5-1-2013

A Digital Image Processing Method for Detecting Pollution in the Atmosphere from Camera Video

Amrita Nikhil Amritphale
University of Nevada, Las Vegas, amritph2@unlv.nevada.edu

A DIGITAL IMAGE PROCESSING METHOD FOR DETECTING POLLUTION IN THE ATMOSPHERE FROM CAMERA VIDEO

by

Amrita Amritphale

Bachelor of Engineering (I.T.)
University of Pune,India
2008

A thesis submitted in partial fulfillment of
the requirements for the

Master of Science in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
May 2013

THE GRADUATE COLLEGE

We recommend the thesis prepared under our supervision by

Amrita Amritphale

entitled

A Digital Image Processing Method for Detecting Pollution in the Atmosphere from Camera Video

be accepted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Department of Computer Science

Evangelos Yfantis, Ph.D., Committee Chair

John Minor, Ph.D., Committee Member

Hal Berghel, Ph.D., Committee Member

Jacimaria Batista, Ph.D., Graduate College Representative

Tom Piechota, Ph.D., Interim Vice President for Research &
Dean of the Graduate College

May 2013

Abstract

In this thesis we examine the use of digital cameras to detect the magnitude of atmospheric pollution present in the atmosphere. Digital cameras are inexpensive and are being used in countless areas, many of which are outdoors and very public. For example, we see digital cameras located at street intersections, city and state parks, and recreation areas. The theory presented in this paper could help agencies to monitor air quality at any of these sites. Our theory is based on how certain molecules and particles that are present in clean air absorb, luminesce, refract, reflect, or scatter the red, green, and blue (RGB) visible light spectrum in a measurable manner. The longer wavelength components (red side) of visible light through the atmosphere are scattered less than the shorter wavelength components (blue side). The blue component is scattered more than the other color components (and, thus, is responsible for our blue sky). The longer wavelength components of visible light are also refracted less than the shorter wavelength components. The presence of certain pollutants and suspended particles in air will cause different levels of absorption, re-emission, refraction, or scattering in the RGB spectrum than that for cleaner air.

Acknowledgements

"I am thankful to my thesis supervisor, Dr. Evangelos Yfantis, Professor, Department of Computer Science UNLV, for his guidance, support, motivation and encouragement throughout the period this work was carried out. His readiness for consultation at all times, his educative comments, his concern and assistance with practical things have been invaluable.

I am grateful to have Dr. John Minor, Dr. Hal Berghel and Dr. Jacimaria Batista in this thesis committee and for their help and guidance. I also thank the other staff members of my department for providing me the necessary opportunities for completion of this thesis."

Amrita Amritphale

University of Nevada, Las Vegas
May 2013

Table of Contents

List of Figures

1 Introduction

The only reason Earth can sustain life is because of its atmosphere, which keeps air readily available for chemical reactions. A number of different gases, including oxygen, make up the Earth's atmosphere in a mixture that keeps plants, animals and people alive. In addition to sustaining life, air plays a role in many other important functions that are best performed when air quality is high.

Air is important to humankind and so is its quality. Air pollutants can cause a variety of health problems - including breathing problems, asthma, reduced lung function and lung damage. Air pollution can also irritate the eyes, nose and throat, and reduce resistance to colds and other illnesses. Air pollution can be especially harmful to the very young, the very old, and those with certain preexisting medical conditions. Air pollution also causes reduce in visibility, damages to buildings and other landmarks, harms trees, lakes and animals. In many countries pollution is the biggest issue.

There are many researches done in environmental analysis and how pollutants affect the atmosphere around us (like [7] [11] [5]). However very few give an easy and inexpensive way by which one can analyse the air quality we breath. In this chapter we will briefly look into the basic concepts used to prove the theory which is highly based on digital image and environmental science.

1.1 Digital Image

The basic way of representing a digital colored image in a computer's memory is a bitmap. A bitmap is constituted of rows of pixels. Each pixel has a particular value which determines its appearing color. This value is qualified by three numbers giving the decomposition of the color in the three primary colors Red, Green and Blue. Any color visible to human eye can be represented this way. The decomposition of a color in the three primary colors is quantified by a number between 0 and 255. For example, white will be coded as R = 255, G = 255, B = 255; black will be known as (R,G,B) = (0,0,0); and say, bright pink will be : (255,0,255). In other words, an image is an enormous two

dimensional array of color values of pixels, each of which is coded on 3 bytes, representing the three primary colors. This allows the image to contain a total of 256x256x256 = 16.8 million different colors. This technique is also known as RGB encoding, and is specifically adapted to human vision. With cameras or other measuring instruments we are capable of seeing thousands of other colors, in which cases the RGB encoding is inappropriate. The range of 0-255 was agreed for two good reasons: The first is that the human eye is not sensible enough to make the difference between more than 256 levels of intensity (1/256 = 0.39%) for a color. That is to say, an image presented to a human observer will not be improved by using more than 256 levels of gray (256 shades of gray between black and white). Therefore 256 seems enough quality. The second reason for the value of 255 is obviously that it is convenient for computer storage. Indeed on a byte, which is the computers memory unit, can be coded up to 256 values.

In a bitmap, colors are coded on three bytes representing their decomposition on the three primary colors. We can interpret colors as vectors in a three dimension space where each axis stands for one of the primary colors as shown in figure 1.1.

1.2 Wavelet Transformation

The wavelet transformation has emerged as a cutting edge technology, within the field of signal and image analysis. As mentioned in [16] and [17], Wavelets are a mathematical tool for hierarchically decomposing functions. Though rooted in approximation theory, signal processing, and physics, wavelets have also recently been applied to many problems in computer graphics. Mathematical transformations are applied to an image to obtain further information from that image that is not readily available in the original image. The most distinguished information is hidden in the frequency content of an image. The frequency SPECTRUM of a image is basically the frequency components (spectral components) of that image. The frequency spectrum of an image shows what frequencies exist in an image.

Intuitively, we all know that the frequency is something to do with the change in rate of something. If something (a mathematical or physical variable, would be the technically correct term) changes rapidly, we say that it is of high frequency, where as if this variable does not change rapidly, i.e., it changes smoothly, we say that it is of low frequency. If this variable does not change at all, then we say it has zero frequency, or no frequency. For example the publication frequency of a daily newspaper is higher than that of a monthly magazine (it is published more frequently). How do we find the frequency content of an image? One of the ways is Haar wavelet transform , which is

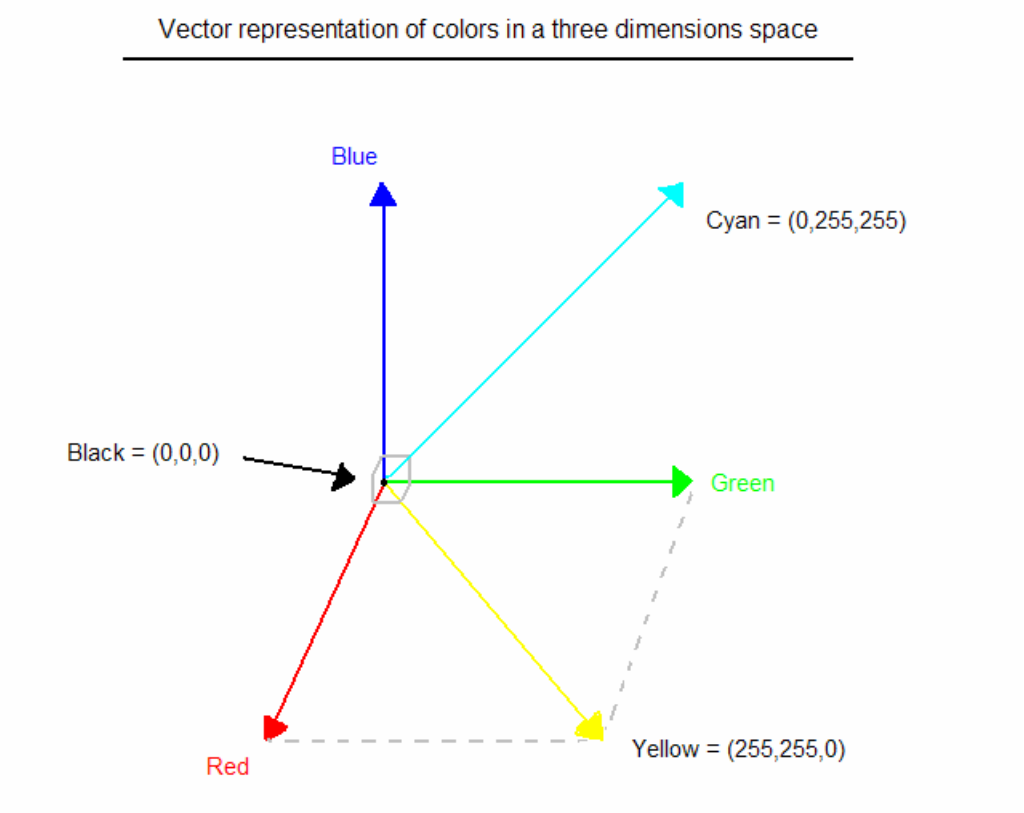

Figure 1.1: Vector representation of colors [20]

explained in Chapter 3 in more detail.

