually orthogonal to each other so that the process is invertible. Basis functions vary in scale by chopping the same function using different scale sizes. For example: we can divide our signal using 2 step signal into 0 to 1/2 and 1/2 to 1. Again, we can divide using 4 step signal into 0 to 1/4, 1/4 to 1/2, 1/2 to 3/4 and 3/4 to 1, each represnting the signal with different resolution or scale. Different wavelets have different set of basis

functions for eg: symlet, biorthogonal, haar.

2.7.3) Wavelet Analysis:

2.7.3.1) Haar Wavelet Transform:



Haar wavelet is the simplest possible wavelet. It consists of square shaped functions which together form the wavelet basis functions. Wavelet transform like Fourier transform represents the input data using the wavelet basis functions. Using Haar wavelet, haar basis functions are used for transforming the data.

Basis functions are orthonormal to each other, and thus reverse transform will reconstruct the original image.

The 2x2 Haar matrix is given by:

$$H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

Haar transform is derived from the Haar matrix. An example of 4x4 Haar matrix is:

$$H_4 = \frac{1}{\sqrt{4}} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ \sqrt{2} & -\sqrt{2} & 0 & 0 \\ 0 & 0 & \sqrt{2} & -\sqrt{2} \end{bmatrix}$$

As seen, first and second row form a orthonormal basis pair, while 3rd and 4th row form another such pair.

Image after 2 level Haar Wavelet Transform

Using any of the basis pairs, the 1-D transformation can be applied by moving these filters through the data. For 2-D transformation, 1-D transformation is first applied to each row, and then to column at each level. Reconstruction of image is performed in just the reverse order of transformation at each level.

2.7.3.2) De-noising:

De-noising is performed either by using hard thresholding or by soft thresholding. If λ represents the threshold value for the input data.[17]

In hard thresholding it selects the coefficients greater than threshold and sets others to zero:

$$f_h(\mathbf{x}) = \begin{cases} \mathbf{x}, \text{ if } |\mathbf{x}| \geq \lambda \\ 0, \text{ otherwise.} \end{cases}$$

In soft thresholding, the wavelet function shrinks the wavelet coefficients towards zero:

$$f_s(\mathbf{x}) = \begin{cases} \mathbf{x} - \lambda, \text{ if } \mathbf{x} \geq \lambda \\ 0, \quad |\mathbf{x}| < \lambda \\ \mathbf{x} + \lambda, \text{ if } \mathbf{x} \leq -\lambda. \end{cases}$$

There are various methods of selecting the threshold value λ. If the value of λ is not given, [17][16]

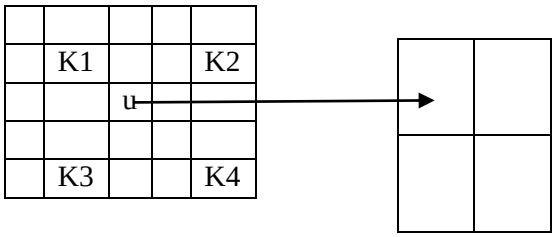
$\lambda = \sigma * \sqrt{2}*\log(n)/n$

where σ = Median of coefficients/(0.6745*n)

3. Implementation

3.1) Scaling: Nearest Neighbor Method

Nearest Neighbour chooses the nearest possible known pixel intensity value and then gives its value to the pixel whose intensity value is unknown. To find the nearest pixel, it just scales the current pixel down to the original zoom level and then finds the integer value closest to it.



Zoomed image **Original image**

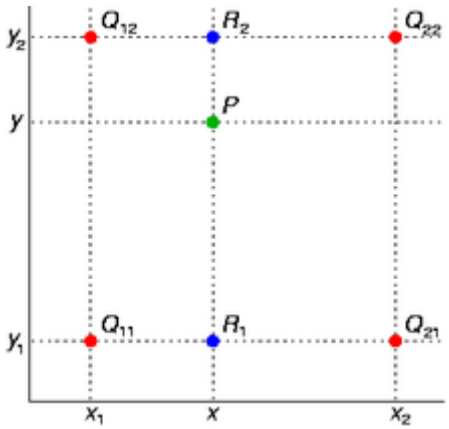
K1, K2, K3, K4 are the known pixel values, u lies nearest to K1 and hence the u is give the intensity value of K1.

width-> width of original image
height-> height of original image
hfactor->horizontal scaling factor
vfactor-> vertical scaling factor
nearestX,nearestY-> nearest pixel in original image to current pixel in zoomed image.

nearestX= X/vfactor
nearestY= Y/hfactor
zoomed_image[X,Y]=original_image[nearestX,nearestY]

3.2) Scaling: Bilinear Method

In this method, first the interpolation is done in horizontal direction and then in vertical direction.



To find the value of the unknown function f at the point $P = (x, y)$,when it is assumed that we know the value of f at the four points $Q_{11} = (x_1, y_1)$, $Q_{12} = (x_1, y_2)$, $Q_{21} = (x_2, y_1)$, and $Q_{22} = (x_2, y_2)$.

First in horizontal direction, the values are calculated and then the average is taken in vertical direction. To calculated the unknown pixel's intensity value.

Horizontal Interpolation

$$f(R_1) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21})$$

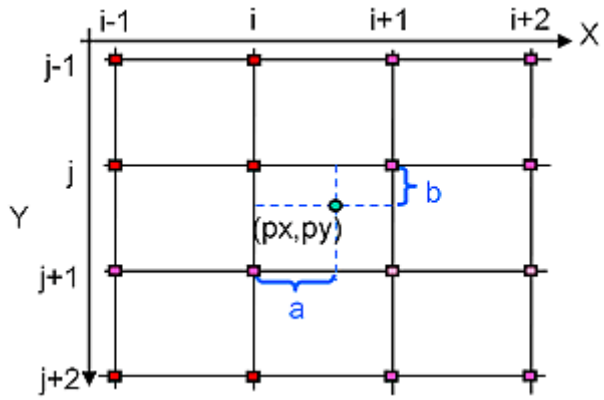
$$f(R_2) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22})$$

Vertical Interpolation

$$f(P) \approx \frac{y_2 - y}{y_2 - y_1} f(R_1) + \frac{y - y_1}{y_2 - y_1} f(R_2).$$

3.3) Scaling: Bicubic Method

In this method, the intensity value for a pixel is calculated from the surrounding sixteen pixels as shown below:



The interpolated surface is written as :

$$p(x, y) = \sum_{i=0}^{3} \sum_{j=0}^{3} a_{ij} x^i y^j.$$

The interpolation problem consists of determining the 16 coefficients a_{ij}. Matching $p(x,y)$ with the function values yields four equations,

1. $f(0,0) = p(0,0) = a_{00}$
2. $f(1,0) = p(1,0) = a_{00} + a_{10} + a_{20} + a_{30}$
3. $f(0,1) = p(0,1) = a_{00} + a_{01} + a_{02} + a_{03}$
4. $f(1,1) = p(1,1) = \sum_{i=0}^{3} \sum_{j=0}^{3} a_{ij}$

Likewise, eight equations for the derivatives in the *x*-direction and the *y*-direction

1. $f_x(0,0) = p_x(0,0) = a_{10}$
2. $f_x(1,0) = p_x(1,0) = a_{10} + 2a_{20} + 3a_{30}$
3. $f_x(0,1) = p_x(0,1) = a_{10} + a_{11} + a_{12} + a_{13}$
4. $f_x(1,1) = p_x(1,1) = \sum_{i=1}^{3} \sum_{j=0}^{3} a_{ij} i$
5. $f_y(0,0) = p_y(0,0) = a_{01}$
6. $f_y(1,0) = p_y(1,0) = a_{01} + a_{11} + a_{21} + a_{31}$
7. $f_y(0,1) = p_y(0,1) = a_{01} + 2a_{02} + 3a_{03}$
8. $f_y(1,1) = p_y(1,1) = \sum_{i=0}^{3} \sum_{j=1}^{3} a_{ij} j$

And four equations for the cross derivative *xy*.

1. $f_{xy}(0,0) = p_{xy}(0,0) = a_{11}$
2. $f_{xy}(1,0) = p_{xy}(1,0) = a_{11} + 2a_{21} + 3a_{31}$
3. $f_{xy}(0,1) = p_{xy}(0,1) = a_{11} + 2a_{12} + 3a_{13}$
4. $f_{xy}(1,1) = p_{xy}(1,1) = \sum_{i=1}^{3} \sum_{j=1}^{3} a_{ij} ij$

where the expressions above have used the following identities

$$p_x(x,y) = \sum_{i=1}^{3} \sum_{j=0}^{3} a_{ij} i x^{i-1} y^j$$
$$p_y(x,y) = \sum_{i=0}^{3} \sum_{j=1}^{3} a_{ij} x^i j y^{j-1}$$
$$p_{xy}(x,y) = \sum_{i=1}^{3} \sum_{j=1}^{3} a_{ij} i x^{i-1} j y^{j-1}.$$

Derivatives are approximated from the function values at points neighbouring the corners of the unit square.

The problem can be reformulated in the form Aα = x,

Where α is defined as

$$\alpha = \begin{bmatrix} a_{00} & a_{10} & a_{20} & a_{30} & a_{01} & a_{11} & a_{21} & a_{31} & a_{02} & a_{12} & a_{22} & a_{32} & a_{03} & a_{13} & a_{23} & a_{33} \end{bmatrix}^T$$

x is defined as

$$x = \begin{bmatrix} f(0,0) & f(1,0) & f(0,1) & f(1,1) & f_x(0,0) & f_x(1,0) & f_x(0,1) & f_x(1,1) & f_y(0,0) & f_y(1,0) & f_y(0,1) & f_y(1,1) & f_{xy}(0,0) & f_{xy}(1,0) & f_{xy}(0,1) & f_{xy}(1,1) \end{bmatrix}^T.$$

And inv(A) is given by

$$A^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 & -2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & -2 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -3 & 3 & 0 & 0 & -2 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & -2 & 0 & 0 & 1 & 1 & 0 & 0 \\ -3 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -3 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & -2 & 0 & -1 & 0 \\ 9 & -9 & -9 & 9 & 6 & 3 & -6 & -3 & 6 & -6 & 3 & -3 & 4 & 2 & 2 & 1 \\ -6 & 6 & 6 & -6 & -3 & -3 & 3 & 3 & -4 & 4 & -2 & 2 & -2 & -2 & -1 & -1 \\ 2 & 0 & -2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & -2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ -6 & 6 & 6 & -6 & -4 & -2 & 4 & 2 & -3 & 3 & -3 & 3 & -2 & -1 & -2 & -1 \\ 4 & -4 & -4 & 4 & 2 & 2 & -2 & -2 & 2 & -2 & 2 & -2 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

3.4) Scaling: Lanczos Method

In Lanczos resampling, the filter applied is Lanczos filter which is a normalized sinc function windowed by the Lanczos window. The Lanczos window is the *central* lobe of a horizontally-stretched sinc, sinc(*x*/*a*) for $-a \le x \le a$. Due to its form, the Lanczos window is also called the sinc window.

The filter is formulated as :

$$L(x) = \begin{cases} \operatorname{sinc}(x)\operatorname{sinc}(x/a) & -a < x < a, x \neq 0 \\ 1 & x = 0 \\ 0 & \text{otherwise} \end{cases}$$

Where a is the size of the window, typically taken 1, 2 or 3.
Expanding the sinc terms:

$$\operatorname{sinc}(x)\operatorname{sinc}(x/a) = \frac{a\sin(\pi x)\sin(\pi x/a)}{\pi^2 x^2}.$$

Given an image, the corresponding interpolation formula is:

$$\hat{I}(x_0, y_0) = \sum_{i=\lfloor x_0 \rfloor - a + 1}^{\lfloor x_0 \rfloor + a} \sum_{j=\lfloor y_0 \rfloor - a + 1}^{\lfloor y_0 \rfloor + a} I(i,j) L(x_0 - i) L(y_0 - j).$$

3.5) Partial Differentiation based de-noising[3]

In the project, image has first been zoomed by any of the scaling methods and then denoised using the partial differentiation equation.
The partial differentiation equation used in this project is given by

$$du/dt = c * Du - P^1 u_0 + u_0$$

$$\Rightarrow u_{n+1} = u_n + \Delta t *((c^N * Du_n{}^N + c^S * Du_n{}^S + c^E * Du_n{}^E + c^W * Du_n{}^W) - P^1u_0 + u_0)$$

Where
u_{n+1} is the current iteration result,
u_n is the previous iteration result,
Du is the gradient and $Du_n{}^N$, $Du_n{}^S$, $Du_n{}^E$, $Du_n{}^W$ are its components in North, South, East and West directions
c^i is defined as

$$c^i = e^{-(x/K)\wedge 2}$$

where $x = Du_n{}^i$

k=60
u_0 is the original image obtained after zooming.
P^1u_0 is u_0 with average filter applied on it.
$\Delta t = 0.25$

3.6) Wavelet based De-noising[17]

It takes input the image scaled by one of scaling algorithms containing noise. De-noising involves following steps:

a) Apply Haar wavelet transformation to the image for L levels, each time for rows and then for columns
b) For the transformed image, calculate the threshold value for the levels 1 to L.
c) Apply either the hard or soft thresholding depending upon requirement to the image at each level.
d) Reconstruct the image by applying inverse transform to the image.

4. Critical Analysis and Results

4.1 Performance Measurement Metrices

4.1.1) Mean Square Error[3]

Mean Square Error(MSE) is given by

$$MSE = \frac{1}{m \times n} \sum_{i=1}^{m} \sum_{j=1}^{n} [I'(i,j) - I(i,j)]^2$$

Where I is the ideal result image and I', the final result image.

4.1.2) Peak Signal Noise Ratio[3]

Peak Signal Noise Ratio(PSNR) is given by

$$PSNR = 20 \log_{10} \left[\frac{255}{RMSE} \right]$$

Where RMSE(Root Mean Square Error) is given by

$$RMSE = \sqrt{MSE}$$

4.1.3) Correlation Parameter[3]

Correlation Parameter (CP) is given by

$$CP = \frac{\sum_{i=1}^{m} \sum_{j=1}^{n} (\Delta I - \Delta \bar{I}) \times \left(\Delta \hat{I} - \Delta \bar{\hat{I}}\right)}{\sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} (\Delta I - \Delta \bar{I})^2 \times \sum_{i=1}^{m} \sum_{j=1}^{n} \left(\Delta \hat{I} - \Delta \bar{\hat{I}}\right)^2}}$$

Where ΔI is high pass filtered versions of original image obtained via a 3x3 pixel standard approximation of the Laplacian operator
$\Delta \hat{I}$ is high pass filtered versions of filtered image obtained via a 3x3 pixel standard approximation of the Laplacian operator
$\Delta \bar{I}$ and $\Delta \bar{\hat{I}}$ are mean values of I and $\hat{I}$ respectively

4.1.4) Structure Similarity Index Map [3]

Structure similarity index map for a pair of images (X,Y) is given by

$$SSIM(X,Y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}$$

where μi (i=X or Y) is the mean intensity, σi(i=X or Y) is the standard deviation, σxy= σx. σy and Ci (i=1or2) is the constant to avoid instability when μx^2+ μy^2 is very close to zero and is defined as Ci=(kiL)^2 in which ki << 1and L is the dynamic range of pixel values.

4.2 Results and Discussion

4.2.1) With Static Image

Fig 4.2.1: Used Sample Image

Fig 4.2.2: Ideal Output for 8x

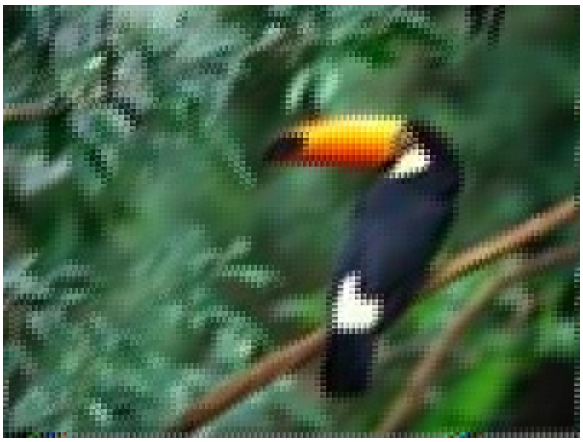


Fig: 4.2.3: Bicubic 8x

Fig: 4.2.4: Haar+Bicubic 8x

Fig: 4.2.5: PDE+Bicubic 8x

Fig 4.2.6: Lanczos 8x

Fig: 4.2.7: Haar+Lanczos 8x

Fig 4.2.8: PDE+Lanczos 8x

Table 4.2.1: Performance Metrics for various methods with static image

Denoise Method	Scaling Algorithm	Scale Factor	PSNR	MSE	CP	SSIM
None	Nearest Neighbor	2x	32.20773333	39.1562	0.1961	79.28%
		4x	31.79463333	43.05226667	0.084966667	64.14%
		8x	31.7082	43.9169	0.028433333	61.18%
		10x	31.47093333	46.3734	0.0215	62.99%
None	Bilinear	2x	32.45406667	37.00596667	0.137633333	80.92%
		4x	31.3006	48.21606667	0.0769	57.98%
		8x	31.1321	50.12486667	0.029166667	50.81%
		10x	30.90446667	52.82306667	0.020033333	52.22%
None	Bicubic	2x	31.65	44.53	0.135066667	73.22%
		4x	31.1475	49.94023333	0.071466667	57.82%
		8x	31.17776667	49.59083333	0.0284	55.47%
		10x	30.953	52.22393333	0.022166667	57.20%
None	Lanczos(sinc 3)	2x	32.20666667	39.155	0.194	79.28%
		4x	31.7237	43.76996667	0.077533333	65.08%
		8x	31.77253333	43.2751	0.028566667	62.41%
		10x	31.55286667	45.52106667	0.017533333	63.95%
PDE	Nearest Neighbor	2x	32.13363333	39.83296667	0.201733333	78.92%
		4x	31.67866667	44.2309	0.0836	68.77%
		8x	31.654	44.4832	0.028066667	68.64%

		10x	31.45523333	46.56463333	0.018766667	70.62%
PDE	Bilinear	2x	31.84516667	42.54386667	0.0856	75.55%
		4x	31.46703333	46.42593333	NaN	67.93%
		8x	31.49626667	46.12073333	Nan	70.07%
		10x	31.32753333	47.94873333	Nan	72.35%
PDE	Bicubic	2x	32.08643333	40.2651	0.1343	78.65%
		4x	31.6019	45.0167	Nan	69.51%
		8x	31.6413	44.6099	Nan	70.71%
		10x	31.4556	46.5551	Nan	72.84%
PDE	Lanczos(sinc 3)	2x	32.17563333	39.45016667	0.163866667	78.96%
		4x	31.67576667	44.2569	Nan	70.24%
		8x	31.6958	44.05656667	Nan	70.97%
		10x	31.51136667	45.9635	Nan	73.05%
Wavelet	Nearest Neighbor	2x	27.3597	120.2297333	0.1766	67.90%
		4x	28.37886667	95.11423333	0.088466667	59.61%
		8x	29.4118	74.7324	0.0304	61.27%
		10x	29.43716667	74.2464	0.036833333	63.67%
Wavelet	Bilinear	2x	27.377	119.7233333	0.113566667	65.22%
		4x	28.34353333	95.8128	0.072733333	54.58%
		8x	29.35303333	75.7613	0.031266667	55.70%
		10x	29.22343333	78.03526667	0.015833333	55.23%
Wavelet	Bicubic	2x	27.25233333	123.1372333	0.127333333	62.56%
		4x	28.16173333	99.88206667	0.071766667	56.26%
		8x	29.185	78.73416667	0.0273	59.10%
		10x	29.09976667	80.2529	0.026633333	59.07%
Wavelet	Lanczos(sinc 3)	2x	27.3597	120.2297333	0.1766	67.90%
		4x	28.437	93.80763333	0.0766	62.08%
		8x	29.63996667	70.94183333	0.0295	61.96%
		10x	29.54866667	72.39613333	0.019266667	64.41%

S. NO.	**Method**
1	None + Nearest Neighbor
2	None + Bilinear
3	None + Bicubic
4	None + Lanczos(sinc
5	PDE + Nearest Neighbor
6	PDE + Bilinear
7	PDE + Bicubic
8	PDE + Lanczos(sinc
9	Wavelet + Nearest Neighbor
10	Wavelet + Bilinear
11	Wavelet + Bicubic
12	Wavelet + Lanczos(sinc

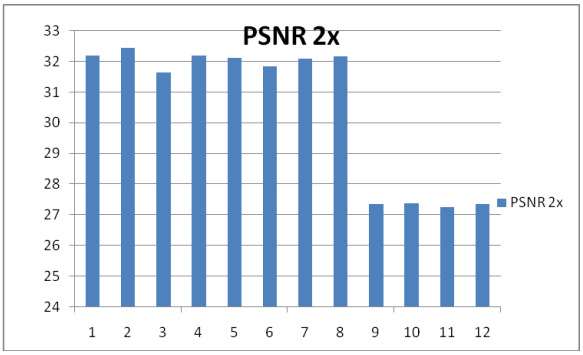


Fig: 4.2.9: PSNR vs Methods for 2x

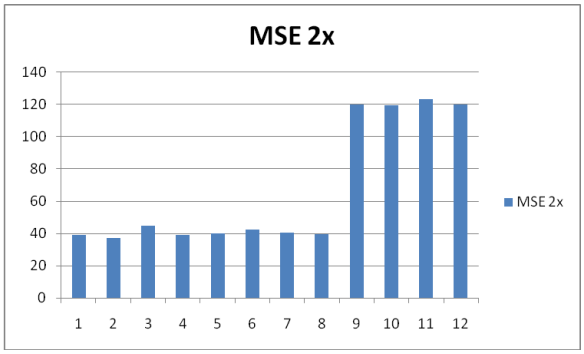


Fig: 4.2.10: MSE vs Methods for 2x

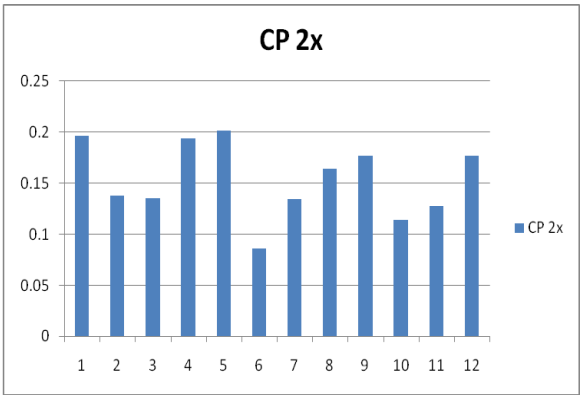


Fig: 4.2.11: CP vs Methods for 2x

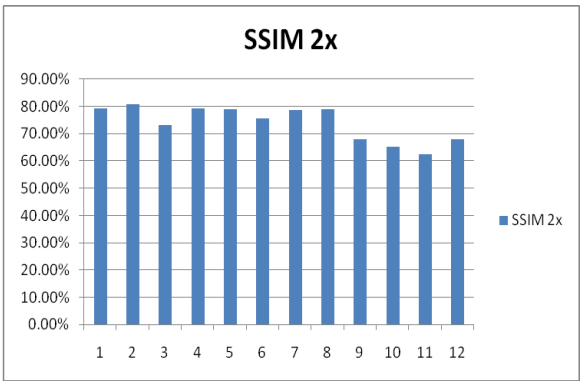


Fig: 4.2.12: SSIM vs Methods for 2x

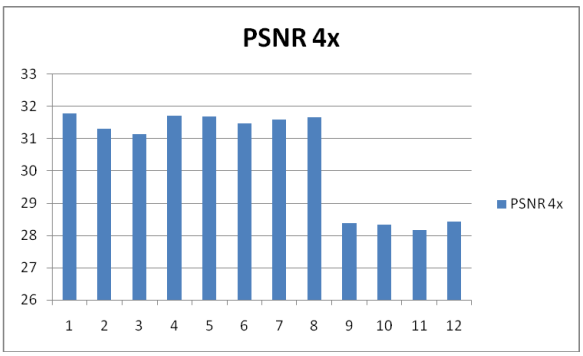


Fig: 4.2.13: PSNR vs Methods for 4x

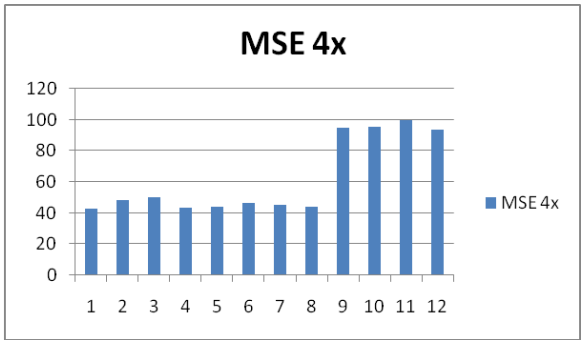


Fig: 4.2.14: MSE vs Methods for 4x.