1.3 The Atmosphere

The atmosphere is the mixture of gas molecules and other materials surrounding the earth. It is made mostly of the gases nitrogen (78%), and oxygen (21%). Argon gas and water (in the form of vapor, droplets and ice crystals) are the next most common things. There are also small amounts of other gases, plus many small solid particles, like dust, soot and ashes, pollen, and salt from ocean. The composition of the atmosphere varies, depending on the location, weather, and many other things. There may be more water in the air after a rainstorm, or near the ocean. Volcanoes can put large amounts of dust particles into the atmosphere. Pollution can add different gases or dust and soot. The atmosphere is densest (thickest) at the bottom, near the Earth. It gradually thins out as you go higher and higher up. There is no sharp break between the atmosphere and space.

1.4 Light Waves

Light is a kind of energy that radiates, or travels, in waves. Many different forms of energy travel in waves. For example, sound is a wave of vibrating air. Light is a wave of vibrating electric and magnetic fields. It is one small part of a larger range of vibrating electromagnetic fields. This range is called the electromagnetic spectrum. Electromagnetic waves travel through space at 299,792 km/sec (186,282 miles/sec). This is called the speed of light.

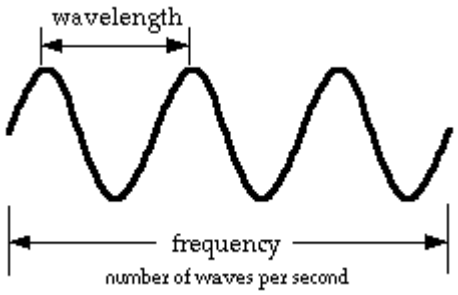

Figure 1.2: Light wave. This picture is made by Amrita N. Amritphale.

The energy of the radiation depends on its wavelength and frequency. Wavelength is the distance between the tops (crests) of the waves. Frequency is the number of waves per second. Longer the wavelength of the light, lower the frequency, and the less energy it contains.

Reflection is one of the methods to control light. Another one is called refraction. When light that is traveling through one substance, such as air, hits another substance, such as the glass of a window, this juncture is called an interface. Refraction occurs when light bends at such an interface.

1.4.1 Light in the Air

Light travels through space in a straight line as long as nothing disturbs it. As light moves through the atmosphere, it continues to go straight until it bumps into a bit of dust or a gas molecule. Then what happens to the light depends on its wave length and the size of the thing it hits. The interaction of light with matter can result in one of three wave behaviors: absorption, transmission, and reflection. Dust particles and water droplets are much larger than the wavelength of visible light. When light hits these large particles, it gets reflected, or bounced off, in different directions. The different colors of light are all reflected by the particle in the same way. The reflected light appears white because it still contains all of the same colors.

If light hits the molecules present in the atmosphere which have smaller wavelength than the visible light it acts differently. Some of the light gets absorbed by such molecules. These molecules then radiate the light in a different direction. The different colors of light are affected differently. Higher frequency colors in light are absorbed more often than the lower frequencies.

1.4.2 Scattering of light

The atmosphere is a gaseous sea that contains a variety of types of particles; the two most common types of matter present in the atmosphere are gaseous nitrogen and oxygen. These particles are most effective in scattering the higher frequency and shorter wavelength portions of the visible light spectrum. This scattering process involves the absorption of a light wave by an atom followed by re emission of a light wave in a variety of directions. The amount of multi- directional scattering that occurs is dependent upon the frequency of the light. So as the white light (ROYGBIV) from the sun passes through the atmosphere, the high frequencies (Blue) become scattered by atmospheric particles while the lower frequencies (Red, Green) are most likely to pass through the atmosphere without a significant alteration in their direction.

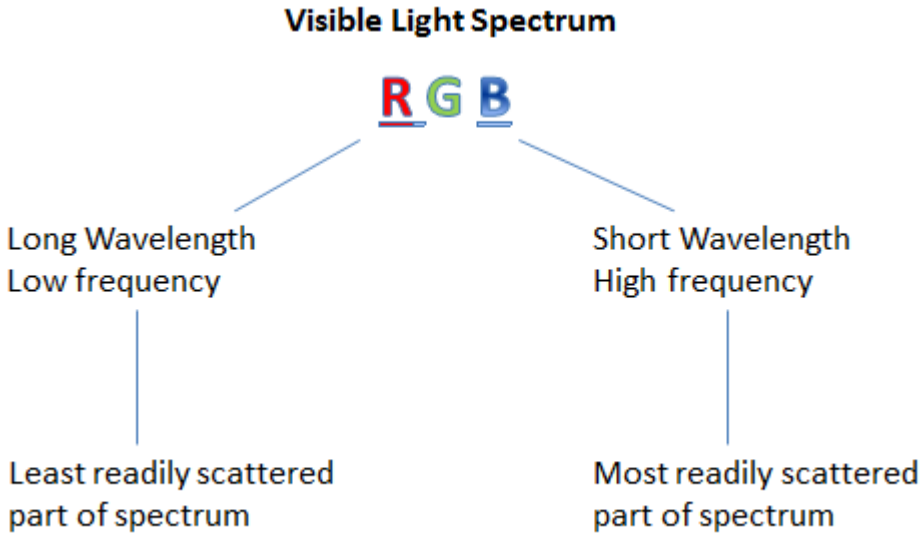

Figure 1.3: Low frequency - High frequency portions in Visible Light Spectrum. This picture is made by Amrita N. Amritphale.

In the figure 1.4, sunlight comes into the atmosphere and can be scattered in any direction as it

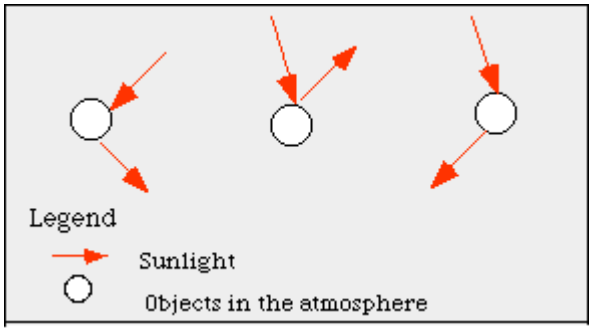

Figure 1.4: Scattering of Sunlight [21]

passes through a medium. This diffuses the light spreading it out in all directions so it is not just a single, straight beam. There are three different types of scattering: Rayleigh scattering, Mie scattering, and non-selective scattering.

Rayleigh scattering mainly consists of scattering from atmospheric gases. Gas molecules are smaller than the wavelength of visible light. If light bumps into them, it acts differently. When light hits a gas molecule, some of it may get absorbed. After awhile, the molecule radiates (releases, or gives off) the light in a different direction. The color that is radiated is the same color that was absorbed. The different colors of light are affected differently. All of the colors can be absorbed. But the higher frequencies (blues) are absorbed more often than the lower frequencies (reds). Because of Rayleigh scattering, the sky appears blue (as in the figure 1.5). This is because blue light is scattered around four times as much as red light, and UV light is scattered about 16 times as much as red light.

Figure 1.5: Blue sky [21]

Mie scattering is caused by pollen, dust, smoke, water droplets, and other particles in the lower portion of the atmosphere. It occurs when the particles causing the scattering are larger than the

wavelengths of radiation in contact with them. Mie scattering is responsible for the white appearance of the clouds (as seen in figure 1.6). The effects are also wavelength dependent.

Figure 1.6: White cloud [21]

The last type of scattering is non-selective scattering. It occurs in the lower portion of the atmosphere when the particles are much larger than the incident radiation. This type of scattering is not wavelength dependent and is the primary cause of haze. Figure 1.7 shows Grand Canyon haze.

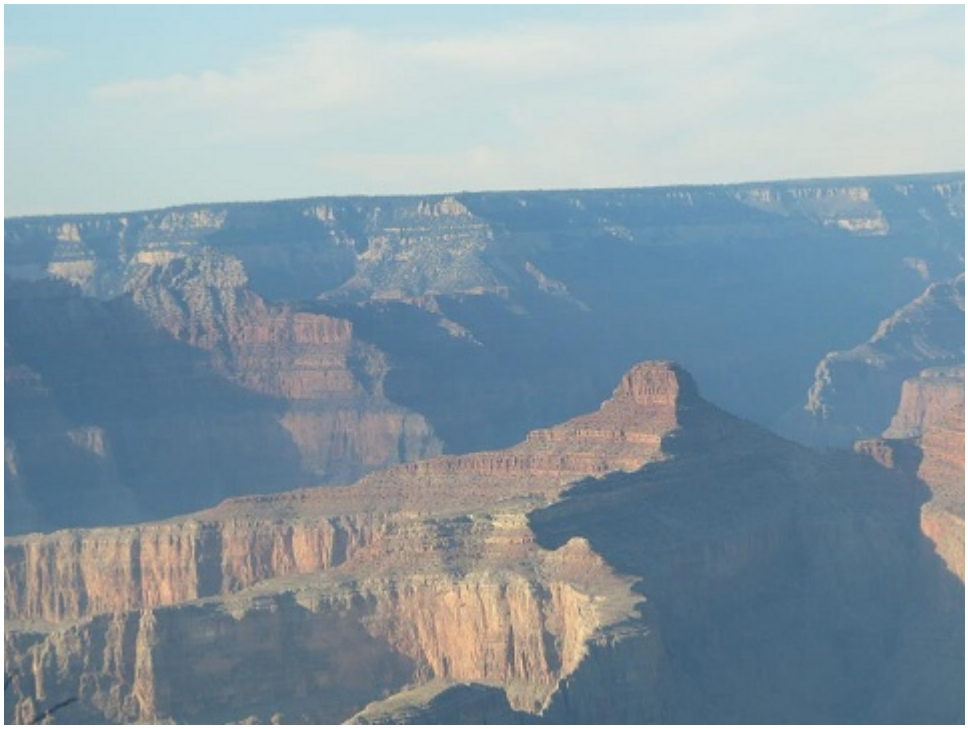

Figure 1.7: Haze in Grand Canyon south rim. This picture is taken by Amrita N. Amritphale.