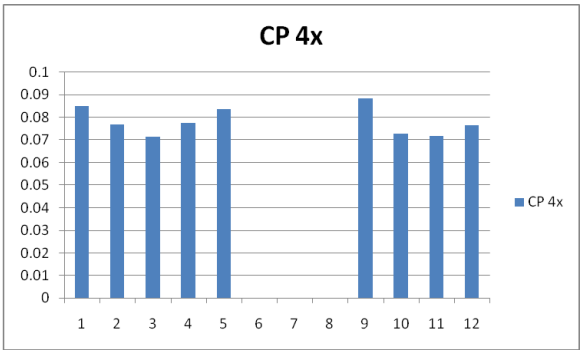


Fig: 4.2.15: CP vs Methods for 4x

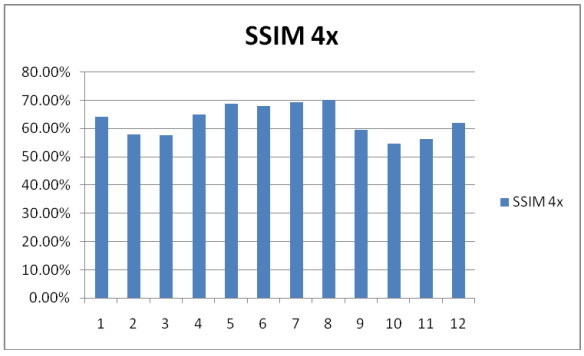


Fig: 4.2.16: SSIM vs Methods for 4x.

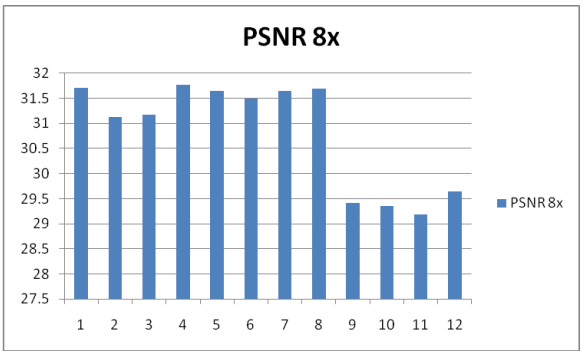


Fig: 4.2.17: PSNR vs Methods for 8x

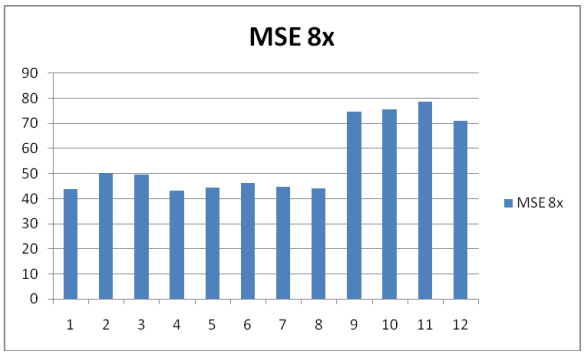


Fig: 4.2.18: MSE vs Methods for 8x.

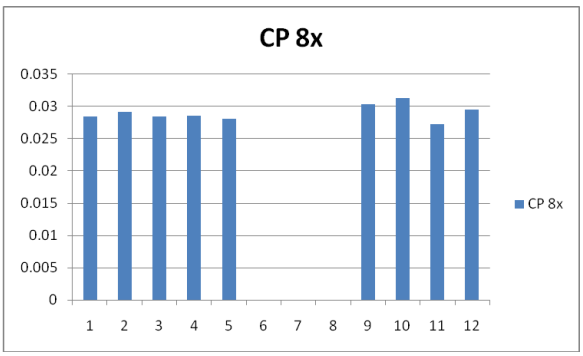


Fig: 4.2.19: CP vs Methods for 8x

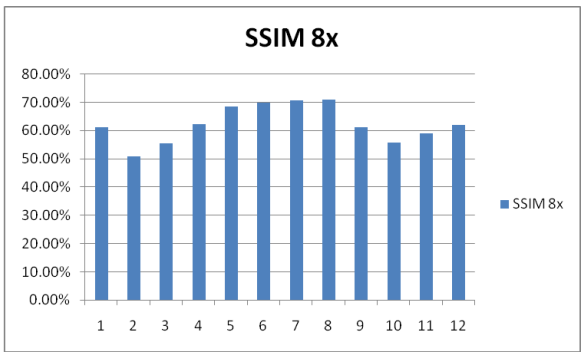


Fig: 4.2.20: SSIM vs Methods for 8x.

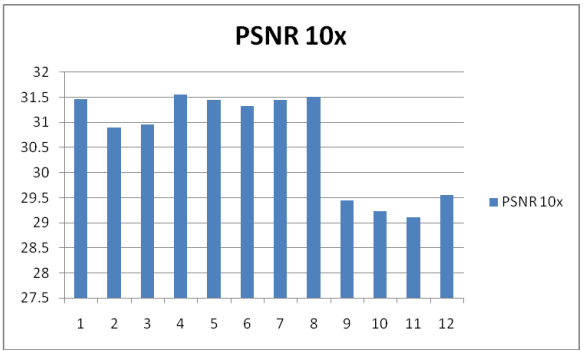


Fig: 4.2.21: PSNR vs Methods for 10x

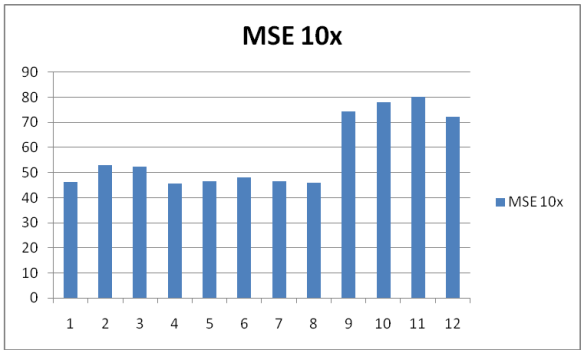


Fig: 4.2.22: MSE vs Methods for 10x.

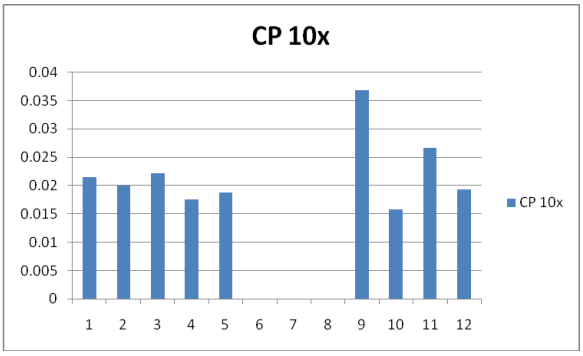


Fig: 4.2.23: CP vs Methods for 10x

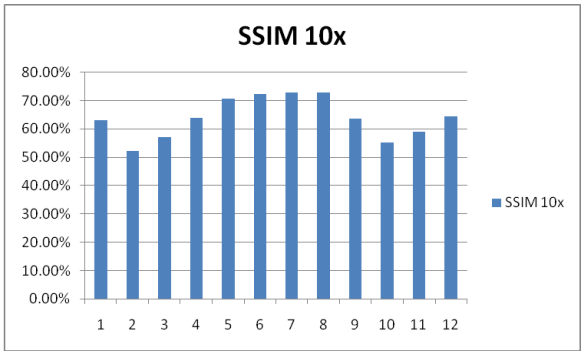
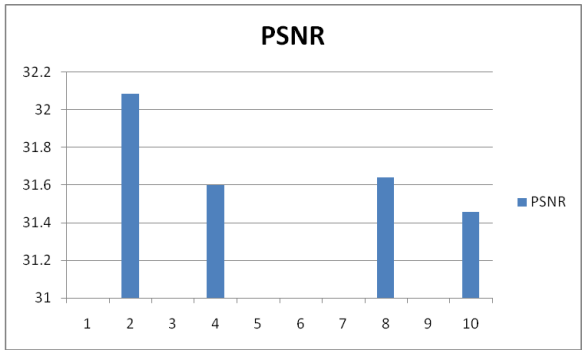


Fig: 4.2.24: SSIM vs Methods for 10x.

vs Zoom factor for PDE-Bicubic



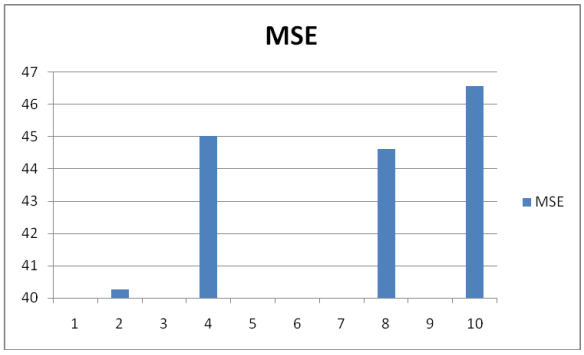


Fig: 4.2.25: PSNR vs Zoom Levels **Fig: 4.2.26: MSE vs Zoom Levels**

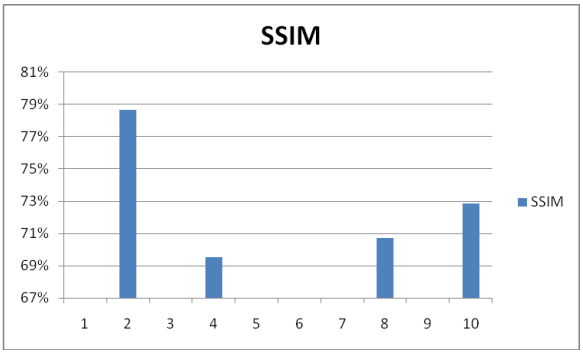
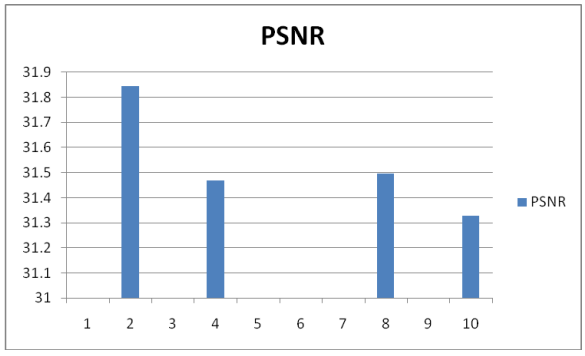


Fig: 4.2.27: SSIM vs Zoom Levels

vs Zoom factor for PDE-Bilinear



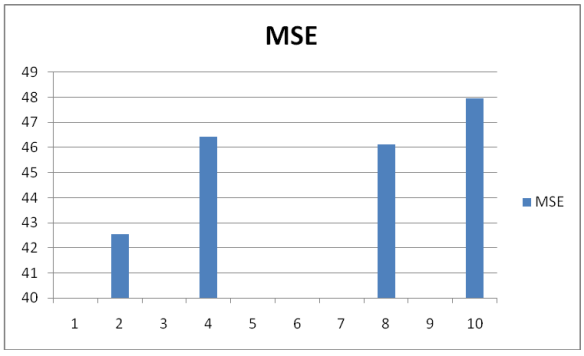


Fig: 4.2.28: PSNR vs Zoom Levels **Fig: 4.2.29: MSE vs Zoom Levels**

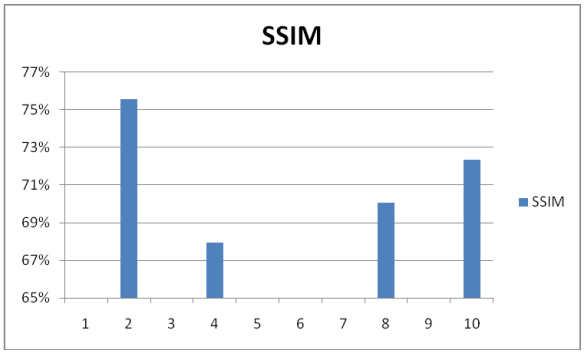
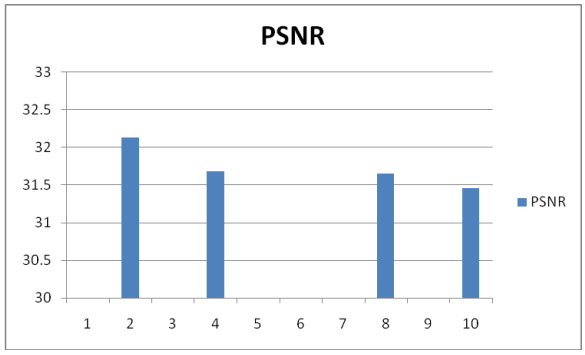


Fig: 4.2.30: SSIM vs Zoom Levels

vs Zoom factor for PDE-Nearest Neighbor



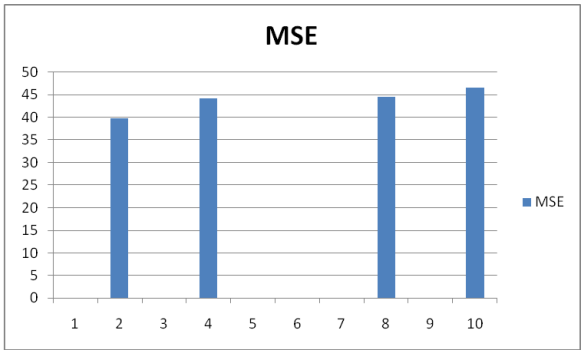
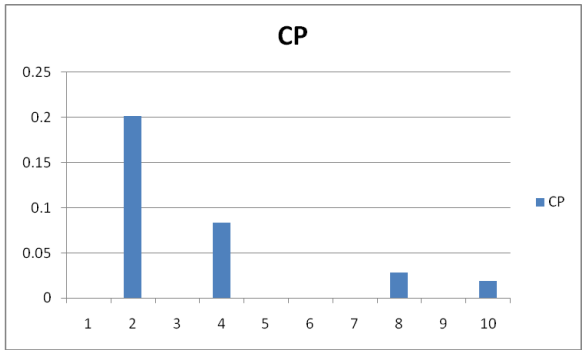


Fig: 4.2.31: PSNR vs Zoom Levels **Fig: 4.2.32: MSE vs Zoom Levels**



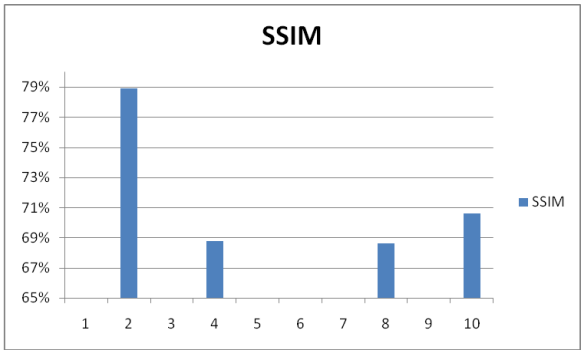
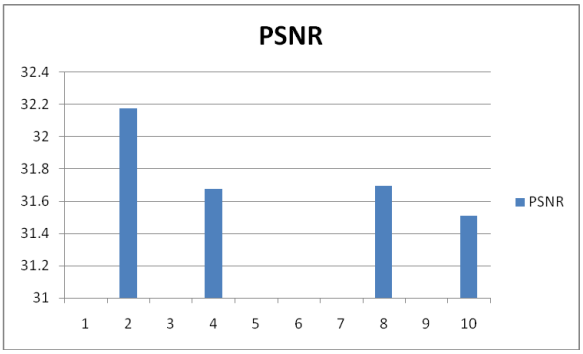


Fig: 4.2.33: CP vs Zoom Levels **Fig: 4.2.34: SSIM vs Zoom Levels**

vs Zoom factor for PDE-Lanczos



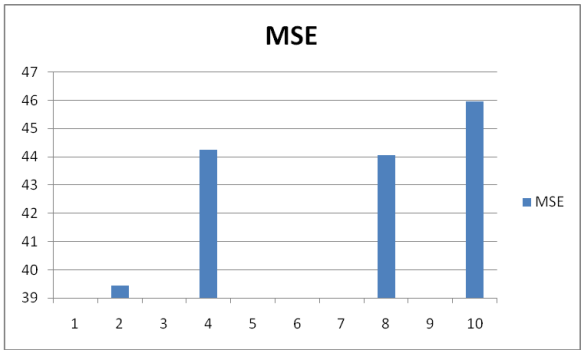


Fig: 4.2.35: PSNR vs Zoom Levels **Fig: 4.2.36: MSE vs Zoom Levels**

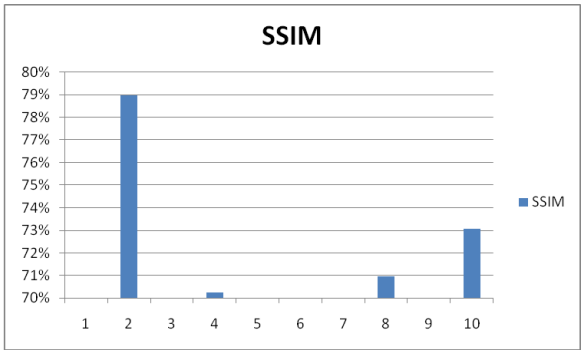


Fig: 4.2.37: SSIM vs Zoom Levels

vs Zoom factor for Haar-Bicubic

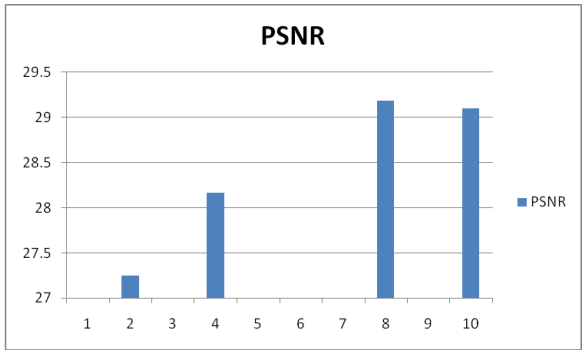


Fig: 4.2.38: PSNR vs Zoom Levels

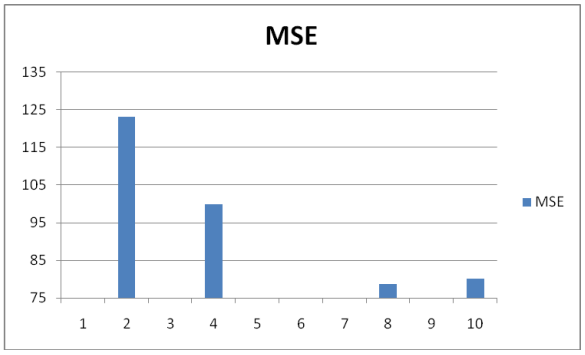


Fig: 4.2.39: MSE vs Zoom Levels

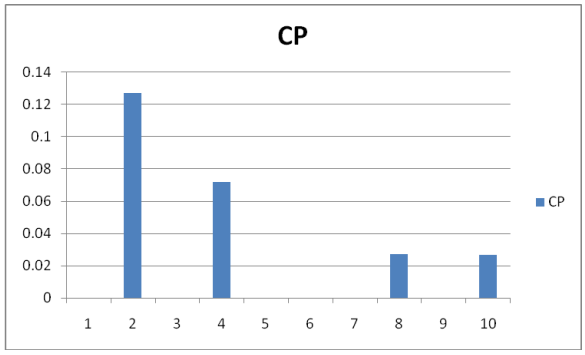


Fig: 4.2.40: CP vs Zoom Levels

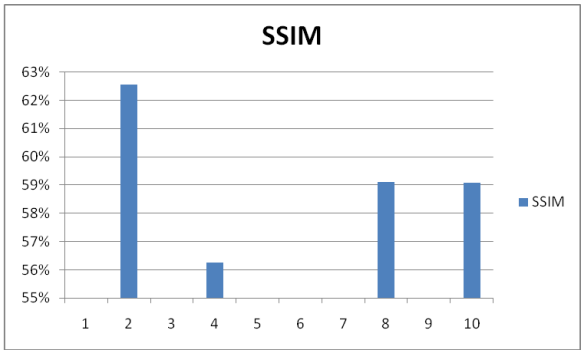


Fig: 4.2.41: SSIM vs Zoom Levels

vs Zoom factor for Haar-Bilinear

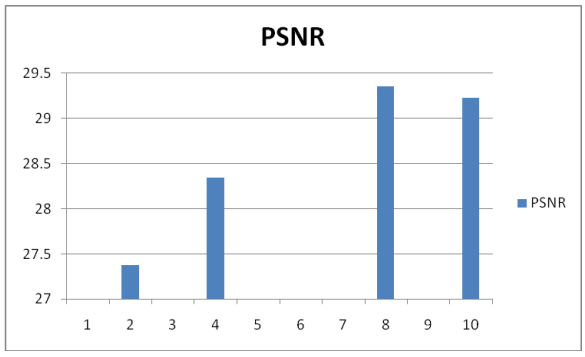


Fig: 4.2.42: PSNR vs Zoom Levels

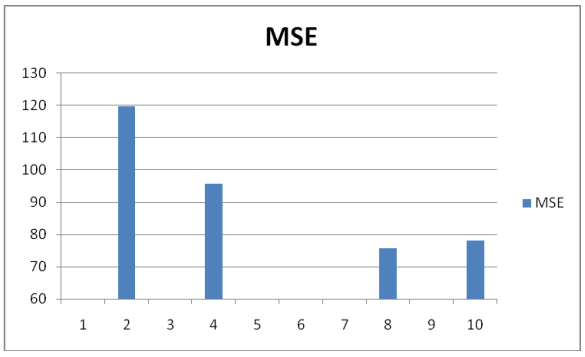


Fig: 4.2.43: MSE vs Zoom Levels

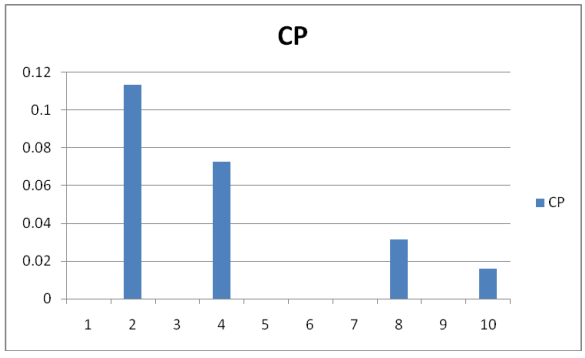


Fig: 4.2.44: CP vs Zoom Levels

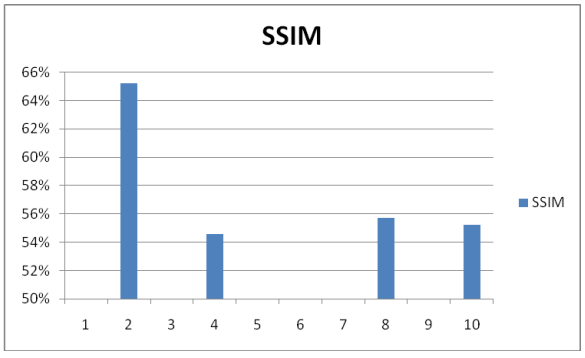


Fig: 4.2.45: SSIM vs Zoom Levels

vs Zoom factor for Haar-Nearest-Neighbor

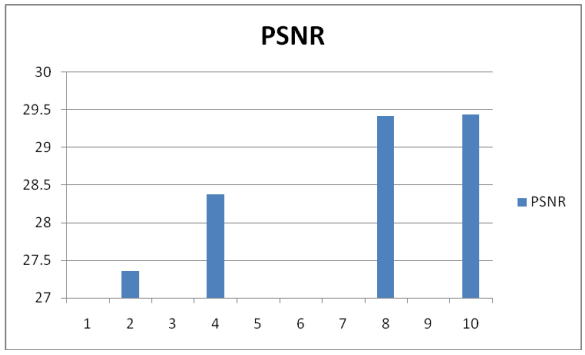


Fig: 4.2.46: PSNR vs Zoom Levels

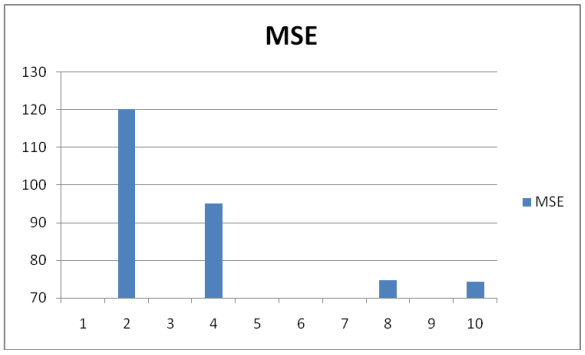


Fig: 4.2.47: MSE vs Zoom Levels

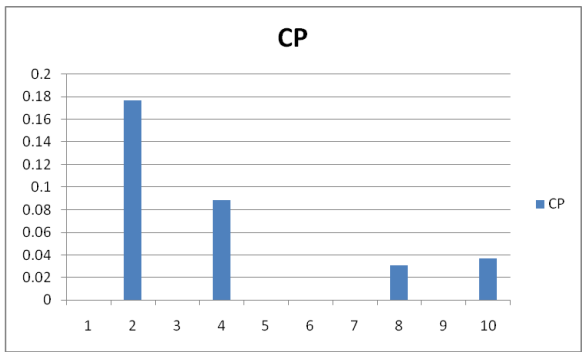


Fig: 4.2.48: CP vs Zoom Levels

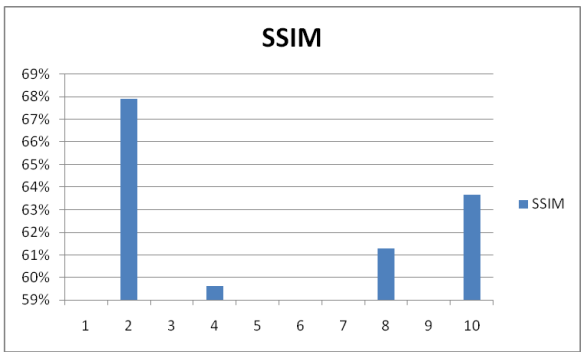


Fig: 4.2.49: SSIM vs Zoom Levels

vs Zoom factor for Haar-lanczos

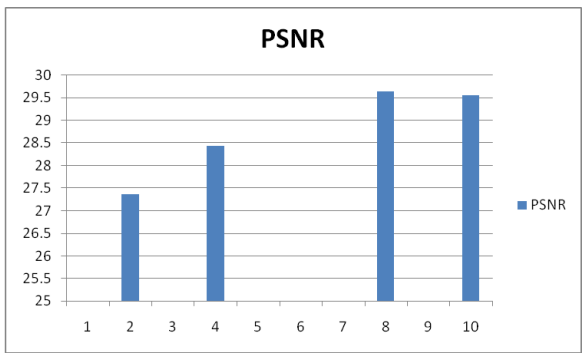


Fig: 4.2.50: PSNR vs Zoom Levels

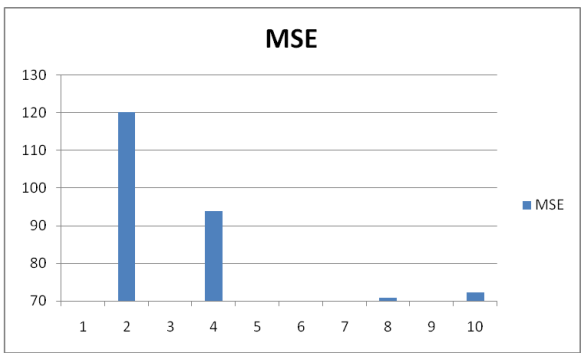


Fig: 4.2.51: MSE vs Zoom Levels

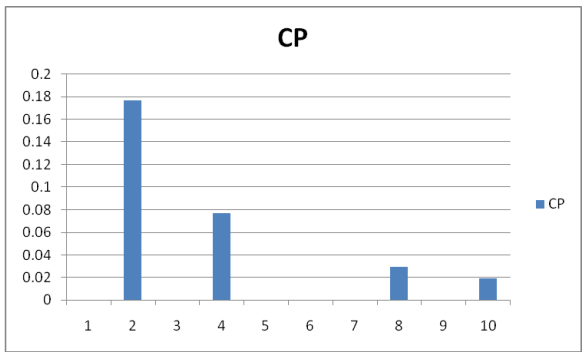


Fig: 4.2.52: CP vs Zoom Levels

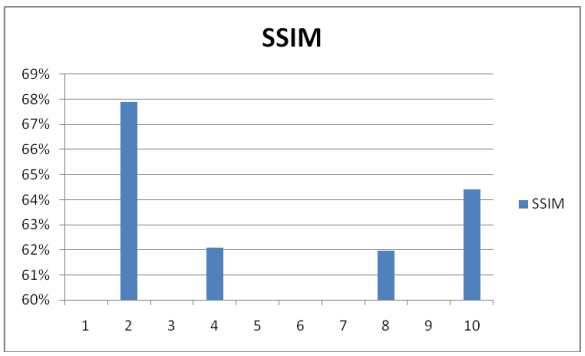


Fig: 4.2.53: SSIM vs Zoom Levels

Discussion:

For scaling factor 2, best metrics results:

- PSNR: Bilinear
- MSE: Bilinear
- CP: PDE+Nearest Neighbor
- SSIM: Bilinear

For scaling factor 4, best metrics results:

- PSNR: Nearest Neighbor
- MSE: Nearest Neighbor
- CP: Haar+Nearest Neighbor
- SSIM: PDE+ Lanczos

For scaling factor 8, best metrics results:

- PSNR: lanczos
- MSE: lanczos
- CP: Haar+Bicubic
- SSIM: PDE+ Lanczos

For scaling factor 10, best metrics results:

- PSNR: lanczos
- MSE: lanczos
- CP: Haar+Nearest Neighbor
- SSIM: PDE+ Lanczos

While varying the Zoom factors, it is observed that for all except haar performance decreses as zoom level increases while for haar, is is opposite.

4.2.2) With Two Different but continuous Video Frames

Fig: 4.2.54: Frame 1

Fig: 4.2.55: Frame 2

Fig: 4.2.56: Ideal Frame 1 for 8x

Fig: 4.2.57: Ideal Frame 2 for 8x

Fig: 4.2.58: Bicubic Frame 1 for 8x

Fig: 4.2.59: Haar+Bicubic Frame 1 for 8x

Fig 4.2.60: PDE+Bicubic Frame 1 for 8x

Fig: 4.2.61: Bicubic Frame 2 for 8x

Fig: 4.2.62: Haar+Bicubic Frame 2 for 8x

Fig: 4.2.63: PDE+Bicubic Frame 2 for 8x

Table 4.2.2: Performance Metrics for various methods with Frame 1 of Video

Denoise Method	Scaling Algorithm	Scale factor	PSNR (Frame 1)	MSE (Frame 1)	CP (Frame 1)	MSSIM (Frame 1)
None	Nearest Neighbour	2x	33.4584	29.3318	0.2062	86.28%
		4x	32.3407	37.9393	0.0619	69.38%
		8x	31.7954	43.0088	0.0068	64.58%
		10x	31.8557	42.4163	0.0052	65.59%
None	Bilinear	2x	32.7096	34.8461	NaN	82.04%
		4x	31.6343	44.6349	0.0229	62.07%
		8x	31.2771	48.4612	0	56.82%
		10x	31.348	47.677	0	57.81%

None	Bicubic	2x	32.3665	37.7103	0.0951	74.78%
		4x	31.7244	43.7171	0.0122	60.73%
		8x	31.4719	46.3342	0	58.60%
		10x	31.5543	45.4631	0.0032	59.79%
None	Lanczos(sinc 3)	2x	33.4584	29.3318	0.2062	86.28%
		4x	32.9439	33.0198	0.0382	74.45%
		8x	32.7214	34.7541	0.005	69.42%
		10x	32.8748	33.5485	0.0121	69.54%
PDE	Nearest Neighbour	2x	32.6663	35.1979	0.2323	82.27%
		4x	32.0865	40.2238	0.0441	73.13%
		8x	31.7104	43.8628	0.037	70.41%
		10x	31.7909	43.0553	0.0165	71.02%
PDE	Bilinear	2x	31.9748	41.2818	NaN	79.21%
		4x	31.5563	45.4542	NaN	71.48%
		8x	31.3225	47.9694	NaN	71.06%
		10x	31.4158	46.9496	NaN	72.21%
PDE	Bicubic	2x	32.571	35.9787	0.1438	82.25%
		4x	31.9198	41.7991	NaN	73.65%
		8x	31.584	45.1629	NaN	72.13%
		10x	31.6823	44.151	NaN	73.06%
PDE	Lanczos(sinc 3)	2x	32.7778	34.3059	0.2373	83.37%
		4x	32.0304	40.7491	NaN	74.48%
		8x	31.6739	44.2376	NaN	72.54%
		10x	31.78	43.1685	NaN	73.39%
Wavelet	Nearest Neighbour	2x	37.1975	12.3987	0.177	75.96%
		4x	35.004	20.5453	0.054	64.43%
		8x	34.0242	25.7431	0.0169	64.64%
		10x	33.8098	27.0462	0.0096	65.91%
Wavelet	Bilinear	2x	36.0712	16.068	0	70.62%
		4x	34.0289	25.7154	0.023	59.38%
		8x	33.3294	30.2095	0	60.29%
		10x	33.1479	31.4987	0	60.49%
Wavelet	Bicubic	2x	35.5396	18.1609	0.0815	65.99%
		4x	34.2577	24.3956	0.0123	59.74%
		8x	33.544	28.7528	0	61.54%
		10x	33.3607	29.9927	0.0035	61.50%
Wavelet	Lanczos(sinc 3)	2x	37.1975	12.3987	0.177	75.96%
		4x	36.2678	15.359	0.0395	70.72%
		8x	35.3861	18.8187	0.014	68.75%
		10x	35.4612	18.4949	0.0119	69.91%

Table 4.2.3: Performance Metrics for various methods with Frame 2 of Video

Denoise Method	Scaling Algorithm	Scale factor	PSNR(Frame 2)	MSE(Frame 2)	CP(Frame 2)	MSSIM(Frame 2)
None	Nearest Neighbour	2x	33.5337	28.8252	0.0996	86.36%
		4x	32.5116	36.4741	0.0134	69.83%
		8x	31.9651	41.3599	0	65.30%

		10x	32.0216	40.8254	0	66.34%
None	Bilinear	2x	32.9105	33.2689	NaN	82.05%
		4x	31.7642	43.3198	0.0048	62.53%
		8x	31.4264	46.8239	0	57.33%
		10x	31.4949	46.0908	0	58.39%
None	Bicubic	2x	32.5086	36.4956	0.0368	74.85%
		4x	31.8616	42.3567	0.017	60.97%
		8x	31.606	44.9249	0	59.12%
		10x	31.6883	44.0819	0	60.40%
None	Lanczos(sinc 3)	2x	33.5337	28.8252	0.0996	86.36%
		4x	33.0446	32.2597	0.004	74.48%
		8x	32.8455	33.7747	0.0112	69.82%
		10x	32.9895	32.6729	0	70.01%
PDE	Nearest Neighbour	2x	32.7305	34.6812	0.1371	82.64%
		4x	32.1866	39.3054	0.0298	73.56%
		8x	31.7982	42.9835	0.0215	70.98%
		10x	31.8662	42.3143	0.0035	71.64%
PDE	Bilinear	2x	32.1199	39.9264	NaN	79.96%
		4x	31.7114	43.8595	NaN	72.25%
		8x	31.4847	46.2114	NaN	71.95%
		10x	31.5703	45.3087	NaN	73.08%
PDE	Bicubic	2x	32.7181	34.7846	0.1524	82.90%
		4x	32.0586	40.4848	NaN	74.37%
		8x	31.741	43.5591	NaN	72.98%
		10x	31.8257	42.7167	NaN	73.90%
PDE	Lanczos(sinc 3)	2x	32.9199	33.2023	0.1459	83.96%
		4x	32.1673	39.4838	NaN	75.22%
		8x	31.8267	42.7078	NaN	73.44%
		10x	31.9262	41.7392	NaN	74.24%
Wavelet	Nearest Neighbour	2x	37.1999	12.3907	0.102	75.56%
		4x	35.1918	19.6756	0.015	64.85%
		8x	34.3017	24.1499	0	65.27%
		10x	34.0909	25.3506	0	67.10%
Wavelet	Bilinear	2x	36.005	16.3165	0	69.73%
		4x	34.1731	24.8758	0.0106	59.98%
		8x	33.539	28.7858	0	61.04%
		10x	33.3498	30.068	0	61.50%
Wavelet	Bicubic	2x	35.606	17.8851	0.0658	65.42%
		4x	34.5036	23.0529	0.0178	60.27%
		8x	33.7764	27.2546	0	62.32%
		10x	33.5872	28.468	0	62.62%
Wavelet	Lanczos(sinc 3)	2x	37.1999	12.3907	0.102	75.56%
		4x	36.3517	15.066	0.0072	70.97%
		8x	35.5457	18.1398	0.0187	69.17%
		10x	35.662	17.6586	0.0004	71.14%

S. NO.	Method
1	None + Nearest Neighbor
2	None + Bilinear
3	None + Bicubic
4	None + Lanczos(sinc
5	PDE + Nearest Neighbor
6	PDE + Bilinear
7	PDE + Bicubic
8	PDE + Lanczos(sinc
9	Wavelet + Nearest Neighbor
10	Wavelet + Bilinear
11	Wavelet + Bicubic
12	Wavelet + Lanczos(sinc