2 Background

Historically, Visibility has been defined as the greatest distance at which an observer can just see a black object viewed against the horizon sky. An object is usually referred to as at threshold contrast when the difference between the brightness of the sky and brightness of the object is reduced to such a degree that an observer can just barely see the object. Nevertheless, visibility is more than being able to see a black object at a distance for which the contrast reaches a threshold value. Visibility is more closely associated with conditions that allow appreciation of the inherent beauty of landscape features. It is important to recognize and appreciate the form, contrast detail, and color of near and distant features. Because visibility includes a psychophysical process and concurrent value judgment of visual impacts, as well as the physical interaction of the light with particles in the atmosphere, it is of interest to:

- understand the psychological process involved in viewing a scenic resource,
- specify and understand the value that an observer places on visibility and
- be able to establish a link between the physical and psychological process.

Figure 2.1: Seward bay area. Notice the brightly colored river.This picture is taken by Amrita N. Amritphale.

Figure 2.2: Captured at Denali national park. Observe the sunlight effects added to the background mountains and sky. This picture is taken by Amrita N. Amritphale.

Introduction of particulate matter and certain gases into the atmosphere interfaces with the ability of an observer to see landscape features. Monitoring, modeling and controlling sources of visibility-reducing particulate matter and gases depend on scientific and technical understanding of how these pollutants:

- interact with light
- transform from a gas into particles that impair visibility and,
- are dispersed across land masses and into canyons and valleys.

One of our principal contacts with the world around us is through light. Not only are we personally dependent on the light to carry visual information, but also much of what we know about stars and the solar system is derived from light waves registering on our eyes and on optical instruments. Light can be thought of as waves and to a certain extent they are analogous to water and sound waves. Waves of all kinds, including light waves, carry energy. Electromagnetic energy is unique in that energy is carried in small, discrete parcels called photons. Schematic representations of a blue, red and green photons are shown in figure (2.3). Blue, green and red photons have wavelengths of around 0.45, 0.55 and 0.65 microns, respectively. The color properties of light depend on its behaviour both as waves and as particles.

Colors created from white light by passing it through a prism, are a result of the wave like nature

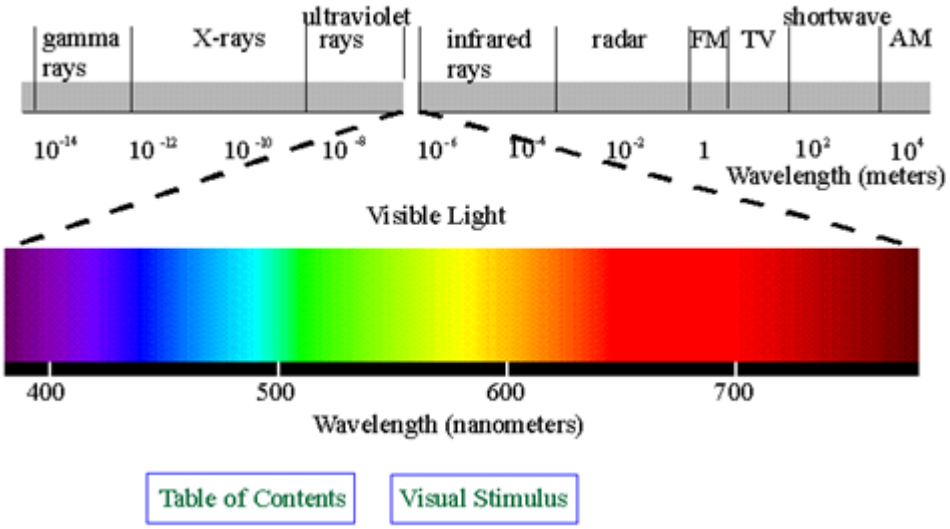

Figure 2.3: Electromagnetic Spectrum [19]

of light. A prism separates the colors of light by bending (refracting) each color to a different degree. Colors in a rainbow are the result of water droplets acting like small prisms, dispersed through the atmosphere. Each water droplet refracts light into the component colors of the visible spectrum. More commonly, the colors of light are separated in other ways. When light strikes an object, certain color of photons are captured by molecules in that object. Different types of molecules capture photons of different colors. The only colors we see are those photons that the surface reflects. For instance chlorophyll in leaves captures red and blue light and allows green photons to bounce back , thus providing green appearance of leaves. Nitrogen dioxide, gas emitted into the atmosphere by combustion sources, captures blue photons. Consequently, nitrogen dioxide gas tends to look reddish brown. Figure 2.4 shows an apple reflects mostly red light while absorbing all others, so the apple, to an eye-brain system, appears to be red. It is important to understand the significance of the light that is scattered in the sight path toward the observer. The amount of light scattered by atmosphere and particles between the object and observer can be so bright and dominant that the light reflected by landscape features becomes insignificant. This is somewhat analogous to viewing a candle in a brightly lit room and in a room that would otherwise be in total darkness. In the first case, the candle can hardly be seen, while in other it becomes the dominant feature in the room.

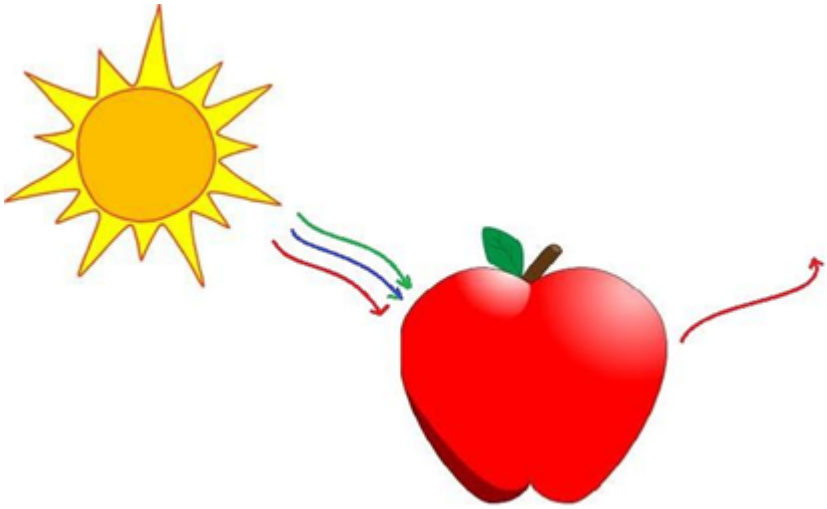

Figure 2.4: Apple appear red.This picture is made by Amrita N. Amritphale

2.1 Interaction of Light and Particles

A photon (light particle) is said to be scattered when it is received by a particle and re-radiated at the same wavelength in any direction. Visibility degradation results from light scattering and absorption by atmospheric particles and gases that are nearly the same size as the wavelength of the light. Particles somewhat larger than the wavelength of light can scatter light as a result of Diffraction, Refraction and Phase shift.

The efficiency with which a particle can scatter light and the direction in which the incident light is redistributed are dependent on all three of these effects and absorption effect. If the particles are small, the amount of light scattered in the forward and backward directions are nearly the same. As the particle size increases in size, more light tends to scatter in the forward direction. Very small particles and molecules are very insufficient at scattering light. As particle increases in size, it becomes a more efficient light scatter until, at a size that is close to the wavelength of the incident light, it can scatter more light than a particle five times its size. It is this scattering phenomenon that is responsible for the colors of hazes in the sky. The sky is blue because blue photons, with their shorter wavelengths, are nearer the size of molecules that make up the atmosphere than are their green and red counterparts. Thus blue photons are scattered more efficiently by air molecules than red photons and as a consequence, the sky looks blue.

When the red, blue and green photons of white light strike small particles, only blue photons are

scattered because scattering efficiency is greatest when the size relationship of photon wavelength to particle is close to 1:1. The red and green photons pass on through the particles. To an observer standing to the side of particle concentration, the haze would appear to be blue. When the particles are about the same size as incoming radiation, all photons are scattered equally and the haze would appear to be gray or white.

The camera can be an effective tool in capturing the visual impact that pollutants have on a visual resource. The following pictures show polluted blue sky Figure 2.6 and unpolluted sky Figure 2.5.

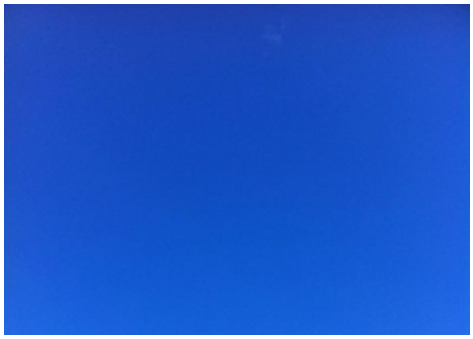

Figure 2.5: Unpolluted sky.This picture is taken by Amrita N. Amritphale

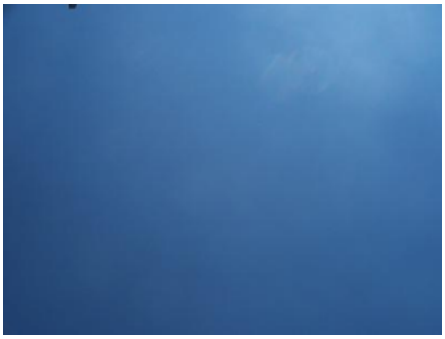

Figure 2.6: Polluted sky.This picture is taken by Amrita N. Amritphale.

Carbon absorbs all wavelengths of light and scatters very little. Thus the scene will always tend to be darkened. NO2 on other hand absorbs blue photons and thus the result will be dark brown.

2.2 Transport and Transformation of Atmospheric particulates and gases affecting visibility

Understanding how air moves across the oceans and land masses is key to understanding how pollutants are transported and transformed as they move from their source to locations where they impair visibility.

Meteorological factors, such as wind, cloud cover, rain and temperature are interesting in that they are affected by pollution and they in turn affect pollution. The rate at which pollutants are converted to other pollutants- sulfur dioxide gas to sulfate particles or nitrogen oxides and hydrocarbons to ozone- is determined by the availability of sunlight and presence or absence of clouds.

Heating of the earths surface and the resultant vertical temperature profile determine whether pollutants are dispersed or mixed vertically. A second and important process for mixing of the earths

atmosphere is wind and the resultant mechanical mixing when wind passes over surface structures such as tall buildings or mountainous terrain. Some of the cleanest air is found on the windiest day. Pollutants emitted that are well mixed will appear as a uniform haze. This is shown in figure 2.7. When pollutants are emitted into stable atmosphere, usually one of the two things will happen, depending on whether there is surface wind or not. If a wind is present, the emitted pollutants usually form a plume. If there is no surface wind or if pollutants are emitted into a stagnant air mass over a period of days, a condition as shown in figure 2.8 can occur. A layer of haze forms near the ground and continues to build as long as the stagnation condition persists. Layered hazes are usually associated with emissions that are local in nature as opposed to pollutants that are transported over hundreds of kilometers.

Figure 2.7: Uniform Haze [18]

Figure 2.8: Stagnant air mass over a period of days

2.3 Atmospheric Chemistry

Particulates and gases in the atmosphere can originate from natural or man-made sources. The ability to see and appreciate a visual resource is limited, in the unpolluted atmosphere, by light scattering molecules that make up the atmosphere. These molecules are primarily nitrogen and oxygen along with some trace gases such as argon and hydrogen. Other forms of natural aerosol that limit our ability to see are condensed water vapor, wind-blown dust and organic aerosols such as pollen and smoke from wildfires.

Aerosols, whether they are manmade or natural, are said to be primary or secondary in nature. Primary refers to gases or particles emitted from a source directly, while secondary refers to airborne dispersions of gases and particles formed by atmospheric reactions of precursor or primary emissions.

Near a source (within 0-100 km), such as an urban center, power plant or other industrial facilities, haze is usually a mixture of gases and secondary and primary aerosols. After these pollutants have

been transported hundreds of kilometers, gaseous emissions have either deposited to aquatic or terrestrial surfaces or converted to secondary aerosols. Thus in remote areas of the United States, man-made components of haze are usually composed of secondary particles. However in some parts of the forested United States, fire emissions can contribute significantly to primary carbon particles.

2.4 Measurement of Scattering and Extinction

The scattering coefficient is a measure of the ability of particles to scatter photons out of the beam of light, while the absorption coefficient is a measure of how many photons are absorbed. Each parameter is expressed as a number proportional to the amount of photons scattered or absorbed per distance. The sum of scattering and absorption is referred to as extinction or attenuation.

2.5 Particle concentration and visibility trends

There are and have been a number of particle and visibility monitoring programs implemented in the United States, most notably the Interagency Monitoring of Protected Visual Environments (IMPROVE) and the National Weather Service (NWS) program. The IMPROVE program, by design, has its focus on non-urban environments, while the NWS program was carried out at airports across the United States.

The outstanding feature of all four geographic areas is a similar seasonal trend in the total fine mass concentration represented as the sum of the aerosol species, in the concentration represented as the sum of aerosol species and in the concentration of each individual species. The highest fine mass concentration occurs in summer while winter has the lowest. Concentrations of sulfates and organics have similar trends in all four areas.

Nitrates tend to be higher in winter and spring than in summer and fall. Trends in soil are variable, while element carbon shows little variation from season to season. Sulfates are by far the largest contributor to fine mass in the eastern United States, while in the Northwest organics contribute most to fine mass. Nitrates edge out organics and sulfates in southern California, while in the southwest sulfates, organics and soil all contribute equally to fine mass.

2.6 Identification of sources Contributing to Visibility Impairment

Goal of identifying the particles affecting visibility is to reduce their concentration and thereby improve the seeing of land-scape features. It becomes necessary to identify the sources emitting

the precursor pollutants that form visibility reducing particles. There are generally two ways to go about this.

The model must predict transport of gases such as sulfur dioxide, nitrogen dioxide and reactive hydrocarbons, convert them into secondary particles, deposit them as wet and dry deposition and form estimates of size and composition of concentration that affect visibility. Since the model will only be as accurate as the emission estimates that are input into the model, it is crucial to develop an accurate emission inventory. These types of models are referred to as deterministic or first principle source-oriented models. They tend to capture only broad-scale temporal and spatial characteristic of haze formation and are computer intensive.

Diagnostic receptor-oriented models have evolved as a clear alternative to source-oriented dispersion models. Receptor models start with the measurement of specific features in order to develop estimates of aerosol contributions of specific source types and/or source locations. In a most general sense, geographic regions with high emissions will have high particle loadings. For instance, high sulfur dioxide emissions will be associated with high ambient sulfate concentrations and sulfate deposition, and conversely low emissions will correlate with low ambient concentrations. In North America, about 27% of emitter sulfur dioxide is dry deposited, 34% wet deposited and 39% remains in the atmosphere and is eventually exported from continent primarily to the Atlantic ocean. The single largest source of sulfur dioxide is the electric utility industry (coal-fired power), while sources of nitrogen oxides are nearly evenly split between utility industry and transportation. Most hydrocarbons gases are emitted by transportation sources.

2.6.1 Human Perception of Visual Air Quality

A major challenge in establishing visibility values is to develop ways of quantitatively measuring visibility impairment as perceived by the human eye. Quantification of visual impairment of scenic resource requires two crucial components:

- The establishment of the level of air pollution that is just noticeable.
- A determination of the functional relationship between air pollution and perceived visual air quality.

The first goal is important when it is necessary to quantitatively specify visible pollution under a given atmospheric condition. The second object is important when trying to access the societal value of clean air, whether it be social, psychological or economical. The first step in assessing values is

to understand the relationship between perceived changes in visual air quality and an appropriate physical parameter, such as vista contrast or atmospheric extinction. For example, if a visitor is willing to pay $5 for a given decrease in atmospheric extinction (air pollution) at the Grand Canyon, but is unwilling to pay that same amount for a similar decrease at some other park, is it because a) the person values scenic resource differently at two parks or b) the perceived change in visual air quality is different at the two parks? That is , at one national park a given decrease in extinction can readily be seen, while at the other park same decrease might go unnoticed.

3 Exploratory Analysis and Classification Vector Formulation

The sky is blue because blue photons, with their shorter wavelengths, are closer to the size of molecules that make up the atmosphere than their green and red counterparts. Thus blue photons are scattered more efficiently by air molecules than red and green photons and as a consequence, the sky looks blue. Alternatively, the atmosphere is the mixture of gas molecules and other materials surrounding the earth. It is made mostly of the gases nitrogen (78%), and oxygen (21%). Thus when light passes through atmosphere red and green components gets filtered out. However the blue component does not get filtered out. Due to its shorter wavelength it is then absorbed by the gas molecules. The absorbed blue light is then radiated in different directions. It gets scattered all around the sky. Hence sky appears blue.