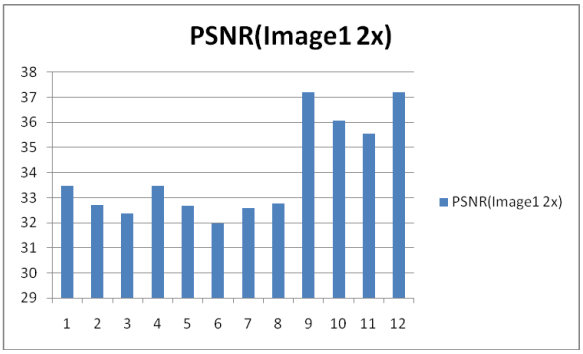


Fig: 4.2.64: PSNR vs Methods 2x Frame 1

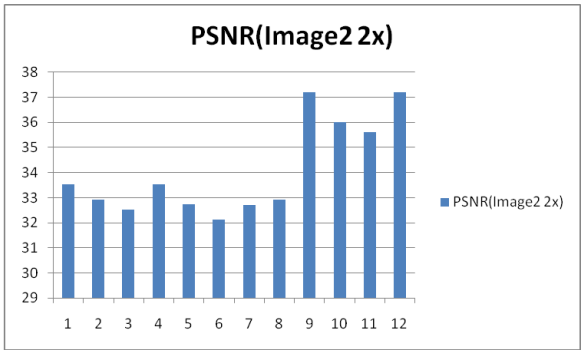


Fig: 4.2.65: PSNR vs Methods 2x Frame 2

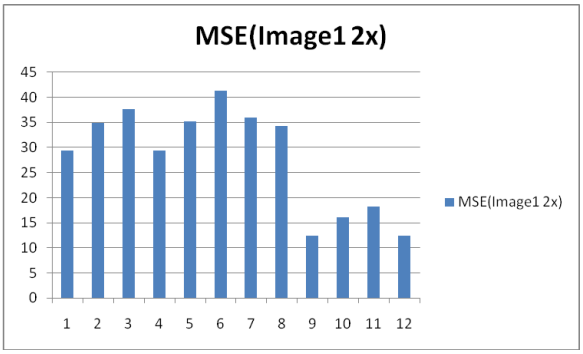


Fig: 4.2.66: MSE vs Methods 2x Frame 1

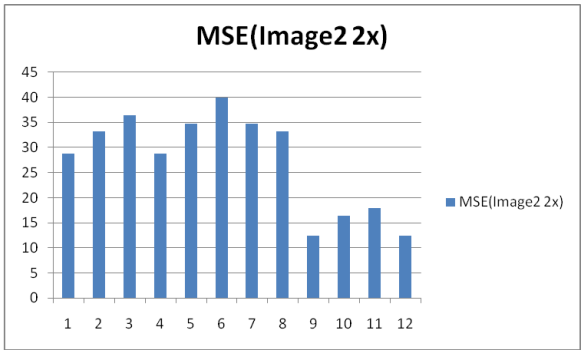


Fig: 4.2.67: MSE vs Methods 2x Frame 2

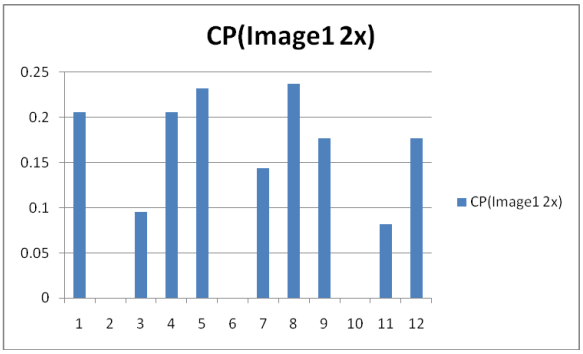


Fig: 4.2.68: CP vs Methods 2x Frame 1

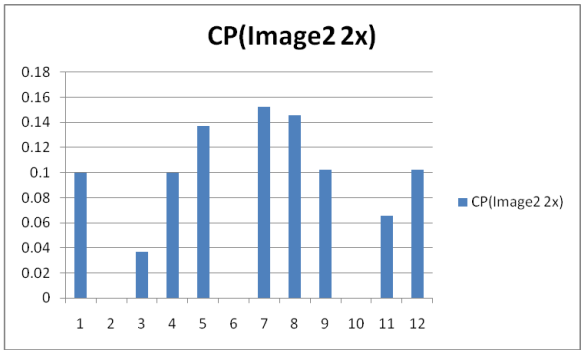


Fig: 4.2.69: CP vs Methods 2x Frame 2

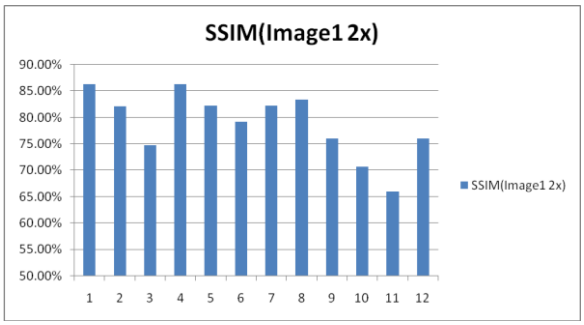


Fig: 4.2.70: SSIM vs Methods 2x Frame 1

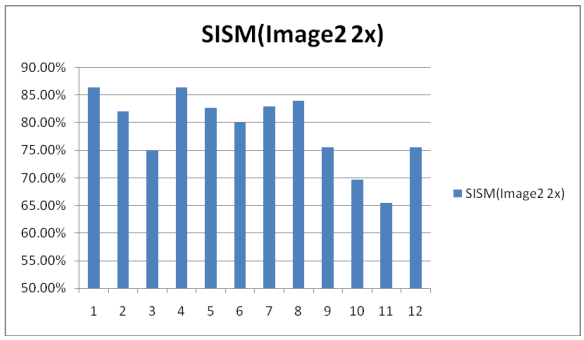


Fig: 4.2.71: SSIM vs Methods 2x Frame 2

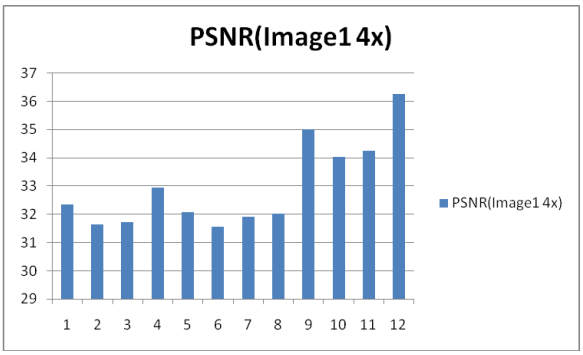


Fig: 4.2.72: PSNR vs Methods 4x Frame 1

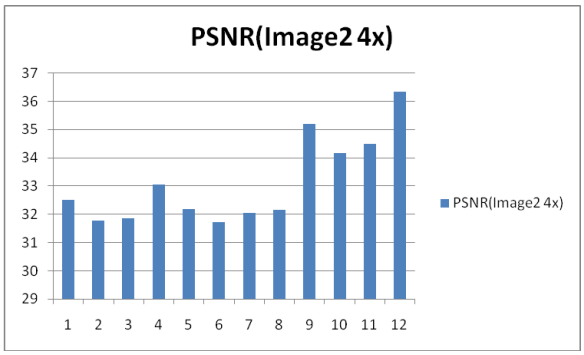


Fig: 4.2.73: PSNR vs Methods 4x Frame 2

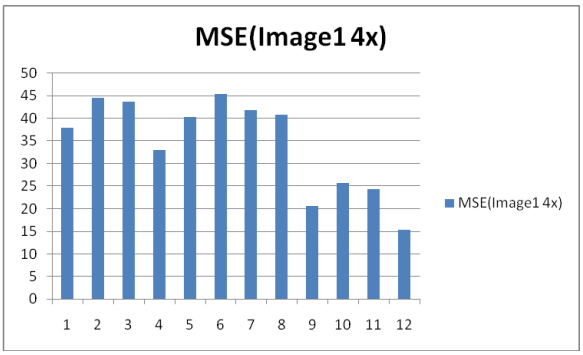


Fig: 4.2.74: MSE vs Methods 4x Frame 1

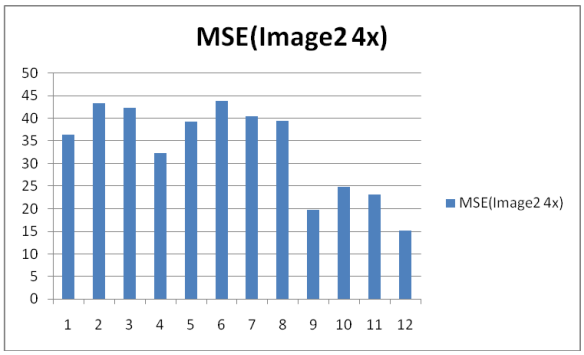


Fig: 4.2.75: MSE vs Methods 4x Frame 2

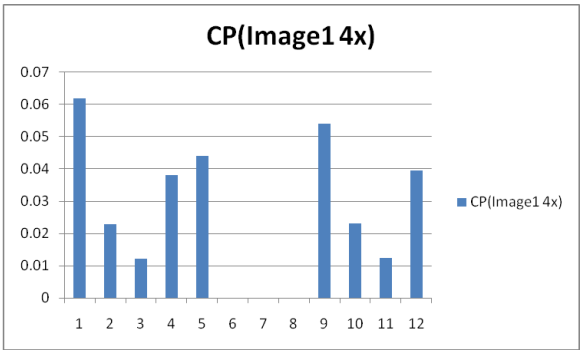


Fig: 4.2.76: CP vs Methods 4x Frame 1

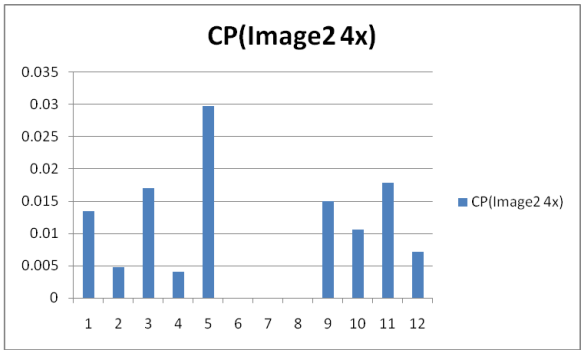


Fig: 4.2.77: PSNR vs Methods 4x Frame 2

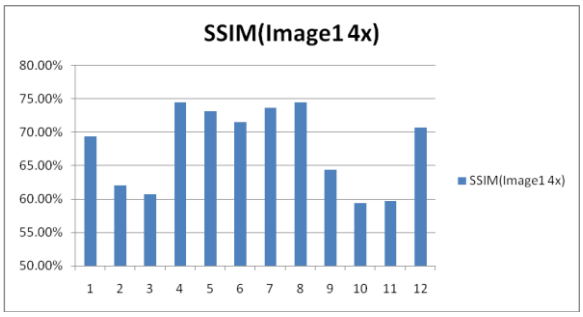


Fig: 4.2.78: SSIM vs Methods 4x Frame 1

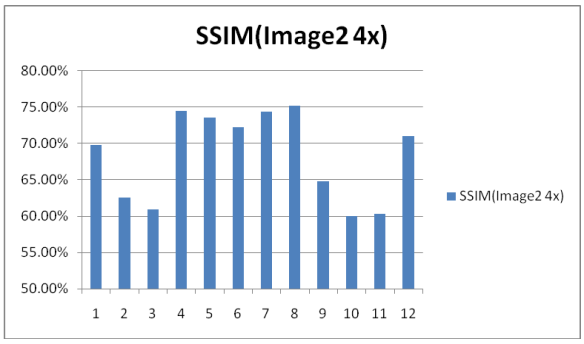


Fig: 4.2.79: SSIM vs Methods 4x Frame 2

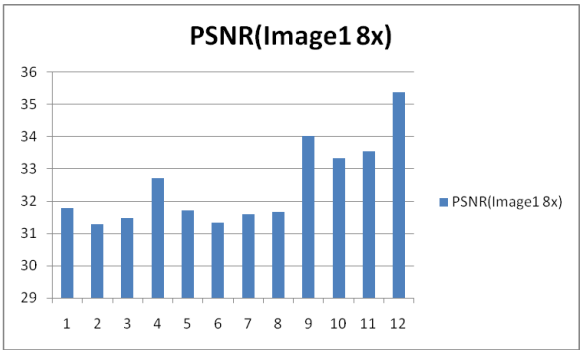


Fig: 4.2.80: PSNR vs Methods 8x Frame 1

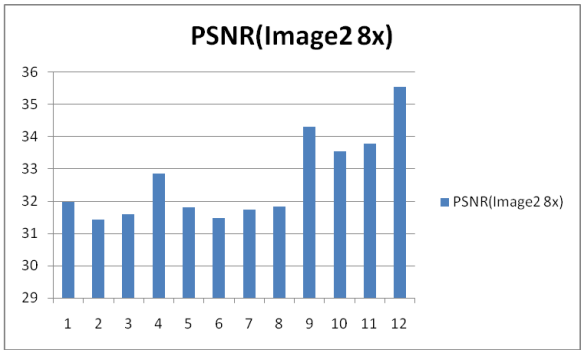


Fig: 4.2.81: PSNR vs Methods 8x Frame 2

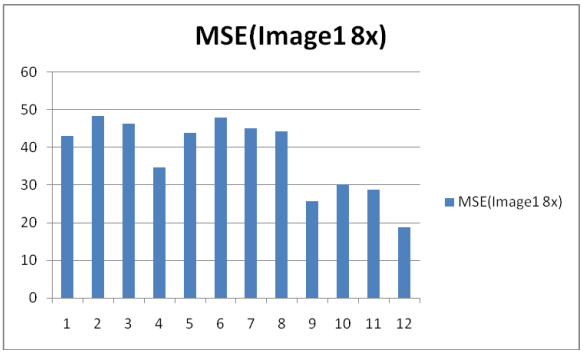


Fig: 4.2.82: MSE vs Methods 8x Frame 1

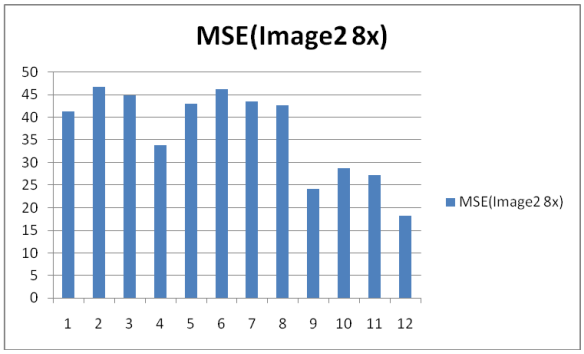


Fig: 4.2.83: MSE vs Methods 8x Frame 2

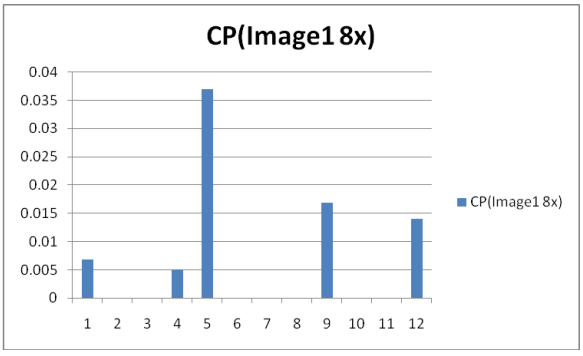


Fig: 4.2.82: CP vs Methods 8x Frame 1

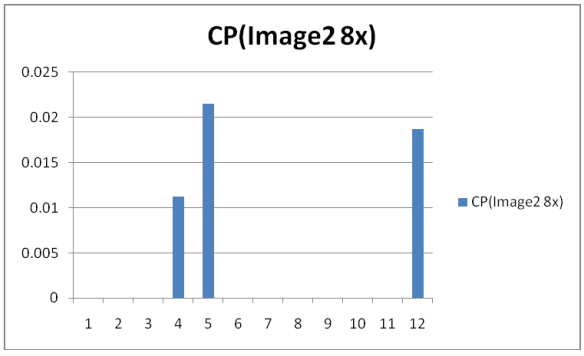
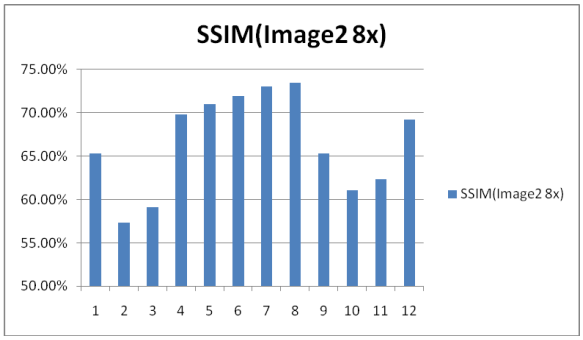


Fig: 4.2.83: CP vs Methods 8x Frame 2



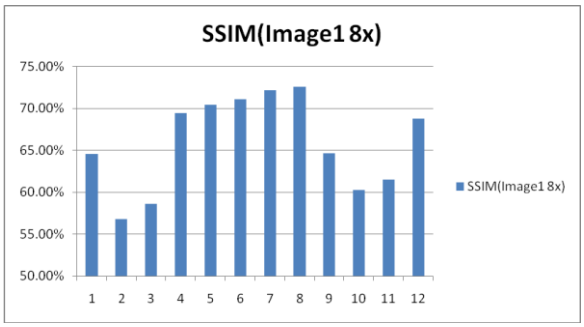
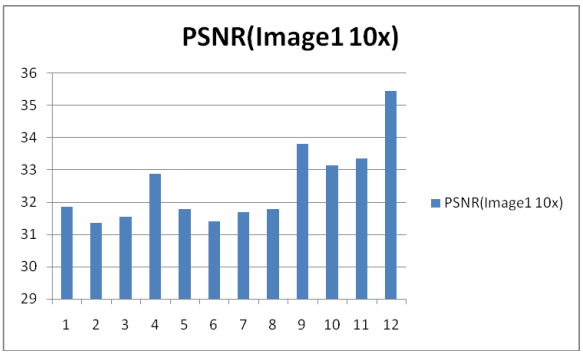


Fig: 4.2.84: SSIM vs Methods 8x Frame 1 **Fig: 4.2.85: SSIM vs Methods 8x Frame 2**



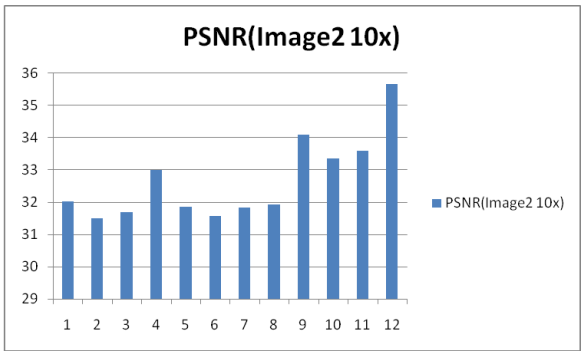
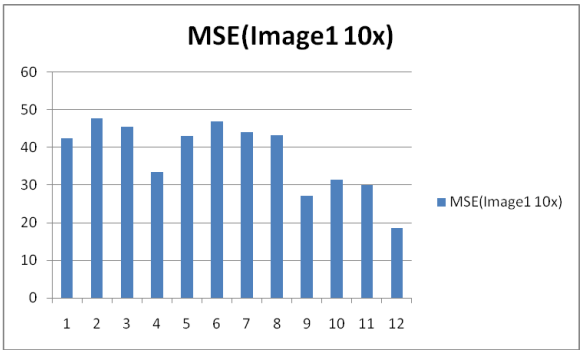


Fig: 4.2.86: PSNR vs Methods 10x Frame 1 **Fig: 4.2.87: PSNR vs Methods 8x Frame 2**



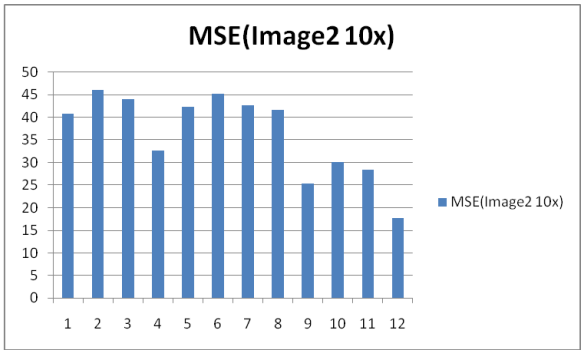
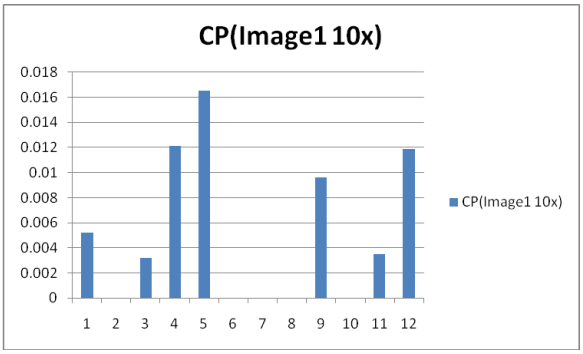


Fig: 4.2.88: MSE vs Methods 10x Frame 1 **Fig: 4.2.89: MSE vs Methods 8x Frame 2**



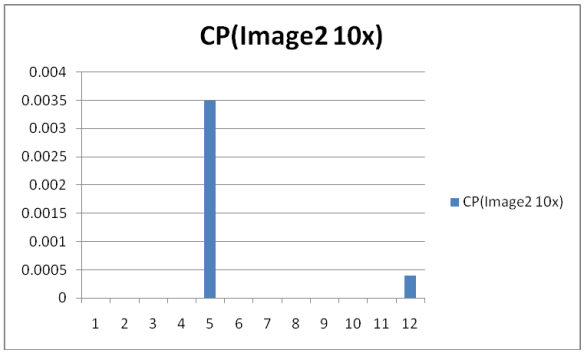
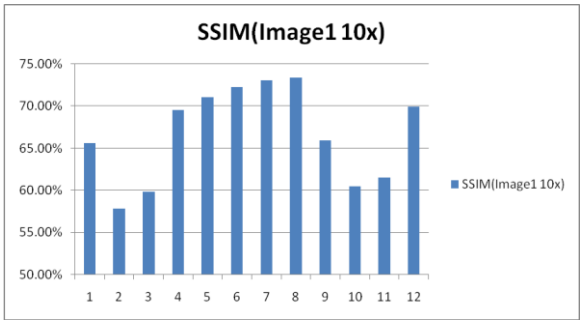


Fig: 4.2.90: CP vs Methods 10x Frame 1 **Fig: 4.2.91: CP vs Methods 8x Frame 2**



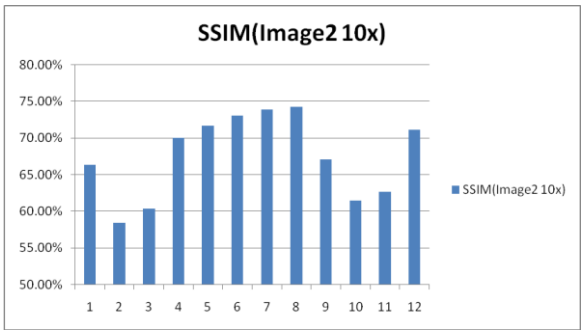


Fig: 4.2.92: SSIM vs Methods 10x Frame 1 **Fig: 4.2.93: SSIM vs Methods 8x Frame 2**

Discussion:

a) Frame 1

For scaling factor 2, best metrics results:

- PSNR: Haar+lanczos
- MSE: Haar+lanczos
- CP: PDE+lanczos
- SSIM: Lanczos

For scaling factor 4, best metrics results:

- PSNR: Haar+lanczos
- MSE: Haar+lanczos
- CP: Nearest Neighbor
- SSIM: PDE+ Lanczos

For scaling factor 8, best metrics results:

- PSNR: Haar+lanczos
- MSE: Haar+lanczos
- CP: PDE+Nearest Neighbor
- SSIM: PDE+ Lanczos

For scaling factor 10, best metrics results:

- PSNR: Haar+lanczos
- MSE: Haar+lanczos
- CP: PDE+Nearest Neighbor
- SSIM: PDE+ Lanczos

b) Frame 2

For scaling factor 2, best metrics results:

- PSNR: Haar+lanczos
- MSE: Haar+lanczos
- CP: PDE+Bicubic
- SSIM: Lanczos

For scaling factor 4, best metrics results:

- PSNR: Haar+lanczos
- MSE: Haar+lanczos
- CP: PDE+Nearest Neighbor
- SSIM: PDE+ Lanczos

For scaling factor 8, best metrics results:

- PSNR: Haar+lanczos
- MSE: Haar+lanczos
- CP: PDE+Nearest Neighbor
- SSIM: PDE+ Lanczos

For scaling factor 10, best metrics results:

- PSNR: Haar+lanczos
- MSE: Haar+lanczos
- CP: PDE+Nearest Neighbor
- SSIM: PDE+ Lanczos

4.3 Conclusion

We studied scaling of static and video images with different algorithms. Different algorithms used for scaling are Bilinear, Nearest Neighbor, Bicubic, and Lanczos (sinc 3). Images were studied with different zoom factors.
Further, the scaled images by above methods were denoised using haar wavelets and PDE based approach.
The resultant images from different methods were compared with the ideal images, and performance metrics was generated using the values like PSNR, CP, MSE and SSIM.
Based upon the performance data generated by above analysis, it is observed that de-noising improved the quality of image and video-image after scaling. Further, the extent of improvement grows as the scaling factor is increased.
It is also seen that the Partial differentiation based approach used here performs better than de-noising done by haar wavelets for static images, while for videos haar out performs PDE.
Thus, de-noising improves the quality of images and can be used along with scaling algorithms for better results.