3.1 Digital Image Histogram

Digital image pixel tonality (darkness, lightness) for 24 bit RGB color is expressed as a number between 0 and 255. 0 equals pure black and 255 equals pure white. The mid point at about 127 would be the equivalent of middle gray in density. An image histogram is a type of histogram that acts as a graphical representation of the tonal distribution in a digital image. It plots the number of pixels for each tonal value. The horizontal axis of the graph represents the tonal variations, while the vertical axis represents the number of pixels in that particular tone. By looking at the histogram for a specific image, entire tonal distribution can be analyzed at a glance.

To analyze patterns among different blue sky images, red, green , blue or mixed histogram can be drawn as polygons. To draw the polygon, first get all red, green and blue pixels in an image. As any image lies in the visibility spectrum, every pixel has a value between 0 to 255. Determine how many times each pixel value appears in its respective red, green and blue channels. Store these

values in an array (PixelCount[ ]). Get the value which appears the maximum number of times for each color. This value represents the height of a polygon (iMax). To display the polygons of each color, create initial and final points in the left-bottom and right-bottom of the rectangle in which it has to be displayed. Compute the scaling factor so that it will fit in the given rectangle. Scaling factors can be computed for height and width. For scaling height (fheight), the formula should be rectangle's height divided by height of polygon (iMax).For scaling width (fwidth), the formula should be rectangle's width divided by 256. These scaling factors can be used to find the X and Y coordinates of a polygon. As pixels lie in between 0 and 255, there will be a total of 256 points.

The x-coordinate of a point can be computed as-

(i * fwidth) + Rectangle's x-coordinate of the left edge , where i = 0 to 255

And Y-coordinate of a point can be computed as-

Rectangle's bottom y-coordinate (Height) - (fheight * PixelCount[i]) , where i = 0 to 255

After generating scaled points based on histogram data, polygons can be drawn for each red, green and blue channel or for all channels at the same time. Figure 3.1 shows the histogram of figure 2.1 in chapter 2. Notice how the intensity distribution for each color channel varies drastically in regions of nearly pure color. Histogram in figure 3.1 is generated by the program given in appendix B.

3.2 Haar Wavelet

An image is represented as a two-dimensional array of coefficients, each coefficient representing the brightness level in that point. When looking from a higher perspective, we can't differentiate between coefficients as more important ones, and lesser important ones. But thinking more intuitively, we can. Most natural images have smooth color variations, with the fine details being represented as sharp edges in between the smooth variations. Technically, the smooth variations in color can be termed as low frequency variations and the sharp variations as high frequency variations.

The low frequency components (smooth variations) constitute the base of an image, and the high frequency components (the edges which give the detail) add upon them to refine the image, thereby giving a detailed image. Hence, in this case we will focus more on the detailed image than the smooth variations. Separating the smooth variations and details of the image can be done in many ways. One such way is the decomposition of the image using a Haar wavelet transform.

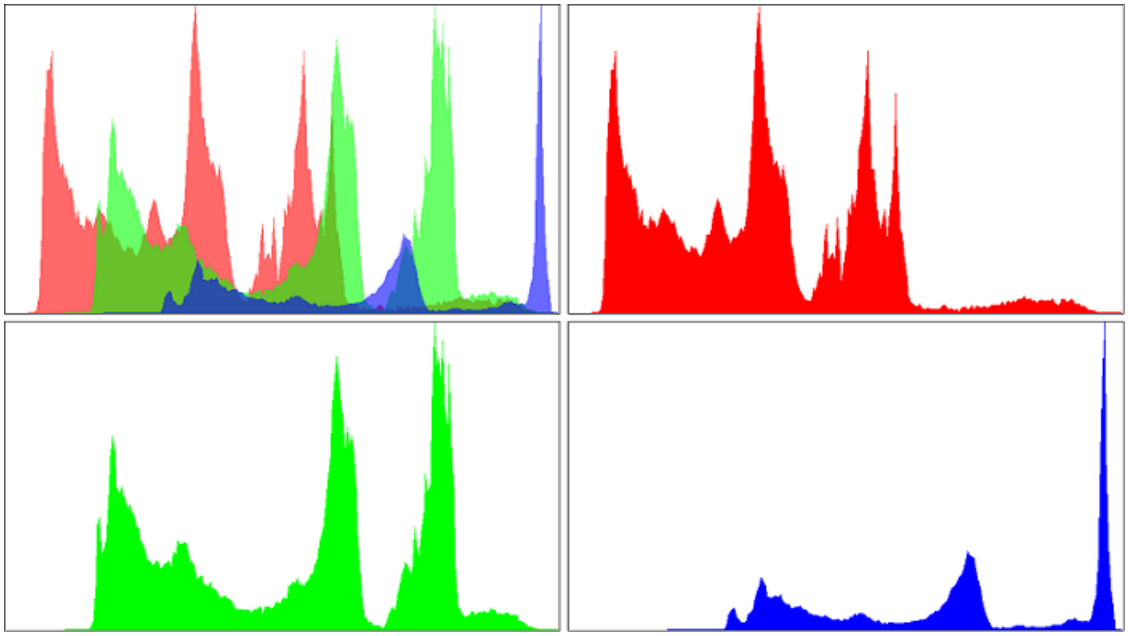

Figure 3.1: Histogram of Seward Bay(figure 2.1) image

A Haar wavelet transform can be used to get the high frequency and low frequency components of red, green or blue pixels of an image. The procedure goes like this. A low pass filter and a high pass filter are chosen, such that they exactly halve the frequency range between themselves. This filter pair is called the Analysis Filter pair. First, the low pass filter is applied to each row of data, thereby getting the low frequency components of the row. But since the lpf (low pass filter) is a half band filter, the output data contains frequencies only in the first half of the original frequency range. So, by Shannon's Sampling Theorem, they can be subsampled by two, so that the output data now contains only half the original number of samples. Now, the high pass filter is applied to the same row of data, and similarly the high pass components are separated, and placed by the side of the low pass components. This procedure is done for all rows.

Next, the filtering is done for each column of the intermediate data. The resulting two-dimensional array of coefficients contains four bands of data, each labelled as LL (low-low), HL (high-low), LH (low-high) and HH (high-high) with most of the energy concentrating in LL subband. The LL band can be decomposed once again in the same manner, thereby producing even more subbands. This can be done upto any level, thereby resulting in a pyramidal decomposition as shown in figure 3.2.

Row transformation is called horizontal pass and column transformation is called vertical pass. Low

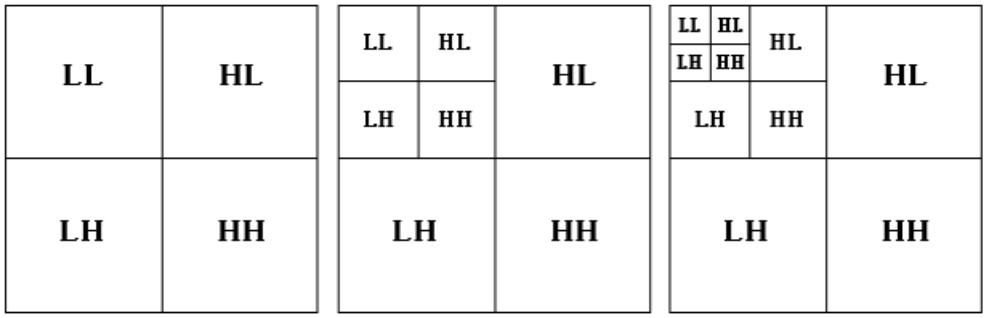

Figure 3.2: Haar wavelet decomposition of an image

components can be computed by adding values of adjacent pixels. High components can be computed by subtracting values of adjacent pixels. Figure 3.3 below shows working of Horizontal and vertical pass and how image pixels (either red, green and blue) are processed to get high components and low component repetitively.

Comparison between two sky images can also be done. The result of it looks like figure 3.5 . Observe that the high-low component of red pixels of an polluted sky image (Image2) has more energy than high-low component of red pixels of a unpolluted sky image. This is because pollutants provide discontinuity. Color channels amplitude are higher for polluted image. Thus high-low frequency is not as similar as low-high and high-high in the case of a polluted image.

3.2.1 Implementation Details

A Haar wavelet can be implemented with the help of a tree structure. The root node will be represented as the original image. In first pass it will get divided into two parts. The right node will represent the high frequency component and the left node will represent the low frequency component of the parent node. The same structure can be added for every new parent node. After consecutive horizontal and vertical passes, these passes again can be applied to only low-low frequency components of an image to see the energy distribution among processed image pixels. For example, similar energy in High-Low and Low-Low-High-Low can be seen. See the figures 3.5 and 3.6.