4.4 Future Work

The Order used in the PDE based method is fixed to 2^{nd} order which can be varied further to get better results.
Wavelet used is the Haar wavelet which is the most elementary of all wavelets. There are many other wavelets that perform better than Haar like biorthogonal, symlet etc in many practical applications. Such wavelet can be explored in place of haar wavelets for better results.
Further, the threshold value in Haar wavelet, can be explored for better performance. There has been a lot of research in this regard proving for different values of this threshold.

References

[1]. A Partial Differential Equation approach to image zoom- Abdelmounim Belahmidi and Frederique Guichard.

[2]. On Exploiting Task Duplication in Parallel Program Scheduling Ishfaq Ahmad, Member, IEEE, and Yu-Kwong Kwok, Member, IEEE -- IEEE transactions on parallel and distributed systems, vol. 9, no. 9, september 1998.

[3]. Comparison of PDE based and other techniques for speckle reduction from digitally reconstructed holographic images –R. Srivastava, JRP Gupta, H. Parthasarthy, doi:10.1016/j.optlaseng.2009.09.012

[4]. Diffusion PDE for edge detection- The Essential Guide to Image Processing by Alan C. Bovik

[5]. http://en.wikipedia.org/wiki/Video

[6] http://en.wikipedia.org/wiki/Image_noise

[7] http://en.wikipedia.org/wiki/Posterization

[8] http://en.wikipedia.org/wiki/Ringing_artifact

[9] http://en.wikipedia.org/wiki/Aliasing

[10]. http://en.wikipedia.org/wiki/Bilinear_interpolation

[11] http://en.wikipedia.org/wiki/Bicubic_interpolation

[12] http://en.wikipedia.org/wiki/Lanczos_resampling

[13] Cubic convolution interpolation for digital image processing. IEEE Transactions on Signal Processing, Acoustics, Speech, and Signal Processing **29**: 1153

[14] S.Kother Mohideen† Dr. S. Arumuga Perumal††, Dr. M.Mohamed Sathik , "Image De-noising using Discrete Wavelet transform", IJCSNS International Journal of Computer Science and Network Security, VOL.8 No.1, January 2008

[15] Claude E. Duchon, Lanczos Filtering in One and Two Dimensions, Journal of Applied Metereology, 1979

[16] Susanna Minasyan1, Jaakko Astola1, Karen Egiazarian1 , David Guevorkian, PARAMETRIC HAAR-LIKE TRANSFORMS IN IMAGE DENOISING, IEEE 2006

[17] D.L.Donoho,"Denoising by soft thresholding", IEEE Trans. Inform. Theory, vol. 41, 1995, pp. 613-627.

[18] http://en.wikipedia.org/wiki/wavelet

Appendix: Source Codes

Bicubic Interpolation:

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <cv.h>
#include <highgui.h>

float a[4][4];
float calculatep(float x,float y)
{
      float p;
      p=a[0][0] + a[0][1]*y + a[0][2]*y*y + a[0][3]*y*y*y + a[1][0]*x +
a[1][1]*x*y + a[1][2]*x*y*y + a[1][3]*x*y*y*y + a[2][0]*x*x + a[2][1]*x*x*y
+ a[2][2]*x*x*y*y + a[2][3]*x*x*y*y*y + a[3][0]*x*x*x + a[3][1]*x*x*x*y
+a[3][2]*x*x*x*y*y +a[3][3]*x*x*x*y*y*y;
      return p;
}
IplImage* bicubic(IplImage* img, int m, int n)
{
    CvScalar v;

      int height,width,step,channels;
      uchar *data;
      int i,j,k,f,g,h;
      height=img->height;
      width=img->width;
      step=img->widthStep;
      channels=img->nChannels;
      data      = (uchar *)img->imageData;      //Get the image data in
'data'
      int ht=height*n;              //new image height
      int wt=width*m;                     //new image width
      IplImage *imgfinal = cvCreateImage(cvSize(wt,ht),img-
>depth,channels);//Create and initialize the new image
      uchar *datafinal =(uchar*) imgfinal->imageData;
      for(i=0;i<height*n;i++)
            for(j=0;j<width*m;j++)
                  for(k=0;k<channels;k++)
                        datafinal[(int)(i*step*m+j*channels+k)]=0;

      float w[4],x[4],y[4],z[4],i1,j1;    //w->f00,f10,f01,f11; x-
>fx00,fx10,fx01,fx11; y->fy00,fy10,fy01,fy11; z->fxy0,fxy10,fxy01,fxy11
      for(f=0;f<ht;f++)             //Traverse for all the pixels in new
image
      {
            for(g=0;g<wt;g++)
            {
                  i1=((float)f)/n;
                  j1=((float)g)/m;
                  i=floor(i1);            //Map to the original image
pixels
                  j=floor(j1);
                  i1=i1-i;
                  j1=j1-j;
```

```
                    for(k=0;k<channels;k++)
                    {

                          //Calculate f values
                          w[0]=data[i*step+j*channels+k];
                          w[1]=data[i*step+(j+1)*channels+k];
                          w[2]=data[(i+1)*step+j*channels+k];
                          w[3]=data[(i+1)*step+(j+1)*channels+k];

                          //Calulate fy values
                          if((i-1)>=0)//Check if value is going beyond the
scope of image
                          {
                                y[0] = 0.5*(data[(i+1)*step+j*channels+k] -
data[(i-1)*step+j*channels+k]);
                                y[1] = 0.5*(data[(i+1)*step+(j+1)*channels+k]
- data[(i-1)*step+(j+1)*channels+k]);
                          }
                          else// Selecting the neighbouring pixels in the
place
                          {
                                y[0]=0.5*(data[(i+1)*step+j*channels+k]-
data[(i)*step+j*channels+k]);
                                y[1]=0.5*(data[(i+1)*step+(j+1)*channels+k]-
data[(i)*step+(j+1)*channels+k]);
                          }
                          if((i+2)<=(height-1))
                          {
                                y[2] = 0.5*(data[(i+2)*step+j*channels+k] -
data[(i)*step+j*channels+k]);
                                y[3] = 0.5*(data[(i+2)*step+(j+1)*channels+k]
- data[(i)*step+(j+1)*channels+k]);
                          }
                          else
                          {
                                y[2] = 0.5*(data[(i+1)*step+j*channels+k] -
data[(i)*step+j*channels+k]);
                                y[3] = 0.5*(data[(i+1)*step+(j+1)*channels+k]
- data[(i)*step+(j+1)*channels+k]);
                          }

                          //Calculate fx values
                          if((j-1)>=0)
                          {
                                x[0]=0.5*(data[i*step+(j+1)*channels+k] -
data[i*step+(j-1)*channels+k]);
                                x[2]=0.5*(data[(i+1)*step+(j+1)*channels+k] -
data[(i+1)*step+(j-1)*channels+k]);
                          }
                          else
                          {
                                x[0]=0.5*(data[i*step+(j+1)*channels+k] -
data[i*step+(j)*channels+k]);
                                x[2]=0.5*(data[(i+1)*step+(j+1)*channels+k] -
data[(i+1)*step+(j)*channels+k]);
                          }
                          if((j+2)<=(width-1))
                          {
                                x[1]=0.5*(data[i*step+(j+2)*channels+k] -
data[(i)*step+j*channels+k]);
                                x[3]=0.5*(data[(i+1)*step+(j+2)*channels+k] -
data[(i+1)*step+(j)*channels+k]);
```

```
                        }
                        else
                        {
                                x[1]=0.5*(data[i*step+(j+1)*channels+k] -
data[(i)*step+j*channels+k]);
                                x[3]=0.5*(data[(i+1)*step+(j+1)*channels+k] -
data[(i+1)*step+(j)*channels+k]);
                        }

                        //Calculate fxy values
                        if((i-1)>=0 && (j-1)>=0)
                                z[0]=0.25*(data[(i+1)*step+(j+1)*channels+k]
- data[(i-1)*step+(j+1)*channels+k] - data[(i+1)*step+(j-1)*channels+k] +
data[(i-1)*step+(j-1)*channels+k]);
                        else if((i-1)>=0)
                        {
                                z[0]=0.25*(data[(i+1)*step+(j+1)*channels+k]
- data[(i-1)*step+(j+1)*channels+k] - data[(i+1)*step+(j)*channels+k] +
data[(i-1)*step+(j)*channels+k]);
                        }
                        else if((j-1)>=0)
                        {
                                z[0]=0.25*(data[(i+1)*step+(j+1)*channels+k]
- data[(i)*step+(j+1)*channels+k] - data[(i+1)*step+(j-1)*channels+k] +
data[(i)*step+(j-1)*channels+k]);
                        }
                        else
                        {
                                z[0]=0.25*(data[(i+1)*step+(j+1)*channels+k]
- data[(i)*step+(j+1)*channels+k] - data[(i+1)*step+(j)*channels+k] +
data[(i)*step+(j)*channels+k]);
                        }
                        if((j+2)<width && (i-1)>=0)
                        {
                                z[1]=0.25*(data[(i+1)*step+(j+2)*channels+k]-
data[(i-1)*step+(j+2)*channels+k]-data[(i+1)*step+(j)*channels+k]+data[(i-
1)*step+(j)*channels+k]);
                        }
                        else if((j+2)<width)
                        {
                                z[1]=0.25*(data[(i+1)*step+(j+2)*channels+k]-
data[(i)*step+(j+2)*channels+k]-
data[(i+1)*step+(j)*channels+k]+data[(i)*step+(j)*channels+k]);
                        }
                        else if((i-1)>=0)
                        {
                                z[1]=0.25*(data[(i+1)*step+(j+1)*channels+k]-
data[(i-1)*step+(j+1)*channels+k]-data[(i+1)*step+(j)*channels+k]+data[(i-
1)*step+(j)*channels+k]);
                        }
                        else
                        {
                                z[1]=0.25*(data[(i+1)*step+(j+1)*channels+k]-
data[(i)*step+(j+1)*channels+k]-
data[(i+1)*step+(j)*channels+k]+data[(i)*step+(j)*channels+k]);
                        }
                        if((i+2)<height && (j-1)>=0)
                        {
                                z[2]=0.25*(data[(i+2)*step+(j+1)*channels+k]
- data[(i)*step+(j+1)*channels+k] - data[(i+2)*step+(j-1)*channels+k] +
data[(i)*step+(j-1)*channels+k]);
                        }
```

```
                        else if((i+2)<height)
                        {
                                z[2]=0.25*(data[(i+2)*step+(j+1)*channels+k]
- data[(i)*step+(j+1)*channels+k] - data[(i+2)*step+(j)*channels+k] +
data[(i)*step+(j)*channels+k]);
                        }
                        else if((j-1)>=0)
                        {
                                z[2]=0.25*(data[(i+1)*step+(j+1)*channels+k]
- data[(i)*step+(j+1)*channels+k] - data[(i+1)*step+(j-1)*channels+k] +
data[(i)*step+(j-1)*channels+k]);
                        }
                        else
                        {
                                z[2]=0.25*(data[(i+1)*step+(j+1)*channels+k]
- data[(i)*step+(j+1)*channels+k] - data[(i+1)*step+(j)*channels+k] +
data[(i)*step+(j)*channels+k]);
                        }
                        if((j+2)<width && (i+2)<height)
                        {
                                z[3]=0.25*(data[(i+2)*step+(j+2)*channels+k]
- data[(i)*step+(j+2)*channels+k] - data[(i+2)*step+(j)*channels+k] +
data[i*step+j*channels+k]);
                        }
                        else if((j+2)<width)
                        {
                                z[3]=0.25*(data[(i+1)*step+(j+2)*channels+k]
- data[(i)*step+(j+2)*channels+k] - data[(i+1)*step+(j)*channels+k] +
data[i*step+j*channels+k]);
                        }
                        else if((i+2)<height)
                        {
                                z[3]=0.25*(data[(i+2)*step+(j+1)*channels+k]
- data[(i)*step+(j+1)*channels+k] - data[(i+2)*step+(j)*channels+k] +
data[i*step+j*channels+k]);
                        }
                        else
                        {
                                z[3]=0.25*(data[(i+1)*step+(j+1)*channels+k]
- data[(i)*step+(j+1)*channels+k] - data[(i+1)*step+(j)*channels+k] +
data[i*step+j*channels+k]);
                        }


                        //Find the sixteen coefficients required for the
equation to find channel values
                        a[0][0]=w[0];
                        a[1][0]=y[0];
                        a[2][0]=0-(3*w[0])+(3*w[2])-(2*y[0])-y[2];
                        a[3][0]=(2*w[0])-(2*w[2])+y[0]+y[2];
                        a[0][1]=x[0];
                        a[1][1]=z[0];
                        a[2][1]=0-(3*x[0])+(3*x[2])-(2*z[0])-z[2];
                        a[3][1]=(2*x[0])-(2*x[2])+z[0]+z[2];
                        a[0][2]=0-(3*w[0])+(3*w[1])-(2*x[0])-x[1];
                        a[1][2]=0-(3*y[0])+(3*y[1])-(2*z[0])-z[1];
                        a[2][2]=(9*(w[0]-w[1]-w[2]+w[3]))+(6*(x[0]-
x[2]+y[0]-y[1]))+(3*(x[1]-x[3]+y[2]-y[3]))+(4*z[0])+(2*z[1])+(2*z[2])+z[3];
                        a[3][2]=(6*(w[1]-w[0]+w[2]-w[3]))+(3*(y[1]-
y[0]+y[3]-y[2]))+(2*(x[3]-x[1]-z[0]-z[2]))+(4*x[2])-4*x[0]-z[1]-z[3];
                        a[0][3]=(2*w[0])-(2*w[1])+x[0]+x[1];
                        a[1][3]=(2*y[0])-(2*y[1])+z[0]+z[1];
```

```
                        a[2][3]=(6*(w[1]-w[0]+w[2]-w[3]))+(3*(x[2]+x[3]-
x[0]-x[1]))+(2*(y[3]-y[2]-z[0]-z[1])) -(4*y[0])+(4*y[1])-z[2]-z[3];
                        a[3][3]=(4*(w[0]-w[1]-w[2]+w[3]))+(2*(x[0]+x[1]-
x[2]-x[3]+y[0]-y[1]+y[2]-y[3]))+z[0]+z[1]+z[2]+z[3];

                        //Calculate the values for the k channels in the
image
                        v.val[k]=(int)(calculatep(j1,i1));

                        //datafinal[(int)(f*step*m +
g*channels+k)]=(int)(calculatep(j1,i1));
                  }
                  //Assign the values to the image
                  cvSet2D(imgfinal,f,g,v);
            }
      }
      return(imgfinal);
}
int main(int argc,char *argv[])
{
      IplImage* img=0;
      float m,n;                    // m is horizontal scaling factor and n
is vertical scaling factor
      if(argc<2)                    //Check for the format of the command
line input
      {
            printf("Usage : ./a.out <image-file-name>\n");
            exit(0);
      }
      img=cvLoadImage(argv[1]);           //Load the given image
      if(!img){
      printf("Could not load image file: %s\n",argv[1]);
      exit(0);
      }
      printf("Enter the scaling factors(m n) :");
      scanf("%f%f",&m,&n);

      IplImage * imgfinal=bicubic(img,m,n);
        // show the image
      cvSaveImage("result.jpg",imgfinal);             //Save the result
image
      cvNamedWindow("Result", CV_WINDOW_AUTOSIZE);
      cvMoveWindow("Result", 100, 100);
      cvShowImage("Result", imgfinal);                //Show the result
image
      cvNamedWindow("Original", CV_WINDOW_AUTOSIZE);
      cvMoveWindow("Original", 100, 100);
      cvShowImage("Original", img);                   //Show the original
image
      cvSaveImage("original.jpg",img);                //Save the original
image
      // wait for a key
      cvWaitKey(0);

      // release the image
      cvReleaseImage(&imgfinal);                      //Free the memory
      cvReleaseImage(&img);
}
```

Bilinear Interpolation:

```
#include <iostream>
#include <stdio.h>
#include <math.h>
#include <cv.h>
#include <highgui.h>
using namespace std;

double valueat(double intensity1,double intensity2,double v1,double
v2,double v)
{
    if(v1==v2)
        return intensity1;
    else
        return((((v2-v)/(v2-v1))*intensity2) +(((v-v1)/(v2-
v1))*intensity1));
}

IplImage* bilinear(IplImage* img, double horizontal_scale, double
vertical_scale)
{
    int
orig_height,orig_width,orig_step,orig_channels,new_step,new_channels,new_he
ight,new_width,i,j,k;
    CvScalar v;
    orig_height=img->height;
    orig_width=img->width;
    orig_step=img->widthStep;
    new_step=orig_step*horizontal_scale;
    new_channels=orig_channels=img->nChannels;
    uchar *orig_data = (uchar *)img->imageData; //Get the image data in
'data'
    new_height=orig_height*vertical_scale;                  //new image
height
      new_width=orig_width*horizontal_scale;                       //new
image width
      IplImage *imgfinal = cvCreateImage(cvSize(new_width,new_height),img-
>depth,new_channels);//Create and initialize the new image
      uchar *new_data =(uchar*) imgfinal->imageData;
      for(i=0;i<new_height;i++)
            for(j=0;j<new_width;j++)
                  for(k=0;k<new_channels;k++)
                        new_data[(int)(i*new_step+j*new_channels+k)]=0;
    for(i=0;i<new_height;i++)
    {
        for(j=0;j<new_width;j++)
        {
            int x1=floor((double)j/(double)horizontal_scale);
            int y1=floor((double)i/(double)vertical_scale);
            int x2=ceil((double)j/(double)horizontal_scale);
            int y2=ceil((double)i/(double)vertical_scale);
            double x = (double)j/(double)horizontal_scale;
            double y = (double)i/(double)vertical_scale;
            for(k=0;k<new_channels;k++)
            {

                double r1=valueat((double)orig_data[(int)(y1*orig_step
+x1*orig_channels+k)],(double)orig_data[(int)(y1*orig_step
+x2*orig_channels+k)],(double)x1,(double)x2,x);
```

```
                double r2=valueat((double)orig_data[(int)(y2*orig_step
+x1*orig_channels+k)],(double)orig_data[(int)(y2*orig_step
+x2*orig_channels+k)],(double)x1,(double)x2,x);
                v.val[k]=valueat(r1,r2,(double)y1,(double)y2,y);

//new_data[(int)(i*new_step+j*new_channels+k)]=orig_data[(int)(temp_height*
orig_step + temp_width*orig_channels+k)];
            }
            cvSet2D(imgfinal,i,j,v);
        }
    }
    return(imgfinal);
}
int main(int argc,char*argv[])
{
    IplImage* img=0;
      double horizontal_scale,vertical_scale;
      if(argc<2)                    //Check for the format of the command
line input
      {
            printf("Usage : ./a.out <image-file-name>\n");
            exit(0);
      }
      img=cvLoadImage(argv[1]);           //Load the given image
      if(!img){
      printf("Could not load image file: %s\n",argv[1]);
      exit(0);
      }
      printf("Enter the scaling factors(m n) :");
      scanf("%lf%lf",&horizontal_scale,&vertical_scale);

      IplImage * imgfinal= bilinear(img,horizontal_scale,vertical_scale);
        // show the image
      cvSaveImage("result.jpg",imgfinal);             //Save the result
image
      cvNamedWindow("Result", CV_WINDOW_AUTOSIZE);
      cvMoveWindow("Result", 100, 100);
      cvShowImage("Result", imgfinal);                //Show the result
image
      cvNamedWindow("Original", CV_WINDOW_AUTOSIZE);
      cvMoveWindow("Original", 100, 100);
      cvShowImage("Original", img);                   //Show the original
image
      cvSaveImage("original.jpg",img);                //Save the original
image
      // wait for a key
      cvWaitKey(0);

      // release the image
      cvReleaseImage(&imgfinal);                      //Free the memory
      cvReleaseImage(&img);
}
```

Nearest Neighbor Interpolation:

```
#include<iostream>
#include <math.h>
#include <stdio.h>
#include <cv.h>
#include <highgui.h>
```

```
using namespace std;

void nearest_neighbour(IplImage * img, IplImage* imgfinal, double
horizontal_scale, double vertical_scale)
{
    int
orig_height,orig_width,orig_step,orig_channels,new_step,new_channels,new_he
ight,new_width,i,j,k;
    CvScalar v;
    orig_height=img->height;
    orig_width=img->width;
    orig_step=img->widthStep;
    new_step=orig_step*horizontal_scale;
    new_channels=orig_channels=img->nChannels;
    uchar *orig_data = (uchar *)img->imageData; //Get the image data in
'data'
    new_height=orig_height*vertical_scale;                    //new image
height
      new_width=orig_width*horizontal_scale;                          //new
image width
      uchar *new_data =(uchar*) imgfinal->imageData;
      for(i=0;i<new_height;i++)
            for(j=0;j<new_width;j++)
                  for(k=0;k<new_channels;k++)
                        new_data[(int)(i*new_step+j*new_channels+k)]=0;
    for(i=0;i<new_height;i++)
    {
        for(j=0;j<new_width;j++)
        {
            double x = (double)j/(double)horizontal_scale;
            double y = (double)i/(double)vertical_scale;
            int temp_width,temp_height;
            if((x - (double)floor(x)) > 0.5)
                temp_width=ceil(x);
            else
                temp_width=floor(x);
            if((y - (double)floor(y)) > 0.5)
                temp_height=ceil(y);
            else
                temp_height=floor(y);
            for(k=0;k<new_channels;k++)
            {
                v.val[k]=orig_data[(int)(temp_height*orig_step +
temp_width*orig_channels+k)];