For displaying low and high frequency components, a pixel value has to be converted to a visible pixel value. To convert it to a visible pixel, first find min_pixel and max_pixel values. If min_pixel value is less than 0 then add the absolute value of min_pixel value to every pixel value of a processed

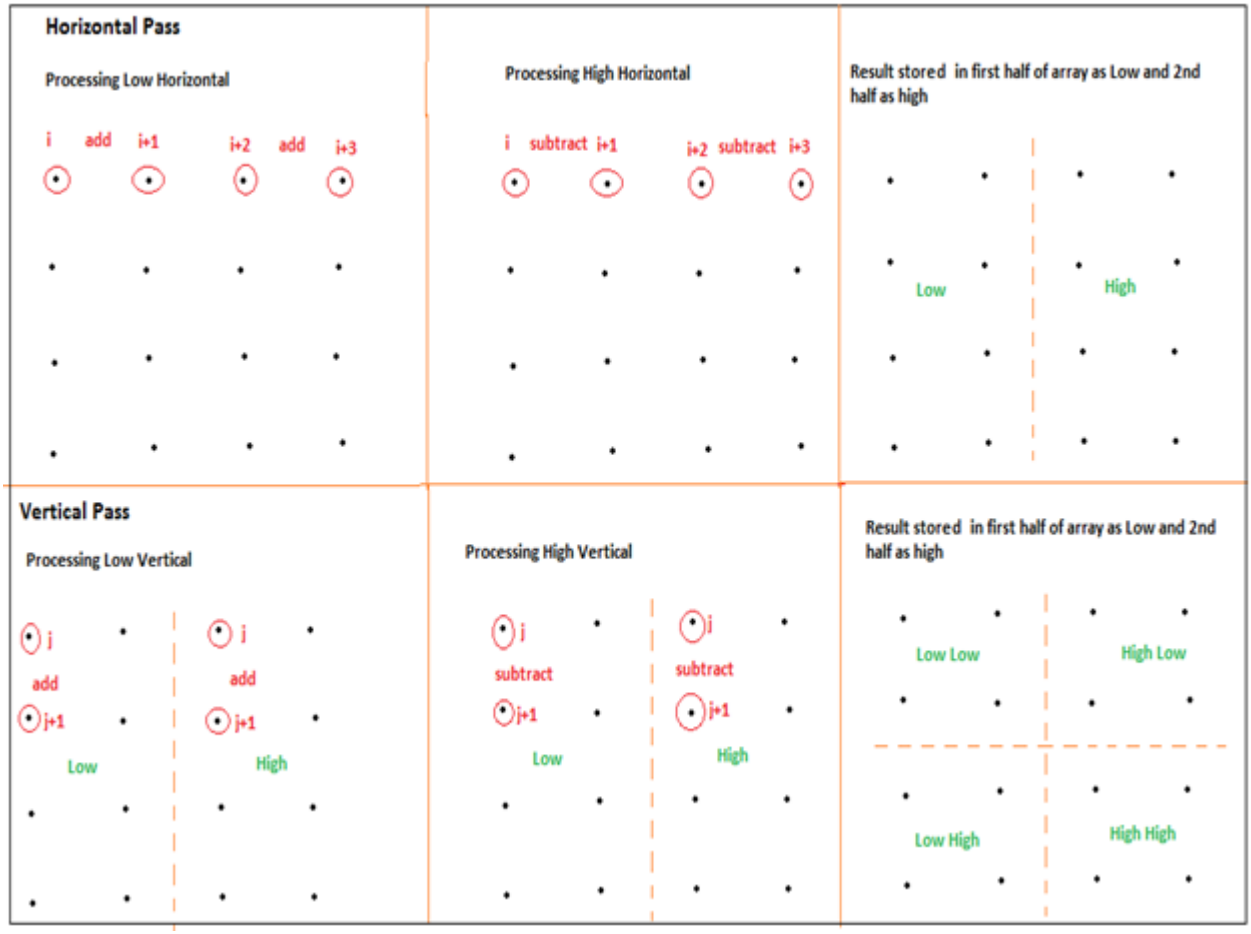

Figure 3.3: Working of Horizontal and Vertical pass to get high and low component of an image

component.

The equation should be-

$$visiblepixel[i][j] = pixels[i][j] + abs_{min} \tag{3.1}$$

Where,

$visiblepixel[i][j]$ is a two dimensional pixel array which has the converted value to display the processed component , where i and j are the width and height of the processed component.

$pixels[i][j]$ is a two dimensional array which has to be converted, where i and j are the width and height of the processed component.

abs_{min} is absolute value of minimum pixel value.

If max_pixel value is greater than 255 then apply the below equation (3.2) to convert it into a visible

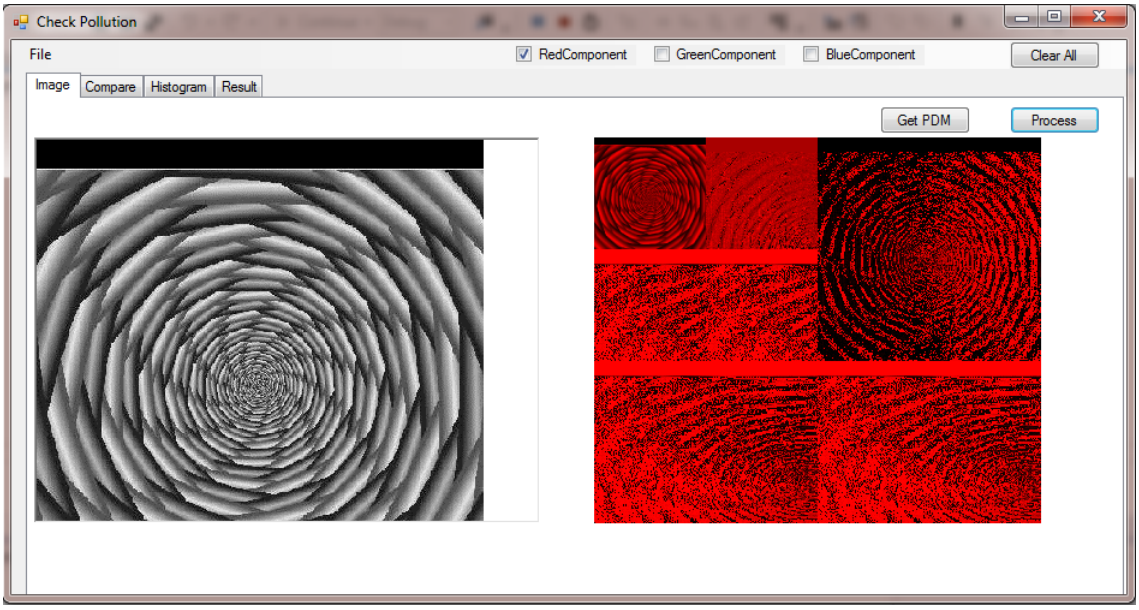

Figure 3.4: Result of Haar wavelet applied to an image. In this example, the red component of an image is shown after horizontal and vertical passes

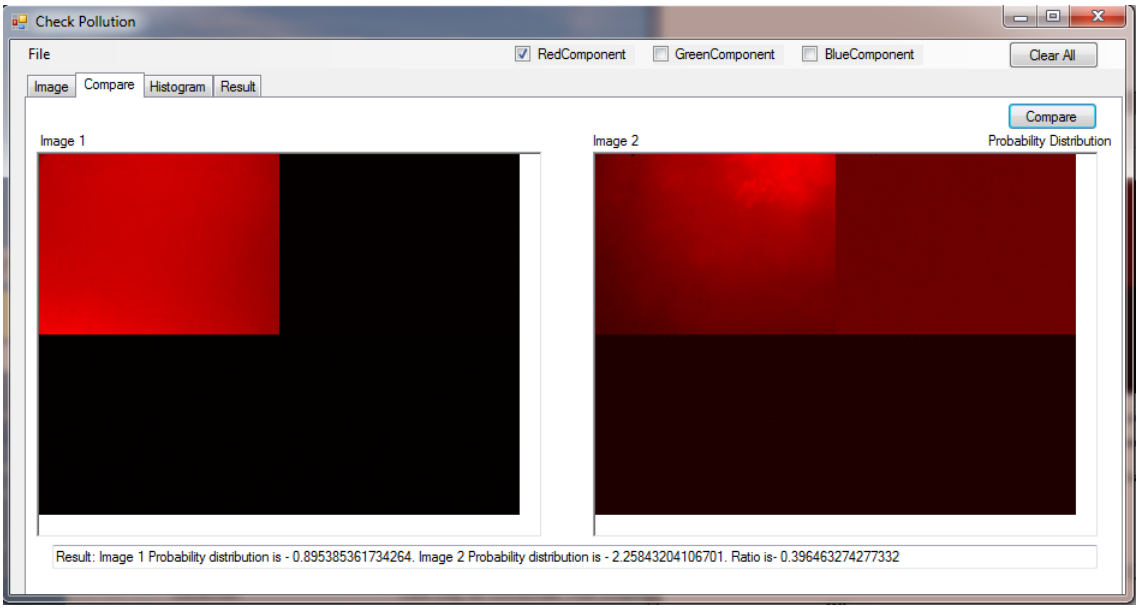

Figure 3.5: Haar wavelet applied to Image1 as unpolluted and Image2 as polluted

pixel value-

$$visiblepixel[i][j] = \frac{pixels[i][j] \times MAXRGB}{n_{max}} \tag{3.2}$$

Where,

$visiblepixel[i][j]$ is a two dimensional pixel array which has the converted value to display the processed component , where i and j are the width and height of the processed component.

$pixels[i][j]$ is a two dimensional array which has to be converted, where i and j are the width and height of the processed component.