//new_data[(int)(i*new_step+j*new_channels+k)]=orig_data[(int)(temp_height*
orig_step + temp_width*orig_channels+k)];
            }
            cvSet2D(imgfinal,i,j,v);
        }
    }
}

int main(int argc,char *argv[])
{
    IplImage* img=0;
      double horizontal_scale,vertical_scale;
      if(argc<2)                    //Check for the format of the command
line input
      {
            printf("Usage : ./a.out <image-file-name>\n");
            exit(0);
```

```
        }
        img=cvLoadImage(argv[1]);           //Load the given image
        if(!img){
        printf("Could not load image file: %s\n",argv[1]);
        exit(0);
        }
        printf("Enter the scaling factors(m n) :");
        scanf("%lf%lf",&horizontal_scale,&vertical_scale);

        IplImage * imgfinal;
        imgfinal = cvCreateImage(cvSize(horizontal_scale*img->width,
vertical_scale*img->height),img->depth,img->nChannels);//Create and
initialize the new image
        nearest_neighbour(img, imgfinal, horizontal_scale,vertical_scale);
          // show the image
        cvSaveImage("result.jpg",imgfinal);             //Save the result
image
        cvNamedWindow("Result", CV_WINDOW_AUTOSIZE);
        cvMoveWindow("Result", 100, 100);
        cvShowImage("Result", imgfinal);                //Show the result
image
        cvNamedWindow("Original", CV_WINDOW_AUTOSIZE);
        cvMoveWindow("Original", 100, 100);
        cvShowImage("Original", img);                   //Show the original
image
        cvSaveImage("original.jpg",img);                //Save the original
image
        // wait for a key
        cvWaitKey(0);

        // release the image
        cvReleaseImage(&imgfinal);                      //Free the memory
        cvReleaseImage(&img);
}
```

Lanczos (sinc 3) Interpolation:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <cv.h>
#include <highgui.h>

#define pi 3.14159265

int a=3;

double Lanczos(double x)
{
    if(x<0.000001 && x>-0.000001)
        return 1;
    else if(-a < x && a > x)
        return (a*sin(pi*x)*sin(pi*x/a))/(pi*pi*x*x);
    else
        return 0;
}

void Lanczose_Scale(IplImage *in, IplImage *out, int owidth, int oheight)
{
    int channels = in->nChannels;
```

```
    int iwidth = in->width;
    int iheight = in->height;

    int i,j,k,x,y;
    for(x=0;x<owidth;x++)
        for(y=0;y<oheight;y++)
        {
            double x0 = floor(x*1.0*iwidth/owidth);
            double y0 = floor(y*1.0*iheight/oheight);
            CvScalar temp, temp2;
            for(k=0;k<channels;k++)
                temp2.val[k] = 0;
            for(i=x0-a+1;i<=x0+a;i++)
                for(j=y0-a+1;j<=y0+a;j++)
                    if(i>=0 && i<iwidth && j>=0 && j<iheight)
                    {
                        temp = cvGet2D(in,j,i);
                        for(k=0;k<channels;k++)
                        {
                            temp2.val[k] += temp.val[k]*Lanczos(x0-
i)*Lanczos(y0-j);
                        }
                    }
            cvSet2D(out,y,x,temp2);
        }
}

int main()
{
    char input[20], output[20];
    int owidth, oheight;
    scanf("%s",input);
    scanf("%s%d%d",output,&owidth,&oheight);

    IplImage *in_image, *out_image;
    in_image = cvLoadImage(input,-1);

    if(!in_image)
    {
        printf("Error!\n");
        return 0;
    }

    out_image = cvCreateImage(cvSize(owidth,oheight),IPL_DEPTH_8U,in_image-
>nChannels);

    Lanczose_Scale(in_image, out_image, owidth, oheight);

    cvSaveImage(output,out_image);
}
```

ReaderWriter: For reading/ writing video files using OpenCV

```
#include "cv.h"
#include "highgui.h"
#include <stdio.h>

int main()
{
    CvCapture *capture = cvCreateFileCapture("test.avi");
```

```
    //CvCapture *capture = cvCreateCameraCapture(0);
    CvVideoWriter *writer = 0;
    double fps  = cvGetCaptureProperty(capture, CV_CAP_PROP_FPS);//30
    int frameW  = cvGetCaptureProperty(capture, CV_CAP_PROP_FRAME_WIDTH);
// 800
    int frameH  = cvGetCaptureProperty(capture, CV_CAP_PROP_FRAME_HEIGHT);
// 600


writer=cvCreateVideoWriter("/home/saket/out.avi",cvGetCaptureProperty(captu
re, CV_CAP_PROP_FOURCC),//CV_FOURCC('P','I','M','1'),
                            fps,cvSize(frameW,frameH));

    IplImage* img = 0;
    int i=0;
    printf("%lf %d %d\n",fps,frameH, frameW);
    while(cvGrabFrame(capture))
    {
        img=cvRetrieveFrame(capture);
        cvWriteFrame(writer,img);
        //cvSaveImage("output.jpg",img);
    }

    cvReleaseVideoWriter(&writer);
    cvReleaseCapture(&capture);
    cvReleaseImage(&img);

}
```

PDE based Scaling:

```
%%%%%%%%%%Anisotropic diffusion based interpolation: Guichard et al
%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%Author: Rajeev Srivastava, ITBHU%%%%%%%%%%%%%%%%%%

I=imread('input.jpg');
p=4;%%%Zoom out factor
%%%%%%%%%%Upsample the image to its original size by applysing Bilinear or Cubic
interepolation
 J = imresize(I,p,'bicubic');% Magnification factor 2x2, 0.5x0.5
h = fspecial('average',[3 3]);
Iproj= imfilter(J,h);
%%%%%%%%%%%%Step 3: Using nonlinear complex diffusion process for magnification, noise
smoothing and edge forming %%%%%%%%%%%
Iproj=double(Iproj);
J=double(J);
[x y z]=size(J);
dt=0.25; %dt<0.25 for stability of numerical scheme
im=double(J);
kappa=60;
option=2;% option 1 is associated with less PSNR
t=1;% niter initialized
% T=4; %Stopping Criteria T=pxp (Magnification size) e.g. T=4 for 2x2
% magnification, T=niter
T=p*p;
im = double(im);
[rows,cols,channels] = size(im);
```

```
diff = im;
for t = 1:T
  diffl = zeros(rows+2, cols+2,channels+2);
  diffl(2:rows+1, 2:cols+1,1:channels) = diff;

  % North, South, East and West differences
  deltaN = diffl(1:rows,2:cols+1,1:channels)   - diff;
  deltaS = diffl(3:rows+2,2:cols+1,1:channels) - diff;
  deltaE = diffl(2:rows+1,3:cols+2,1:channels) - diff;
  deltaW = diffl(2:rows+1,1:cols,1:channels)   - diff;

  % Conduction

  if option == 1
    cN = exp(-(deltaN/kappa).^2);
    cS = exp(-(deltaS/kappa).^2);
    cE = exp(-(deltaE/kappa).^2);
    cW = exp(-(deltaW/kappa).^2);
  elseif option == 2
    cN = 1./(1 + (deltaN/kappa).^2);
    cS = 1./(1 + (deltaS/kappa).^2);
    cE = 1./(1 + (deltaE/kappa).^2);
    cW = 1./(1 + (deltaW/kappa).^2);
  end
 diff = diff + dt*((cN.*deltaN + cS.*deltaS + cE.*deltaE + cW.*deltaW)-Iproj+J);
% diff = diff + dt*(cN.*deltaN + cS.*deltaS + cE.*deltaE + cW.*deltaW);
Ianiso=diff;
end
figure (5), imshow(uint8(diff),[]),title ('Anisotropic diff Interpolation:Zoom out 4x4');
imwrite(uint8(diff),'output.jpg','jpg');
```

Performance Analysis: PSNR, CP and MSE

```
%%%%%%%%%%%%%%%%%Performance analysis of zooming
algorithms%%%%%%%%%%%
function [MSE,rmse,SNR,PSNR,CP]= PerformanceAnalysisZoomMethods(I1,I2)
%%%%%%%%%%%%%%%%%Performance Analysis of Speckle Reduction
Methods%%%%%%%%%%%
%----------   PSNR analysis   ----------------
[x y z]=size(I1);
Q = 255;
MSE= sum(sum((I2-I1) .^ 2)) / (x * y);
rmse=sqrt(MSE);

% SNR ( sum of energies of original image/sum of energies of
% original-reconstructed image)
SNR=10*log10(sum(sum((I1).^2))./(sum(sum((I2-I1) .^ 2)));
PSNR = 10*log10(Q*Q./MSE);

 %%%%%%%%%%%%%%%%%%%%Coorelation Parameter   -CP : metric for edge
preservation%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%% I1 is the original image
%%%%%% I2 is the reconstructed/denoised image
%%%%%%Ih1=high pass filtered version of I1
```

```
%%%%%%Ih2 =high pass filtered version of I2
%%%%%% High pass filtered version of the images are obtained via a 3x3 pixel std  approximation
of the Laplacian operator
h = fspecial('laplacian');
Ih1 = imfilter(I1,h,'replicate');
Ih2 = imfilter(I2,h,'replicate');
mI1=mean2(I1);
mI2=mean2(I2);
X1=sum(sum((Ih1-mI1).* (Ih2-mI2)));
X2=sum(sum((Ih1-mI1).*(Ih1-mI1)));
X3=sum(sum((Ih2-mI2).*(Ih2-mI2)));
CP=X1./sqrt(X2.*X3);% Correlation parameter ,it should be close to unity for an optimal effect of
edge preservation
out(1) = PSNR(1,1,1);
out(1) = out(1)+PSNR(1,1,2);
out(1) = out(1)+PSNR(1,1,3);
out(2) = MSE(1,1,1);
out(2) = out(2)+MSE(1,1,2);
out(2) = out(2)+MSE(1,1,3);
out(3) = CP(1,1,1);
out(3) = out(3)+CP(1,1,2);
out(3) = out(3)+CP(1,1,3);
out./3
```

MSSIM:

```
/*
 * The equivalent of Zhou Wang's SSIM matlab code using OpenCV.
 * from http://www.cns.nyu.edu/~zwang/files/research/ssim/index.html
 * The measure is described in :
 * "Image quality assessment: From error measurement to structural
similarity"
 * C++ code by Rabah Mehdi. http://mehdi.rabah.free.fr/SSIM
 *
 * This implementation is under the public domain.
 * @see http://creativecommons.org/licenses/publicdomain/
 * The original work may be under copyrights.
 */

#include <cv.h>
#include <highgui.h>
#include <iostream>
using namespace std;

/*
 * Parameters : complete path to the two image to be compared
 * The file format must be supported by your OpenCV build
 */
int main(int argc, char** argv)
{
      if(argc!=3)
            return -1;

      // default settings
      double C1 = 6.5025, C2 = 58.5225;

      IplImage
```

```
            *img1=NULL, *img2=NULL, *img1_img2=NULL,
            *img1_temp=NULL, *img2_temp=NULL,
            *img1_sq=NULL, *img2_sq=NULL,
            *mu1=NULL, *mu2=NULL,
            *mu1_sq=NULL, *mu2_sq=NULL, *mu1_mu2=NULL,
            *sigma1_sq=NULL, *sigma2_sq=NULL, *sigma12=NULL,
            *ssim_map=NULL, *temp1=NULL, *temp2=NULL, *temp3=NULL;


      /***************************** INITS
*********************************/
      img1_temp = cvLoadImage(argv[1]);
      img2_temp = cvLoadImage(argv[2]);

      if(img1_temp==NULL || img2_temp==NULL)
            return -1;

      int x=img1_temp->width, y=img1_temp->height;
      int nChan=img1_temp->nChannels, d=IPL_DEPTH_32F;
      CvSize size = cvSize(x, y);

      img1 = cvCreateImage( size, d, nChan);
      img2 = cvCreateImage( size, d, nChan);

      cvConvert(img1_temp, img1);
      cvConvert(img2_temp, img2);
      cvReleaseImage(&img1_temp);
      cvReleaseImage(&img2_temp);


      img1_sq = cvCreateImage( size, d, nChan);
      img2_sq = cvCreateImage( size, d, nChan);
      img1_img2 = cvCreateImage( size, d, nChan);

      cvPow( img1, img1_sq, 2 );
      cvPow( img2, img2_sq, 2 );
      cvMul( img1, img2, img1_img2, 1 );

      mu1 = cvCreateImage( size, d, nChan);
      mu2 = cvCreateImage( size, d, nChan);

      mu1_sq = cvCreateImage( size, d, nChan);
      mu2_sq = cvCreateImage( size, d, nChan);
      mu1_mu2 = cvCreateImage( size, d, nChan);


      sigma1_sq = cvCreateImage( size, d, nChan);
      sigma2_sq = cvCreateImage( size, d, nChan);
      sigma12 = cvCreateImage( size, d, nChan);

      temp1 = cvCreateImage( size, d, nChan);
      temp2 = cvCreateImage( size, d, nChan);
      temp3 = cvCreateImage( size, d, nChan);

      ssim_map = cvCreateImage( size, d, nChan);
      /*************************** END INITS
*********************************/


      //////////////////////////////////////////////////////////////////
/////
      // PRELIMINARY COMPUTING
```

```
	cvSmooth( img1, mu1, CV_GAUSSIAN, 11, 11, 1.5 );
	cvSmooth( img2, mu2, CV_GAUSSIAN, 11, 11, 1.5 );

	cvPow( mu1, mu1_sq, 2 );
	cvPow( mu2, mu2_sq, 2 );
	cvMul( mu1, mu2, mu1_mu2, 1 );


	cvSmooth( img1_sq, sigma1_sq, CV_GAUSSIAN, 11, 11, 1.5 );
	cvAddWeighted( sigma1_sq, 1, mu1_sq, -1, 0, sigma1_sq );

	cvSmooth( img2_sq, sigma2_sq, CV_GAUSSIAN, 11, 11, 1.5 );
	cvAddWeighted( sigma2_sq, 1, mu2_sq, -1, 0, sigma2_sq );

	cvSmooth( img1_img2, sigma12, CV_GAUSSIAN, 11, 11, 1.5 );
	cvAddWeighted( sigma12, 1, mu1_mu2, -1, 0, sigma12 );


	//////////////////////////////////////////////////////////////////////
/////
	// FORMULA

	// (2*mu1_mu2 + C1)
	cvScale( mu1_mu2, temp1, 2 );
	cvAddS( temp1, cvScalarAll(C1), temp1 );

	// (2*sigma12 + C2)
	cvScale( sigma12, temp2, 2 );
	cvAddS( temp2, cvScalarAll(C2), temp2 );

	// ((2*mu1_mu2 + C1).*(2*sigma12 + C2))
	cvMul( temp1, temp2, temp3, 1 );

	// (mu1_sq + mu2_sq + C1)
	cvAdd( mu1_sq, mu2_sq, temp1 );
	cvAddS( temp1, cvScalarAll(C1), temp1 );

	// (sigma1_sq + sigma2_sq + C2)
	cvAdd( sigma1_sq, sigma2_sq, temp2 );
	cvAddS( temp2, cvScalarAll(C2), temp2 );

	// ((mu1_sq + mu2_sq + C1).*(sigma1_sq + sigma2_sq + C2))
	cvMul( temp1, temp2, temp1, 1 );

	// ((2*mu1_mu2 + C1).*(2*sigma12 + C2))./((mu1_sq + mu2_sq +
C1).*(sigma1_sq + sigma2_sq + C2))
	cvDiv( temp3, temp1, ssim_map, 1 );


	CvScalar index_scalar = cvAvg( ssim_map );

	// through observation, there is approximately
	// 1% error max with the original matlab program

	cout << "(R, G & B SSIM index)" << endl ;
	cout << index_scalar.val[2] * 100 << "%" << endl ;
	cout << index_scalar.val[1] * 100 << "%" << endl ;
	cout << index_scalar.val[0] * 100 << "%" << endl ;

	// if you use this code within a program
	// don't forget to release the IplImages
	return 0;
```

```
}
```

Haar(level 3) with Lanzose(sinc 3):

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <cv.h>
#include <string.h>
#include <highgui.h>

//.....................................
Scaling.................................................//

#define pi 3.14159265

int a=3;

double Lanczos(double x)
{
    if(x<0.000001 && x>-0.000001)
        return 1;
    else if(-a < x && a > x)
        return (a*sin(pi*x)*sin(pi*x/a))/(pi*pi*x*x);
    else
        return 0;
}

void Lanczose_Scale(IplImage *in, IplImage *out, int owidth, int oheight)
{
    int channels = in->nChannels;
    int iwidth = in->width;
    int iheight = in->height;

    int i,j,k,x,y;
    for(x=0;x<owidth;x++)
        for(y=0;y<oheight;y++)
        {
            double x0 = floor(x*1.0*iwidth/owidth);
            double y0 = floor(y*1.0*iheight/oheight);
            CvScalar temp, temp2;
            for(k=0;k<channels;k++)
                temp2.val[k] = 0;
            for(i=x0-a+1;i<=x0+a;i++)
                for(j=y0-a+1;j<=y0+a;j++)
                    if(i>=0 && i<iwidth && j>=0 && j<iheight)
                    {
                        temp = cvGet2D(in,j,i);
                        for(k=0;k<channels;k++)
                        {
                            temp2.val[k] += temp.val[k]*Lanczos(x0-
i)*Lanczos(y0-j);
                        }
                    }
            cvSet2D(out,y,x,temp2);
        }
}
```

```
//.....................................................................
.....//

void Haar1DRow(double *image, int row, int col, int width, int channels)
{
    double *data = new double [col*channels];
    int i,j,k;
    for(i=0;i<row;i++)
    {
        for(j=0;j<col/2;j++)
        {
            for(k=0;k<channels;k++)
            {
                double a = image[(i*width + 2*j)*channels + k];
                double b = image[(i*width + (2*j+1))*channels + k];
                data[j*channels + k] = (a+b)/sqrt(2);
                data[(col/2+j)*channels + k] = (a-b)/sqrt(2);
            }
        }
        for(j=0;j<col/2;j++)
        {   for(k=0;k<channels;k++)
            {
                image[(i*width+j)*channels + k] = data[j*channels + k];
                image[(i*width+col/2+j)*channels + k] =
data[(col/2+j)*channels + k];
            }
        }
    }

}

void UnHaar1DRow(double *image, int row, int col, int width, int channels)
{
    double *data = new double [col*channels];
    int i,j,k;
    for(i=0;i<row;i++)
    {
        for(j=0;j<col/2;j++)
        {
            for(k=0;k<channels;k++)
            {
                double a = image[(i*width+j)*channels + k];
                double b = image[(i*width+col/2+j)*channels + k];
                data[j*channels + k] = (a+b)/sqrt(2);
                data[(col/2+j)*channels + k] = (a-b)/sqrt(2);
            }
        }
        for(j=0;j<col/2;j++)
        {   for(k=0;k<channels;k++)
            {
                image[(i*width + 2*j)*channels + k] = data[j*channels + k];
                image[(i*width + (2*j+1))*channels + k] =
data[(col/2+j)*channels + k];
            }
        }
    }

}

void Haar1DCol(double *image, int row, int col, int width, int channels)
{
    double *data = new double [row*channels];
```

```
    int i,j,k;
    for(i=0;i<col;i++)
    {
        for(j=0;j<row/2;j++)
        {
            for(k=0;k<channels;k++)
            {
                double a = image[(2*j*width + i)*channels + k];
                double b = image[((2*j+1)*width + i)*channels + k];
                data[j*channels + k] = (a+b)/sqrt(2);
                data[(row/2+j)*channels + k] = (a-b)/sqrt(2);
            }
        }
        for(j=0;j<row/2;j++)
        {   for(k=0;k<channels;k++)
            {
                image[(j*width+i)*channels + k] = data[j*channels + k];
                image[((row/2+j)*width+i)*channels + k] =
data[(row/2+j)*channels + k];
            }
        }
    }
}

void UnHaar1DCol(double *image, int row, int col, int width, int channels)
{
    double *data = new double [row*channels];
    int i,j,k;
    for(i=0;i<col;i++)
    {
        for(j=0;j<row/2;j++)
        {
            for(k=0;k<channels;k++)
            {
                double a = image[(j*width+i)*channels + k];
                double b = image[((row/2+j)*width+i)*channels + k];
                data[j*channels + k] = (a+b)/sqrt(2);
                data[(row/2+j)*channels + k] = (a-b)/sqrt(2);
            }
        }
        for(j=0;j<row/2;j++)
        {   for(k=0;k<channels;k++)
            {
                image[(2*j*width + i)*channels + k] = data[j*channels + k];
                image[((2*j+1)*width + i)*channels + k] =
data[(row/2+j)*channels + k];
            }
        }
    }
}

void Haar2D(double *image, int width, int height, int channels)
{
    int level=3, crow = height, ccol = width;

    while(level--)
    {
        Haar1DRow(image, crow, ccol, width, channels);
        Haar1DCol(image, crow, ccol, width, channels);
        ccol /= 2;
        crow /= 2;
    }
```