$MAXRGB$ is 255 value.

n_{max} is maximum value in a pixels array, which has processed component pixel values.

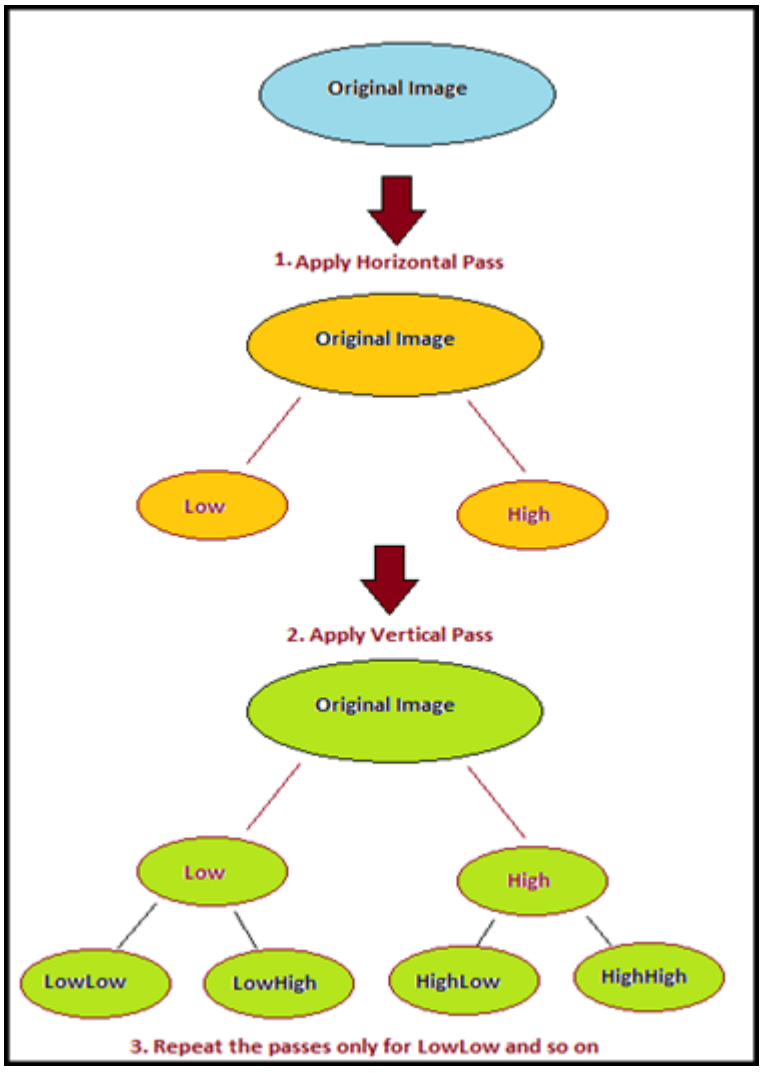

Figure 3.6: Implementing Haar Wavelet with the help of binary tree

3.3 Variance-Covariance

3.3.1 Variance/ Probability Distribution

Variance is a measure of the variability or diversity in a set of data. Mathematically, it is the average squared deviation from the mean score. We use the following formula to compute variance. Thus the larger the diversity the larger will be the variance in RGB color channel in an image.

$$\hat{\sigma}^2_{R'} = \frac{\sum_{c=0}^{C-1}\sum_{r=0}^{R-1}(R'_{cr} - \overline{R'})^2}{RC-1} \tag{3.3}$$

$$\hat{\sigma}^2_{G} = \frac{\sum_{c=0}^{C-1}\sum_{r=0}^{R-1}(G_{cr} - \overline{G})^2}{RC-1} \tag{3.4}$$

$$\hat{\sigma}^2_{B} = \frac{\sum_{c=0}^{C-1}\sum_{r=0}^{R-1}(B_{cr} - \overline{B})^2}{RC-1} \tag{3.5}$$

Where,

R is the total number of rows

C is the total number of columns

$\overline{R'}$, $\overline{G}$ and $\overline{B}$ are means of the total of red , green and blue color pixels values in an image.

R'_{cr} , G_{cr} and B_{cr} are the cr^{th} values of red, green and blue color pixels.

$\hat{\sigma}^2_{R'}$, $\hat{\sigma}^2_{R'}$ and $\hat{\sigma}^2_{R'}$ are the variance or probability distribution of red, green and blue color components in an image.

3.3.2 Covariance

Covariance is a measure of the extent to which corresponding elements from two sets of ordered data move in the same direction. We use the following formula to compute covariance.

$$\hat{\sigma}_{R'G} = \frac{\sum_{c=0}^{C-1}\sum_{r=0}^{R-1}(R'_{cr} - \overline{R'})(G_{cr} - \overline{G})}{RC-1} \tag{3.6}$$

$$\hat{\sigma}_{R'B} = \frac{\sum_{c=0}^{C-1}\sum_{r=0}^{R-1}(R'_{cr} - \overline{R'})(B_{cr} - \overline{B})}{RC-1} \tag{3.7}$$

$$\hat{\sigma}_{GB} = \frac{\sum_{c=0}^{C-1}\sum_{r=0}^{R-1}(G_{cr} - \overline{G})(B_{cr} - \overline{B})}{RC-1} \tag{3.8}$$

Where,

R is the total number of rows

C is the total number of columns

$\overline{R'}$, $\overline{G}$ and $\overline{B}$ are means of the total of red , green and blue color pixels values in an image.

R'_{cr} , G_{cr} and B_{cr} are the cr^{th} values of red, green and blue color pixels.

$\hat{\sigma}_{R'G}$, $\hat{\sigma}_{R'B}$ and $\hat{\sigma}_{GB}$ are the covariance of corresponding pixel values in the two sets of color (Red, green or blue).

3.3.3 Variance-Covariance Matrix

Variance and covariance are often displayed together in a variance-covariance matrix, (aka, a covariance matrix). The variances appear along the diagonal and covariances appear in the off-diagonal elements, as shown below.

$$\Sigma = \begin{pmatrix} \hat{\sigma}^2_{R'} & \hat{\sigma}_{R'G} & \hat{\sigma}_{R'B} \\ \hat{\sigma}_{GR'} & \hat{\sigma}^2_{G} & \hat{\sigma}_{GB} \\ \hat{\sigma}_{BR'} & \hat{\sigma}_{BG} & \hat{\sigma}^2_{B} \end{pmatrix} \tag{3.9}$$

Where ,

Σ is a 3 x 3 variance-covariance matrix

$\hat{\sigma}_{R'G}$, $\hat{\sigma}_{R'B}$ and $\hat{\sigma}_{GB}$ are the covariance of corresponding pixel values in the two sets of color (Red, green or blue).

$\hat{\sigma}^2_{R'}$, $\hat{\sigma}^2_{R'}$ and $\hat{\sigma}^2_{R'}$ are the variance or probability distribution of red, green and blue color component in an Image.

3.3.4 Correlation

Correlation between two colors can be defined as the covariance of two colors divided by the square root of the variance of color one and color two. Mathematically it can represented as-

$$\rho_{R'G} = \frac{\hat{\sigma}_{R'G}}{\sqrt{(\hat{\sigma}^2_{R'} * \hat{\sigma}^2_{G})}} \tag{3.10}$$

$$\rho_{R'B} = \frac{\hat{\sigma}_{R'B}}{\sqrt{(\hat{\sigma}^2_{R'} * \hat{\sigma}^2_{R'})}} \tag{3.11}$$

$$\rho_{GB} = \frac{\hat{\sigma}_{GB}}{\sqrt{(\hat{\sigma}^2_{G} * \hat{\sigma}^2_{B})}} \tag{3.12}$$

Where ,

$\rho_{R'G}$ is correlation between red and green.

$\rho_{R'B}$ is correlation between red and blue.

ρ_{GB} is correlation between blue and green.