```
}

void Haar_Reconstruct(double *image, int width, int height, int channels)
{
    int level=3, crow = height/4, ccol = width/4;

    while(level--)
    {
        UnHaar1DCol(image, crow, ccol, width, channels);
        UnHaar1DRow(image, crow, ccol, width, channels);
        ccol *= 2;
        crow *= 2;
    }
}

void Denoise(double *data, int width, int height, int nChannels)
{
    int i,j,k, l;
    //double sigma[3],temp, thr[3];
    double sigx[2000][3], sigy[2000][3], temp, th;

    for(l=1;l<=3;l++)
    {
        /*sigma[0] = sigma[1] = sigma[2] = 0.0;
        for(i=0;i<height/pow(2,l);i++)
            for(j=0;j<width/pow(2,l);j++)
                for(k=0;k<nChannels;k++)
                {
                    sigma[k] += abs(data[(i*width + j)*nChannels + k]);
                }
        sigma[0] /= 0.6745*width*height/pow(2,2*l);
        sigma[1] /= 0.6745*width*height/pow(2,2*l);
        sigma[2] /= 0.6745*width*height/pow(2,2*l);

        thr[0] =
sigma[0]*sqrt(log(width*height/pow(2,2*l))/(width*height/pow(2,2*l)));
        thr[1] =
sigma[1]*sqrt(log(width*height/pow(2,2*l))/(width*height/pow(2,2*l)));
        thr[2] =
sigma[2]*sqrt(log(width*height/pow(2,2*l))/(width*height/pow(2,2*l)));
        */

        for(i=0;i<2000;i++)
            sigx[i][0] = sigx[i][1] = sigx[i][2] = sigy[i][0] = sigy[i][1]
= sigy[i][2] = 0.0;

        for(i=0;i<height/pow(2,l-1);i++)
            for(j=0;j<width/pow(2,l-1);j++)
                for(k=0;k<3;k++)
                {
                    sigx[j][k] += abs(data[(i*width + j)*nChannels + k]);
                    sigy[i][k] += abs(data[(i*width + j)*nChannels + k]);
                }

        for(i=0;i<height/pow(2,l-1);i++)
            for(j=0;j<width/pow(2,l-1);j++)
                for(k=0;k<nChannels;k++)
                {
                    temp = data[(i*width + j)*nChannels + k];

                    double sig = (sigx[j][k]+sigy[i][k])/2;
                    sig /= 0.6745*(width+height)/pow(2,l-1);
```

```
                    th = sig*sqrt(log((width+height)/pow(2,l-
1))/((width+height)/pow(2,l-1)));

                    if(abs(temp) < th)
                        data[(i*width + j)*nChannels + k] = 0.0;
                    else if(temp > th)
                        data[(i*width + j)*nChannels + k] -= th;
                    else
                        data[(i*width + j)*nChannels + k] += th;
                }
    }
}

void Wavelet_Denoise(IplImage *image)
{
    int i,j,k, width, height, nChannels;
    CvScalar temp;
    width = image->width;
    height = image->height;
    nChannels = image->nChannels;
    double *data = new double [width * height * nChannels];

    for(i=0;i<height;i++)
        for(j=0;j<width;j++)
        {
            temp = cvGet2D(image, i, j);
            for(k=0;k<nChannels;k++)
                data[(i*width + j)*nChannels + k] = temp.val[k];
        }

    Haar2D(data, width, height, nChannels);

    Denoise(data, width, height, nChannels);

    Haar_Reconstruct(data, width, height, nChannels);

    for(i=0;i<height;i++)
        for(j=0;j<width;j++)
        {
            for(k=0;k<nChannels;k++)
                temp.val[k] = data[(i*width + j)*nChannels + k];
            cvSet2D(image, i, j, temp);
        }

    delete [] data;
}

int main()
{
    int o_height, o_width;
    char input[20], output[20];
    scanf("%s%s%d%d", input, output, &o_width, &o_height);

    IplImage *in_image, *out_image;
    in_image = cvLoadImage(input,-1);

    if(!in_image)
    {
        printf("Error!\n");
        return 0;
    }
```

```
    out_image =
cvCreateImage(cvSize(o_width,o_height),IPL_DEPTH_8U,in_image->nChannels);

    Lanczose_Scale(in_image, out_image, o_width, o_height);
    Wavelet_Denoise(out_image);

    cvSaveImage(output,out_image);
}
```

Haar(level 3) with Bilinear:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <cv.h>
#include <string.h>
#include <highgui.h>

//.....................................
Scaling.................................................//

double valueat(double intensity1,double intensity2,double v1,double
v2,double v)
{
    if(v1==v2)
        return intensity1;
    else
        return((((v2-v)/(v2-v1))*intensity2) +(((v-v1)/(v2-
v1))*intensity1));
}

void bilinear(IplImage* img, IplImage *imgfinal, double horizontal_scale,
double vertical_scale)
{
    int
orig_height,orig_width,orig_step,orig_channels,new_step,new_channels,new_he
ight,new_width,i,j,k;
    CvScalar v;
    orig_height=img->height;
    orig_width=img->width;
    orig_step=img->widthStep;
    new_step=orig_step*horizontal_scale;
    new_channels=orig_channels=img->nChannels;
    uchar *orig_data = (uchar *)img->imageData; //Get the image data in
'data'
    new_height=orig_height*vertical_scale;                    //new image
height
      new_width=orig_width*horizontal_scale;                        //new
image width
      uchar *new_data =(uchar*) imgfinal->imageData;
      for(i=0;i<new_height;i++)
            for(j=0;j<new_width;j++)
                  for(k=0;k<new_channels;k++)
                        new_data[(int)(i*new_step+j*new_channels+k)]=0;
    for(i=0;i<new_height;i++)
    {
        for(j=0;j<new_width;j++)
        {
            int x1=floor((double)j/(double)horizontal_scale);
            int y1=floor((double)i/(double)vertical_scale);
```

```
            int x2=ceil((double)j/(double)horizontal_scale);
            int y2=ceil((double)i/(double)vertical_scale);
            double x = (double)j/(double)horizontal_scale;
            double y = (double)i/(double)vertical_scale;
            for(k=0;k<new_channels;k++)
            {

                double r1=valueat((double)orig_data[(int)(y1*orig_step
+x1*orig_channels+k)],(double)orig_data[(int)(y1*orig_step
+x2*orig_channels+k)],(double)x1,(double)x2,x);
                double r2=valueat((double)orig_data[(int)(y2*orig_step
+x1*orig_channels+k)],(double)orig_data[(int)(y2*orig_step
+x2*orig_channels+k)],(double)x1,(double)x2,x);
                v.val[k]=valueat(r1,r2,(double)y1,(double)y2,y);

//new_data[(int)(i*new_step+j*new_channels+k)]=orig_data[(int)(temp_height*
orig_step + temp_width*orig_channels+k)];
            }
            cvSet2D(imgfinal,i,j,v);
        }
    }
}

//.....................................................................
.....//

void Haar1DRow(double *image, int row, int col, int width, int channels)
{
    double *data = new double [col*channels];
    int i,j,k;
    for(i=0;i<row;i++)
    {
        for(j=0;j<col/2;j++)
        {
            for(k=0;k<channels;k++)
            {
                double a = image[(i*width + 2*j)*channels + k];
                double b = image[(i*width + (2*j+1))*channels + k];
                data[j*channels + k] = (a+b)/sqrt(2);
                data[(col/2+j)*channels + k] = (a-b)/sqrt(2);
            }
        }
        for(j=0;j<col/2;j++)
        {   for(k=0;k<channels;k++)
            {
                image[(i*width+j)*channels + k] = data[j*channels + k];
                image[(i*width+col/2+j)*channels + k] =
data[(col/2+j)*channels + k];
            }
        }
    }

}

void UnHaar1DRow(double *image, int row, int col, int width, int channels)
{
    double *data = new double [col*channels];
    int i,j,k;
    for(i=0;i<row;i++)
    {
        for(j=0;j<col/2;j++)
        {
```

```
            for(k=0;k<channels;k++)
            {
                double a = image[(i*width+j)*channels + k];
                double b = image[(i*width+col/2+j)*channels + k];
                data[j*channels + k] = (a+b)/sqrt(2);
                data[(col/2+j)*channels + k] = (a-b)/sqrt(2);
            }
        }
        for(j=0;j<col/2;j++)
        {   for(k=0;k<channels;k++)
            {
                image[(i*width + 2*j)*channels + k] = data[j*channels + k];
                image[(i*width + (2*j+1))*channels + k] =
data[(col/2+j)*channels + k];
            }
        }
    }

}

void Haar1DCol(double *image, int row, int col, int width, int channels)
{
    double *data = new double [row*channels];
    int i,j,k;
    for(i=0;i<col;i++)
    {
        for(j=0;j<row/2;j++)
        {
            for(k=0;k<channels;k++)
            {
                double a = image[(2*j*width + i)*channels + k];
                double b = image[((2*j+1)*width + i)*channels + k];
                data[j*channels + k] = (a+b)/sqrt(2);
                data[(row/2+j)*channels + k] = (a-b)/sqrt(2);
            }
        }
        for(j=0;j<row/2;j++)
        {   for(k=0;k<channels;k++)
            {
                image[(j*width+i)*channels + k] = data[j*channels + k];
                image[((row/2+j)*width+i)*channels + k] =
data[(row/2+j)*channels + k];
            }
        }
    }
}

void UnHaar1DCol(double *image, int row, int col, int width, int channels)
{
    double *data = new double [row*channels];
    int i,j,k;
    for(i=0;i<col;i++)
    {
        for(j=0;j<row/2;j++)
        {
            for(k=0;k<channels;k++)
            {
                double a = image[(j*width+i)*channels + k];
                double b = image[((row/2+j)*width+i)*channels + k];
                data[j*channels + k] = (a+b)/sqrt(2);
                data[(row/2+j)*channels + k] = (a-b)/sqrt(2);
            }
```

```
        }
        for(j=0;j<row/2;j++)
        {   for(k=0;k<channels;k++)
            {
                image[(2*j*width + i)*channels + k] = data[j*channels + k];
                image[((2*j+1)*width + i)*channels + k] =
data[(row/2+j)*channels + k];
            }
        }
    }
}

void Haar2D(double *image, int width, int height, int channels)
{
    int level=3, crow = height, ccol = width;

    while(level--)
    {
        Haar1DRow(image, crow, ccol, width, channels);
        Haar1DCol(image, crow, ccol, width, channels);
        ccol /= 2;
        crow /= 2;
    }
}

void Haar_Reconstruct(double *image, int width, int height, int channels)
{
    int level=3, crow = height/4, ccol = width/4;

    while(level--)
    {
        UnHaar1DCol(image, crow, ccol, width, channels);
        UnHaar1DRow(image, crow, ccol, width, channels);
        ccol *= 2;
        crow *= 2;
    }
}

void Denoise(double *data, int width, int height, int nChannels)
{
    int i,j,k, l;
    //double sigma[3],temp, thr[3];
    double sigx[2000][3], sigy[2000][3], temp, th;

    for(l=1;l<=3;l++)
    {
        /*sigma[0] = sigma[1] = sigma[2] = 0.0;
        for(i=0;i<height/pow(2,l);i++)
            for(j=0;j<width/pow(2,l);j++)
                for(k=0;k<nChannels;k++)
                {
                    sigma[k] += abs(data[(i*width + j)*nChannels + k]);
                }
        sigma[0] /= 0.6745*width*height/pow(2,2*l);
        sigma[1] /= 0.6745*width*height/pow(2,2*l);
        sigma[2] /= 0.6745*width*height/pow(2,2*l);

        thr[0] =
sigma[0]*sqrt(log(width*height/pow(2,2*l))/(width*height/pow(2,2*l)));
        thr[1] =
sigma[1]*sqrt(log(width*height/pow(2,2*l))/(width*height/pow(2,2*l)));
```

```
        thr[2] =
sigma[2]*sqrt(log(width*height/pow(2,2*l))/(width*height/pow(2,2*l)));
        */

        for(i=0;i<2000;i++)
            sigx[i][0] = sigx[i][1] = sigx[i][2] = sigy[i][0] = sigy[i][1]
= sigy[i][2] = 0.0;

        for(i=0;i<height/pow(2,l-1);i++)
            for(j=0;j<width/pow(2,l-1);j++)
                for(k=0;k<3;k++)
                {
                    sigx[j][k] += abs(data[(i*width + j)*nChannels + k]);
                    sigy[i][k] += abs(data[(i*width + j)*nChannels + k]);
                }

        for(i=0;i<height/pow(2,l-1);i++)
            for(j=0;j<width/pow(2,l-1);j++)
                for(k=0;k<nChannels;k++)
                {
                    temp = data[(i*width + j)*nChannels + k];

                    double sig = (sigx[j][k]+sigy[i][k])/2;
                    sig /= 0.6745*(width+height)/pow(2,l-1);
                    th = sig*sqrt(log((width+height)/pow(2,l-
1))/((width+height)/pow(2,l-1)));

                    if(abs(temp) < th)
                        data[(i*width + j)*nChannels + k] = 0.0;
                    else if(temp > th)
                        data[(i*width + j)*nChannels + k] -= th;
                    else
                        data[(i*width + j)*nChannels + k] += th;
                }
    }
}

void Wavelet_Denoise(IplImage *image)
{
    int i,j,k, width, height, nChannels;
    CvScalar temp;
    width = image->width;
    height = image->height;
    nChannels = image->nChannels;
    double *data = new double [width * height * nChannels];

    for(i=0;i<height;i++)
        for(j=0;j<width;j++)
        {
            temp = cvGet2D(image, i, j);
            for(k=0;k<nChannels;k++)
                data[(i*width + j)*nChannels + k] = temp.val[k];
        }

    Haar2D(data, width, height, nChannels);

    Denoise(data, width, height, nChannels);

    Haar_Reconstruct(data, width, height, nChannels);

    for(i=0;i<height;i++)
        for(j=0;j<width;j++)
```

```
        {
            for(k=0;k<nChannels;k++)
                temp.val[k] = data[(i*width + j)*nChannels + k];
            cvSet2D(image, i, j, temp);
        }

    delete [] data;
}

int main()
{
    int o_height, o_width;
    char input[20], output[20];
    scanf("%s%s%d%d", input, output, &o_width, &o_height);

    IplImage *in_image, *out_image;
    in_image = cvLoadImage(input,-1);

    if(!in_image)
    {
        printf("Error!\n");
        return 0;
    }

    out_image =
cvCreateImage(cvSize(o_width,o_height),IPL_DEPTH_8U,in_image->nChannels);

    bilinear(in_image, out_image, o_width*1.0/in_image->width,
o_height*1.0/in_image->height);
    Wavelet_Denoise(out_image);

    cvSaveImage(output,out_image);
}
```

Haar (level 3) with bicubic:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <cv.h>
#include <string.h>
#include <highgui.h>

//.....................................
Scaling.................................................//

double a[4][4];
double calculatep(double x,double y)
{
      double p;
      p=a[0][0] + a[0][1]*y + a[0][2]*y*y + a[0][3]*y*y*y + a[1][0]*x +
a[1][1]*x*y + a[1][2]*x*y*y + a[1][3]*x*y*y*y + a[2][0]*x*x + a[2][1]*x*x*y
+ a[2][2]*x*x*y*y + a[2][3]*x*x*y*y*y + a[3][0]*x*x*x + a[3][1]*x*x*x*y
+a[3][2]*x*x*x*y*y +a[3][3]*x*x*x*y*y*y;
      return p;
}

void bicubic(IplImage* img, IplImage* imgfinal, double m, double n)
{
    CvScalar v;
```

```
	int height,width,step,channels;
	uchar *data;
	int i,j,k,f,g,h;
	height=img->height;
	width=img->width;
	step=img->widthStep;
	channels=img->nChannels;
	data	= (uchar *)img->imageData;	//Get the image data in
'data'
	int ht=height*n;		//new image height
	int wt=width*m;			//new image width
	uchar *datafinal =(uchar*) imgfinal->imageData;
	for(i=0;i<height*n;i++)
		for(j=0;j<width*m;j++)
			for(k=0;k<channels;k++)
				datafinal[(int)(i*step*m+j*channels+k)]=0;

	double w[4],x[4],y[4],z[4],i1,j1;	//w->f00,f10,f01,f11; x-
>fx00,fx10,fx01,fx11; y->fy00,fy10,fy01,fy11; z->fxy0,fxy10,fxy01,fxy11
	for(f=0;f<ht;f++)		//Traverse for all the pixels in new
image
	{
		for(g=0;g<wt;g++)
		{
			i1=((double)f)/n;
			j1=((double)g)/m;
			i=floor(i1);		//Map to the original image
pixels
			j=floor(j1);
			i1=i1-i;
			j1=j1-j;

			for(k=0;k<channels;k++)
			{

				//Calculate f values
				w[0]=data[i*step+j*channels+k];
				w[1]=data[i*step+(j+1)*channels+k];
				w[2]=data[(i+1)*step+j*channels+k];
				w[3]=data[(i+1)*step+(j+1)*channels+k];

				//Calulate fy values
				if((i-1)>=0)//Check if value is going beyond the
scope of image
				{
					y[0] = 0.5*(data[(i+1)*step+j*channels+k] -
data[(i-1)*step+j*channels+k]);
					y[1] = 0.5*(data[(i+1)*step+(j+1)*channels+k]
- data[(i-1)*step+(j+1)*channels+k]);
				}
				else// Selecting the neighbouring pixels in the
place
				{
					y[0]=0.5*(data[(i+1)*step+j*channels+k]-
data[(i)*step+j*channels+k]);
					y[1]=0.5*(data[(i+1)*step+(j+1)*channels+k]-
data[(i)*step+(j+1)*channels+k]);
				}
				if((i+2)<=(height-1))
				{
```

```
                                y[2] = 0.5*(data[(i+2)*step+j*channels+k] -
data[(i)*step+j*channels+k]);
                                y[3] = 0.5*(data[(i+2)*step+(j+1)*channels+k]
- data[(i)*step+(j+1)*channels+k]);
                        }
                        else
                        {
                                y[2] = 0.5*(data[(i+1)*step+j*channels+k] -
data[(i)*step+j*channels+k]);
                                y[3] = 0.5*(data[(i+1)*step+(j+1)*channels+k]
- data[(i)*step+(j+1)*channels+k]);
                        }

                        //Calculate fx values
                        if((j-1)>=0)
                        {
                                x[0]=0.5*(data[i*step+(j+1)*channels+k] -
data[i*step+(j-1)*channels+k]);
                                x[2]=0.5*(data[(i+1)*step+(j+1)*channels+k] -
data[(i+1)*step+(j-1)*channels+k]);
                        }
                        else
                        {
                                x[0]=0.5*(data[i*step+(j+1)*channels+k] -
data[i*step+(j)*channels+k]);
                                x[2]=0.5*(data[(i+1)*step+(j+1)*channels+k] -
data[(i+1)*step+(j)*channels+k]);
                        }
                        if((j+2)<=(width-1))
                        {
                                x[1]=0.5*(data[i*step+(j+2)*channels+k] -
data[(i)*step+j*channels+k]);
                                x[3]=0.5*(data[(i+1)*step+(j+2)*channels+k] -
data[(i+1)*step+(j)*channels+k]);
                        }
                        else
                        {
                                x[1]=0.5*(data[i*step+(j+1)*channels+k] -
data[(i)*step+j*channels+k]);
                                x[3]=0.5*(data[(i+1)*step+(j+1)*channels+k] -
data[(i+1)*step+(j)*channels+k]);
                        }

                        //Calculate fxy values
                        if((i-1)>=0 && (j-1)>=0)
                                z[0]=0.25*(data[(i+1)*step+(j+1)*channels+k]
- data[(i-1)*step+(j+1)*channels+k] - data[(i+1)*step+(j-1)*channels+k] +
data[(i-1)*step+(j-1)*channels+k]);
                        else if((i-1)>=0)
                        {
                                z[0]=0.25*(data[(i+1)*step+(j+1)*channels+k]
- data[(i-1)*step+(j+1)*channels+k] - data[(i+1)*step+(j)*channels+k] +
data[(i-1)*step+(j)*channels+k]);
                        }
                        else if((j-1)>=0)
                        {
                                z[0]=0.25*(data[(i+1)*step+(j+1)*channels+k]
- data[(i)*step+(j+1)*channels+k] - data[(i+1)*step+(j-1)*channels+k] +
data[(i)*step+(j-1)*channels+k]);
                        }
                        else
                        {
```

```
                                z[0]=0.25*(data[(i+1)*step+(j+1)*channels+k]
- data[(i)*step+(j+1)*channels+k] - data[(i+1)*step+(j)*channels+k] +
data[(i)*step+(j)*channels+k]);
                        }
                        if((j+2)<width && (i-1)>=0)
                        {
                                z[1]=0.25*(data[(i+1)*step+(j+2)*channels+k]-
data[(i-1)*step+(j+2)*channels+k]-data[(i+1)*step+(j)*channels+k]+data[(i-
1)*step+(j)*channels+k]);
                        }
                        else if((j+2)<width)
                        {
                                z[1]=0.25*(data[(i+1)*step+(j+2)*channels+k]-
data[(i)*step+(j+2)*channels+k]-
data[(i+1)*step+(j)*channels+k]+data[(i)*step+(j)*channels+k]);
                        }
                        else if((i-1)>=0)
                        {
                                z[1]=0.25*(data[(i+1)*step+(j+1)*channels+k]-
data[(i-1)*step+(j+1)*channels+k]-data[(i+1)*step+(j)*channels+k]+data[(i-
1)*step+(j)*channels+k]);
                        }
                        else
                        {
                                z[1]=0.25*(data[(i+1)*step+(j+1)*channels+k]-
data[(i)*step+(j+1)*channels+k]-
data[(i+1)*step+(j)*channels+k]+data[(i)*step+(j)*channels+k]);
                        }
                        if((i+2)<height && (j-1)>=0)
                        {
                                z[2]=0.25*(data[(i+2)*step+(j+1)*channels+k]
- data[(i)*step+(j+1)*channels+k] - data[(i+2)*step+(j-1)*channels+k] +
data[(i)*step+(j-1)*channels+k]);
                        }
                        else if((i+2)<height)
                        {
                                z[2]=0.25*(data[(i+2)*step+(j+1)*channels+k]
- data[(i)*step+(j+1)*channels+k] - data[(i+2)*step+(j)*channels+k] +
data[(i)*step+(j)*channels+k]);
                        }
                        else if((j-1)>=0)
                        {
                                z[2]=0.25*(data[(i+1)*step+(j+1)*channels+k]
- data[(i)*step+(j+1)*channels+k] - data[(i+1)*step+(j-1)*channels+k] +
data[(i)*step+(j-1)*channels+k]);
                        }
                        else
                        {
                                z[2]=0.25*(data[(i+1)*step+(j+1)*channels+k]
- data[(i)*step+(j+1)*channels+k] - data[(i+1)*step+(j)*channels+k] +
data[(i)*step+(j)*channels+k]);
                        }
                        if((j+2)<width && (i+2)<height)
                        {
                                z[3]=0.25*(data[(i+2)*step+(j+2)*channels+k]
- data[(i)*step+(j+2)*channels+k] - data[(i+2)*step+(j)*channels+k] +
data[i*step+j*channels+k]);
                        }
                        else if((j+2)<width)
                        {
```

```
                                z[3]=0.25*(data[(i+1)*step+(j+2)*channels+k]
- data[(i)*step+(j+2)*channels+k] - data[(i+1)*step+(j)*channels+k] +
data[i*step+j*channels+k]);
                        }
                        else if((i+2)<height)
                        {
                                z[3]=0.25*(data[(i+2)*step+(j+1)*channels+k]
- data[(i)*step+(j+1)*channels+k] - data[(i+2)*step+(j)*channels+k] +
data[i*step+j*channels+k]);
                        }
                        else
                        {
                                z[3]=0.25*(data[(i+1)*step+(j+1)*channels+k]
- data[(i)*step+(j+1)*channels+k] - data[(i+1)*step+(j)*channels+k] +
data[i*step+j*channels+k]);
                        }


                        //Find the sixteen coefficients required for the
equation to find channel values
                        a[0][0]=w[0];
                        a[1][0]=y[0];
                        a[2][0]=0-(3*w[0])+(3*w[2])-(2*y[0])-y[2];
                        a[3][0]=(2*w[0])-(2*w[2])+y[0]+y[2];
                        a[0][1]=x[0];
                        a[1][1]=z[0];
                        a[2][1]=0-(3*x[0])+(3*x[2])-(2*z[0])-z[2];
                        a[3][1]=(2*x[0])-(2*x[2])+z[0]+z[2];
                        a[0][2]=0-(3*w[0])+(3*w[1])-(2*x[0])-x[1];
                        a[1][2]=0-(3*y[0])+(3*y[1])-(2*z[0])-z[1];
                        a[2][2]=(9*(w[0]-w[1]-w[2]+w[3]))+(6*(x[0]-
x[2]+y[0]-y[1]))+(3*(x[1]-x[3]+y[2]-y[3]))+(4*z[0])+(2*z[1])+(2*z[2])+z[3];
                        a[3][2]=(6*(w[1]-w[0]+w[2]-w[3]))+(3*(y[1]-
y[0]+y[3]-y[2]))+(2*(x[3]-x[1]-z[0]-z[2]))+(4*x[2])-4*x[0]-z[1]-z[3];
                        a[0][3]=(2*w[0])-(2*w[1])+x[0]+x[1];
                        a[1][3]=(2*y[0])-(2*y[1])+z[0]+z[1];
                        a[2][3]=(6*(w[1]-w[0]+w[2]-w[3]))+(3*(x[2]+x[3]-
x[0]-x[1]))+(2*(y[3]-y[2]-z[0]-z[1])) -(4*y[0])+(4*y[1])-z[2]-z[3];
                        a[3][3]=(4*(w[0]-w[1]-w[2]+w[3]))+(2*(x[0]+x[1]-
x[2]-x[3]+y[0]-y[1]+y[2]-y[3]))+z[0]+z[1]+z[2]+z[3];

                        //Calculate the values for the k channels in the
image
                        v.val[k]=(int)(calculatep(j1,i1));

                        //datafinal[(int)(f*step*m +
g*channels+k)]=(int)(calculatep(j1,i1));
                }
                //Assign the values to the image
                cvSet2D(imgfinal,f,g,v);
            }
        }
}

//.....................................................................
.....//

void Haar1DRow(double *image, int row, int col, int width, int channels)
{
    double *data = new double [col*channels];
    int i,j,k;
    for(i=0;i<row;i++)
```

```
    {
        for(j=0;j<col/2;j++)
        {
            for(k=0;k<channels;k++)
            {
                double a = image[(i*width + 2*j)*channels + k];
                double b = image[(i*width + (2*j+1))*channels + k];
                data[j*channels + k] = (a+b)/sqrt(2);
                data[(col/2+j)*channels + k] = (a-b)/sqrt(2);
            }
        }
        for(j=0;j<col/2;j++)
        {   for(k=0;k<channels;k++)
            {
                image[(i*width+j)*channels + k] = data[j*channels + k];
                image[(i*width+col/2+j)*channels + k] =
data[(col/2+j)*channels + k];
            }
        }
    }

}

void UnHaar1DRow(double *image, int row, int col, int width, int channels)
{
    double *data = new double [col*channels];
    int i,j,k;
    for(i=0;i<row;i++)
    {
        for(j=0;j<col/2;j++)
        {
            for(k=0;k<channels;k++)
            {
                double a = image[(i*width+j)*channels + k];
                double b = image[(i*width+col/2+j)*channels + k];
                data[j*channels + k] = (a+b)/sqrt(2);
                data[(col/2+j)*channels + k] = (a-b)/sqrt(2);
            }
        }
        for(j=0;j<col/2;j++)
        {   for(k=0;k<channels;k++)
            {
                image[(i*width + 2*j)*channels + k] = data[j*channels + k];
                image[(i*width + (2*j+1))*channels + k] =
data[(col/2+j)*channels + k];
            }
        }
    }

}

void Haar1DCol(double *image, int row, int col, int width, int channels)
{
    double *data = new double [row*channels];
    int i,j,k;
    for(i=0;i<col;i++)
    {
        for(j=0;j<row/2;j++)
        {
            for(k=0;k<channels;k++)
            {
                double a = image[(2*j*width + i)*channels + k];
```

```
                double b = image[((2*j+1)*width + i)*channels + k];
                data[j*channels + k] = (a+b)/sqrt(2);
                data[(row/2+j)*channels + k] = (a-b)/sqrt(2);
            }
        }
        for(j=0;j<row/2;j++)
        {   for(k=0;k<channels;k++)
            {
                image[(j*width+i)*channels + k] = data[j*channels + k];
                image[((row/2+j)*width+i)*channels + k] =
data[(row/2+j)*channels + k];
            }
        }
    }
}

void UnHaar1DCol(double *image, int row, int col, int width, int channels)
{
    double *data = new double [row*channels];
    int i,j,k;
    for(i=0;i<col;i++)
    {
        for(j=0;j<row/2;j++)
        {
            for(k=0;k<channels;k++)
            {
                double a = image[(j*width+i)*channels + k];
                double b = image[((row/2+j)*width+i)*channels + k];
                data[j*channels + k] = (a+b)/sqrt(2);
                data[(row/2+j)*channels + k] = (a-b)/sqrt(2);
            }
        }
        for(j=0;j<row/2;j++)
        {   for(k=0;k<channels;k++)
            {
                image[(2*j*width + i)*channels + k] = data[j*channels + k];
                image[((2*j+1)*width + i)*channels + k] =
data[(row/2+j)*channels + k];
            }
        }
    }
}

void Haar2D(double *image, int width, int height, int channels)
{
    int level=3, crow = height, ccol = width;

    while(level--)
    {
        Haar1DRow(image, crow, ccol, width, channels);
        Haar1DCol(image, crow, ccol, width, channels);
        ccol /= 2;
        crow /= 2;
    }
}

void Haar_Reconstruct(double *image, int width, int height, int channels)
{
    int level=3, crow = height/4, ccol = width/4;

    while(level--)
    {
```

```
        UnHaar1DCol(image, crow, ccol, width, channels);
        UnHaar1DRow(image, crow, ccol, width, channels);
        ccol *= 2;
        crow *= 2;
    }
}

void Denoise(double *data, int width, int height, int nChannels)
{
    int i,j,k, l;
    //double sigma[3],temp, thr[3];
    double sigx[2000][3], sigy[2000][3], temp, th;

    for(l=1;l<=3;l++)
    {
        /*sigma[0] = sigma[1] = sigma[2] = 0.0;
        for(i=0;i<height/pow(2,l);i++)
            for(j=0;j<width/pow(2,l);j++)
                for(k=0;k<nChannels;k++)
                {
                    sigma[k] += abs(data[(i*width + j)*nChannels + k]);
                }
        sigma[0] /= 0.6745*width*height/pow(2,2*l);
        sigma[1] /= 0.6745*width*height/pow(2,2*l);
        sigma[2] /= 0.6745*width*height/pow(2,2*l);

        thr[0] =
sigma[0]*sqrt(log(width*height/pow(2,2*l))/(width*height/pow(2,2*l)));
        thr[1] =
sigma[1]*sqrt(log(width*height/pow(2,2*l))/(width*height/pow(2,2*l)));
        thr[2] =
sigma[2]*sqrt(log(width*height/pow(2,2*l))/(width*height/pow(2,2*l)));
        */

        for(i=0;i<2000;i++)
            sigx[i][0] = sigx[i][1] = sigx[i][2] = sigy[i][0] = sigy[i][1]
= sigy[i][2] = 0.0;

        for(i=0;i<height/pow(2,l-1);i++)
            for(j=0;j<width/pow(2,l-1);j++)
                for(k=0;k<3;k++)
                {
                    sigx[j][k] += abs(data[(i*width + j)*nChannels + k]);
                    sigy[i][k] += abs(data[(i*width + j)*nChannels + k]);
                }

        for(i=0;i<height/pow(2,l-1);i++)
            for(j=0;j<width/pow(2,l-1);j++)
                for(k=0;k<nChannels;k++)
                {
                    temp = data[(i*width + j)*nChannels + k];

                    double sig = (sigx[j][k]+sigy[i][k])/2;
                    sig /= 0.6745*(width+height)/pow(2,l-1);
                    th = sig*sqrt(log((width+height)/pow(2,l-
1))/((width+height)/pow(2,l-1)));

                    if(abs(temp) < th)
                        data[(i*width + j)*nChannels + k] = 0.0;
                    else if(temp > th)
                        data[(i*width + j)*nChannels + k] -= th;
                    else
```

```
                        data[(i*width + j)*nChannels + k] += th;
                }
    }
}

void Wavelet_Denoise(IplImage *image)
{
    int i,j,k, width, height, nChannels;
    CvScalar temp;
    width = image->width;
    height = image->height;
    nChannels = image->nChannels;
    double *data = new double [width * height * nChannels];

    for(i=0;i<height;i++)
        for(j=0;j<width;j++)
        {
            temp = cvGet2D(image, i, j);
            for(k=0;k<nChannels;k++)
                data[(i*width + j)*nChannels + k] = temp.val[k];
        }

    Haar2D(data, width, height, nChannels);

    Denoise(data, width, height, nChannels);

    Haar_Reconstruct(data, width, height, nChannels);

    for(i=0;i<height;i++)
        for(j=0;j<width;j++)
        {
            for(k=0;k<nChannels;k++)
                temp.val[k] = data[(i*width + j)*nChannels + k];
            cvSet2D(image, i, j, temp);
        }

    delete [] data;
}

int main()
{
    int o_height, o_width;
    char input[40], output[40];
    scanf("%s", input);
    scanf("%s", output);
    scanf("%d%d", &o_width, &o_height);

    IplImage *in_image, *out_image;
    in_image = cvLoadImage(input);

    if(!in_image)
    {
        printf("Error!\n");
        return 0;
    }

    out_image =
cvCreateImage(cvSize(o_width,o_height),IPL_DEPTH_8U,in_image->nChannels);

    bicubic(in_image, out_image, o_width*1.0/in_image->width,
o_height*1.0/in_image->height);
    Wavelet_Denoise(out_image);
```

```
    cvSaveImage(output,out_image);
}
```

Haar (level 3) with Nearest Neighbor:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <cv.h>
#include <string.h>
#include <highgui.h>

//.....................................
Scaling.................................................//

void nearest_neighbour(IplImage * img, IplImage* imgfinal, double
horizontal_scale, double vertical_scale)
{
    int
orig_height,orig_width,orig_step,orig_channels,new_step,new_channels,new_he
ight,new_width,i,j,k;
    CvScalar v;
    orig_height=img->height;
    orig_width=img->width;
    orig_step=img->widthStep;
    new_step=orig_step*horizontal_scale;
    new_channels=orig_channels=img->nChannels;
    uchar *orig_data = (uchar *)img->imageData; //Get the image data in
'data'
    new_height=orig_height*vertical_scale;                    //new image
height
      new_width=orig_width*horizontal_scale;                        //new
image width
      uchar *new_data =(uchar*) imgfinal->imageData;
      for(i=0;i<new_height;i++)
            for(j=0;j<new_width;j++)
                  for(k=0;k<new_channels;k++)
                        new_data[(int)(i*new_step+j*new_channels+k)]=0;
    for(i=0;i<new_height;i++)
    {
        for(j=0;j<new_width;j++)
        {
            double x = (double)j/(double)horizontal_scale;
            double y = (double)i/(double)vertical_scale;
            int temp_width,temp_height;
            if((x - (double)floor(x)) > 0.5)
                temp_width=ceil(x);
            else
                temp_width=floor(x);
            if((y - (double)floor(y)) > 0.5)
                temp_height=ceil(y);
            else
                temp_height=floor(y);
            for(k=0;k<new_channels;k++)
            {
                v.val[k]=orig_data[(int)(temp_height*orig_step +
temp_width*orig_channels+k)];
```

```
//new_data[(int)(i*new_step+j*new_channels+k)]=orig_data[(int)(temp_height*
orig_step + temp_width*orig_channels+k)];
            }
            cvSet2D(imgfinal,i,j,v);
        }
    }
}

//.....................................................................
.....//

void Haar1DRow(double *image, int row, int col, int width, int channels)
{
    double *data = new double [col*channels];
    int i,j,k;
    for(i=0;i<row;i++)
    {
        for(j=0;j<col/2;j++)
        {
            for(k=0;k<channels;k++)
            {
                double a = image[(i*width + 2*j)*channels + k];
                double b = image[(i*width + (2*j+1))*channels + k];
                data[j*channels + k] = (a+b)/sqrt(2);
                data[(col/2+j)*channels + k] = (a-b)/sqrt(2);
            }
        }
        for(j=0;j<col/2;j++)
        {   for(k=0;k<channels;k++)
            {
                image[(i*width+j)*channels + k] = data[j*channels + k];
                image[(i*width+col/2+j)*channels + k] =
data[(col/2+j)*channels + k];
            }
        }
    }

}

void UnHaar1DRow(double *image, int row, int col, int width, int channels)
{
    double *data = new double [col*channels];
    int i,j,k;
    for(i=0;i<row;i++)
    {
        for(j=0;j<col/2;j++)
        {
            for(k=0;k<channels;k++)
            {
                double a = image[(i*width+j)*channels + k];
                double b = image[(i*width+col/2+j)*channels + k];
                data[j*channels + k] = (a+b)/sqrt(2);
                data[(col/2+j)*channels + k] = (a-b)/sqrt(2);
            }
        }
        for(j=0;j<col/2;j++)
        {   for(k=0;k<channels;k++)
            {
                image[(i*width + 2*j)*channels + k] = data[j*channels + k];
                image[(i*width + (2*j+1))*channels + k] =
data[(col/2+j)*channels + k];
```

```
            }
        }
    }

}

void Haar1DCol(double *image, int row, int col, int width, int channels)
{
    double *data = new double [row*channels];
    int i,j,k;
    for(i=0;i<col;i++)
    {
        for(j=0;j<row/2;j++)
        {
            for(k=0;k<channels;k++)
            {
                double a = image[(2*j*width + i)*channels + k];
                double b = image[((2*j+1)*width + i)*channels + k];
                data[j*channels + k] = (a+b)/sqrt(2);
                data[(row/2+j)*channels + k] = (a-b)/sqrt(2);
            }
        }
        for(j=0;j<row/2;j++)
        {   for(k=0;k<channels;k++)
            {
                image[(j*width+i)*channels + k] = data[j*channels + k];
                image[((row/2+j)*width+i)*channels + k] =
data[(row/2+j)*channels + k];
            }
        }
    }
}

void UnHaar1DCol(double *image, int row, int col, int width, int channels)
{
    double *data = new double [row*channels];
    int i,j,k;
    for(i=0;i<col;i++)
    {
        for(j=0;j<row/2;j++)
        {
            for(k=0;k<channels;k++)
            {
                double a = image[(j*width+i)*channels + k];
                double b = image[((row/2+j)*width+i)*channels + k];
                data[j*channels + k] = (a+b)/sqrt(2);
                data[(row/2+j)*channels + k] = (a-b)/sqrt(2);
            }
        }
        for(j=0;j<row/2;j++)
        {   for(k=0;k<channels;k++)
            {
                image[(2*j*width + i)*channels + k] = data[j*channels + k];
                image[((2*j+1)*width + i)*channels + k] =
data[(row/2+j)*channels + k];
            }
        }
    }
}

void Haar2D(double *image, int width, int height, int channels)
{
```

```
    int level=3, crow = height, ccol = width;

    while(level--)
    {
        Haar1DRow(image, crow, ccol, width, channels);
        Haar1DCol(image, crow, ccol, width, channels);
        ccol /= 2;
        crow /= 2;
    }
}

void Haar_Reconstruct(double *image, int width, int height, int channels)
{
    int level=3, crow = height/4, ccol = width/4;

    while(level--)
    {
        UnHaar1DCol(image, crow, ccol, width, channels);
        UnHaar1DRow(image, crow, ccol, width, channels);
        ccol *= 2;
        crow *= 2;
    }
}

void Denoise(double *data, int width, int height, int nChannels)
{
    int i,j,k, l;
    //double sigma[3],temp, thr[3];
    double sigx[2000][3], sigy[2000][3], temp, th;

    for(l=1;l<=3;l++)
    {
        /*sigma[0] = sigma[1] = sigma[2] = 0.0;
        for(i=0;i<height/pow(2,l);i++)
            for(j=0;j<width/pow(2,l);j++)
                for(k=0;k<nChannels;k++)
                {
                    sigma[k] += abs(data[(i*width + j)*nChannels + k]);
                }
        sigma[0] /= 0.6745*width*height/pow(2,2*l);
        sigma[1] /= 0.6745*width*height/pow(2,2*l);
        sigma[2] /= 0.6745*width*height/pow(2,2*l);

        thr[0] =
sigma[0]*sqrt(log(width*height/pow(2,2*l))/(width*height/pow(2,2*l)));
        thr[1] =
sigma[1]*sqrt(log(width*height/pow(2,2*l))/(width*height/pow(2,2*l)));
        thr[2] =
sigma[2]*sqrt(log(width*height/pow(2,2*l))/(width*height/pow(2,2*l)));
        */

        for(i=0;i<2000;i++)
            sigx[i][0] = sigx[i][1] = sigx[i][2] = sigy[i][0] = sigy[i][1]
= sigy[i][2] = 0.0;

        for(i=0;i<height/pow(2,l-1);i++)
            for(j=0;j<width/pow(2,l-1);j++)
                for(k=0;k<3;k++)
                {
                    sigx[j][k] += abs(data[(i*width + j)*nChannels + k]);
                    sigy[i][k] += abs(data[(i*width + j)*nChannels + k]);
                }
```

```
        for(i=0;i<height/pow(2,l-1);i++)
            for(j=0;j<width/pow(2,l-1);j++)
                for(k=0;k<nChannels;k++)
                {
                    temp = data[(i*width + j)*nChannels + k];

                    double sig = (sigx[j][k]+sigy[i][k])/2;
                    sig /= 0.6745*(width+height)/pow(2,l-1);
                    th = sig*sqrt(log((width+height)/pow(2,l-
1))/((width+height)/pow(2,l-1)));

                    if(abs(temp) < th)
                        data[(i*width + j)*nChannels + k] = 0.0;
                    else if(temp > th)
                        data[(i*width + j)*nChannels + k] -= th;
                    else
                        data[(i*width + j)*nChannels + k] += th;
                }
    }
}

void Wavelet_Denoise(IplImage *image)
{
    int i,j,k, width, height, nChannels;
    CvScalar temp;
    width = image->width;
    height = image->height;
    nChannels = image->nChannels;
    double *data = new double [width * height * nChannels];

    for(i=0;i<height;i++)
        for(j=0;j<width;j++)
        {
            temp = cvGet2D(image, i, j);
            for(k=0;k<nChannels;k++)
                data[(i*width + j)*nChannels + k] = temp.val[k];
        }

    Haar2D(data, width, height, nChannels);

    Denoise(data, width, height, nChannels);

    Haar_Reconstruct(data, width, height, nChannels);

    for(i=0;i<height;i++)
        for(j=0;j<width;j++)
        {
            for(k=0;k<nChannels;k++)
                temp.val[k] = data[(i*width + j)*nChannels + k];
            cvSet2D(image, i, j, temp);
        }

    delete [] data;
}

int main()
{
    int o_height, o_width;
    char input[20], output[20];
    scanf("%s%s%d%d", input, output, &o_width, &o_height);
```

```
    IplImage *in_image, *out_image;
    in_image = cvLoadImage(input,-1);

    if(!in_image)
    {
        printf("Error!\n");
        return 0;
    }

    out_image =
cvCreateImage(cvSize(o_width,o_height),IPL_DEPTH_8U,in_image->nChannels);

    nearest_neighbour(in_image, out_image, o_width*1.0/in_image->width,
o_height*1.0/in_image->height);
    Wavelet_Denoise(out_image);

    cvSaveImage(output,out_image);
}
```