Below program output shows (figure 3.7) variance-covariance matrix and correlation between different colors in a polluted image-

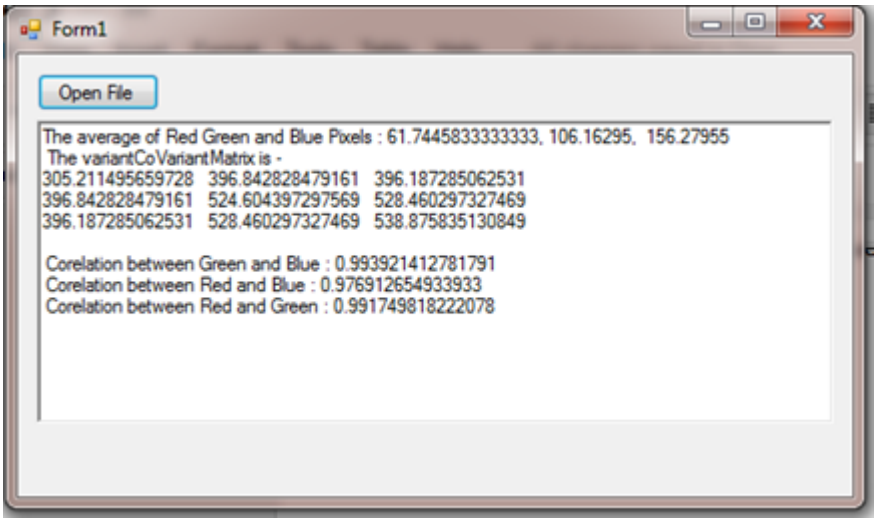

Figure 3.7: Variance-covariance matrix and correlation values for polluted image

3.4 Standard Deviation

Standard deviation (represented by the symbol sigma) shows how much variation or dispersion exists from the average (mean, or expected value). A low standard deviation indicates that the data points tend to be very close to the mean; high standard deviation indicates that the data points are spread out over a large range of values. It can be computed as the square root of variance. The total energy of red, green and blue channels are represented by its variance. By computing the standard deviation, we can reconfirm that the energy observed in high-low frequency bands is greater for a polluted sky in comparison with unpolluted sky (figure 3.5). The more pollutants are present in the atmosphere, the more diversity it will form in the atmosphere, thus increasing the total energy of red, green and blue channels. This can be proved by calculating the standard deviation (square root of variance). The result (Figure 3.8) shows that the standard deviation of red, green and blue in Image2 is almost three times more than what is observed in Image1, thus proving that image2

has more energy in the RGB channels and has pollutants present.

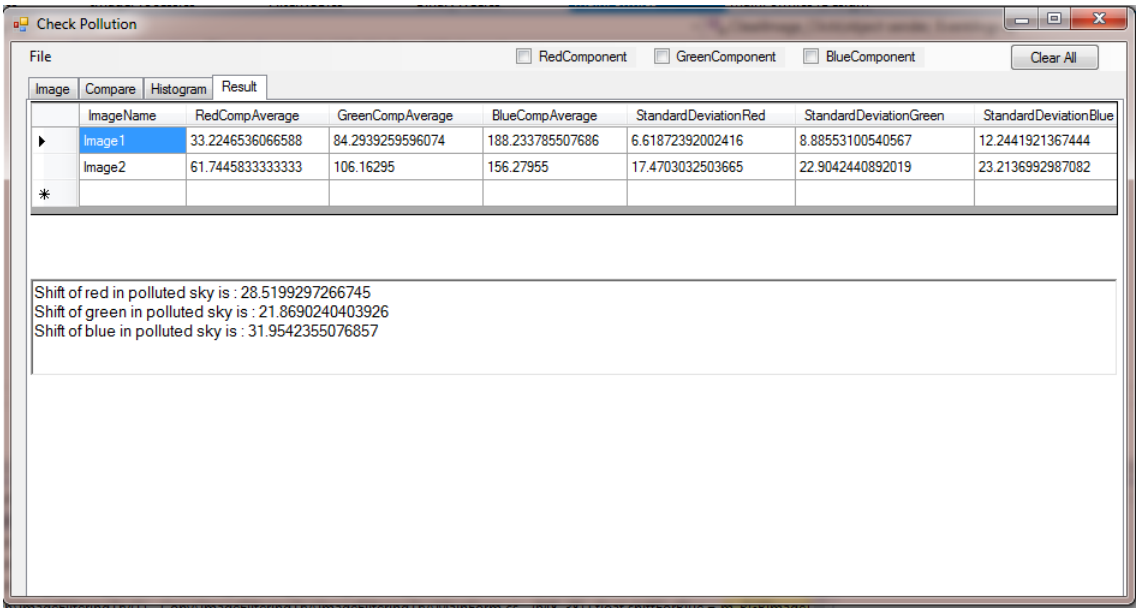

Figure 3.8: Result presenting the standard deviation and average of red, green and blue of an Image1(unpolluted) and Image2(polluted)

3.5 Histogram Analysis

From figure 3.5 and standard deviation results, we analysed the RGB energy in Image1 and Image2, proving image 2 is more polluted. The same can be confirmed by analysing the shift between red, green and blue histograms of polluted sky and unpolluted sky. The peak in every histogram represents the average of red, green and blue components in an image. Compare by how many times the peak has been shifted for a polluted sky in comparison with the unpolluted sky. Figure 3.9 shows the histogram of Image1 unpolluted sky (from Alaska denali national park) and Image2 polluted sky (in Las Vegas). It can be clearly observed that the histograms of red, green and blue are overlapping with each other in the case of a polluted sky. On the other hand, histograms of red, green and blue of the unpolluted sky from alaska shows non-overlapping histograms. This shows that pollutants which are mixed in the air are causing the red, green and blue components values to be changed. This is because pollutants present in the atmosphere cause the red and green components to scatter and attenuate the blue component, thus absorbing the blue component and promoting the red component. Thus blue component histogram shifts to the left and red histogram shifts to the right in the case of polluted sky. (See figure 3.9). The more shift, the more polluted the sky.

Shift value of these images are also given in figure 3.8.

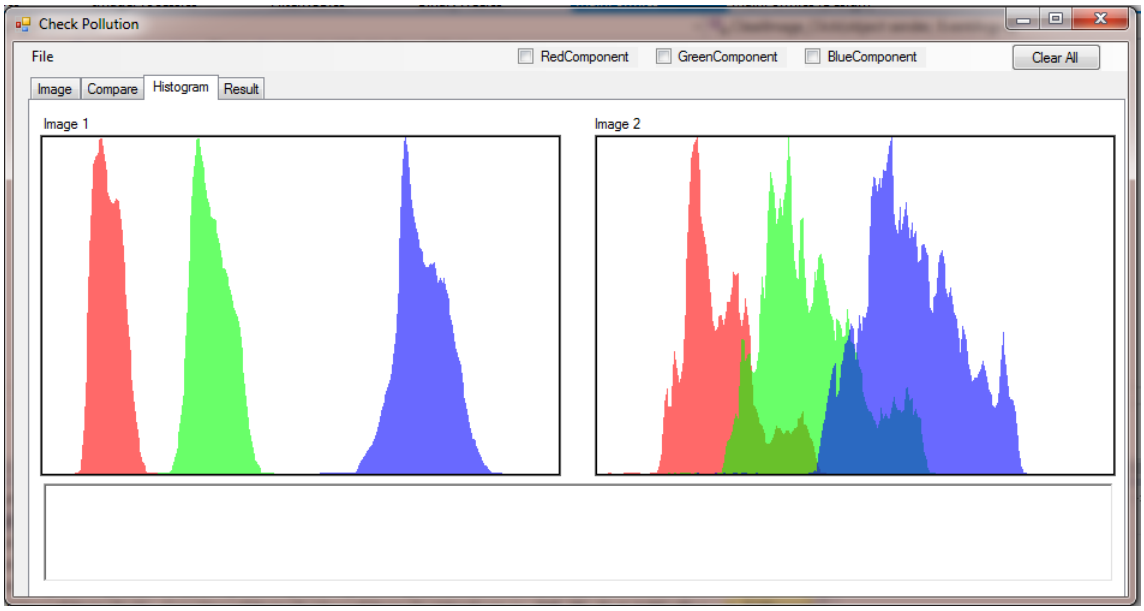

Figure 3.9: Histogram of unpolluted(Image1) and polluted(Image1) sky

There are many different kinds of pollutants. For example particulates and gases in the atmosphere can originate from natural or man-made sources , which includes different types of atmospheric aerosols. Some of them include oil smoke, metallurgical dust, cement dust, smoke, mist, fly ash etc. These types are mentioned in the reference [13]. If a wind is present, the emitted pollutants usually form a plume. Pollutants emitted that are well mixed will appear as a uniform haze. Every pollutant has a different impact when it gets mixed with the air. It disturbs the natural composition and chemistry of the air. Thus when light passes through such polluted air, it tends to deviate from its standard behavior, which can be observed in the figures 3.5 and 3.8. However, more research is needed to analyze which type of pollutants has how much effect on the standard deviation and high-low frequency component. This can be achieved in the laboratory by passing the pollutants in a clean air chamber and analysing the behaviour of air when light is passed through it. This analysis can be done using the program given in the appendix B.

4 Conclusion

In this paper we successfully invented methods to detect pollution in the atmosphere. We achieved this with the help of image processing techniques, which are useful to analyse hidden information in an image. These methods include-

- Analyzing the RGB color channel histogram of an image
- Analyzing the shift in channels in comparison with an unpolluted sky image
- Analyzing the frequency bands in an image using a Haar wavelet
- Analyzing the standard deviation of energies in comparison with an unpolluted sky image

The result of each of the above mentioned methods supports the findings of all the other methods. This paper not only explained the theory regarding the aforementioned methods of pollution detection, but also implemented them effectively. Implemented methods are tested on the sky images from both CMOS and CCD cameras, which showed expected results. This gave the stated theory a stronger proof of its feasibility, which can be implemented in countless areas.

The second chapter briefly explained about transport and transformation of atmospheric pollutants and how it changes atmospheric chemistry. From the histogram and haar wavelet results, we can say that pollutants do affect the color channels (RGB) and their frequency in the atmosphere. However more research is required to check which type of pollutants cause more standard deviation and frequency bands to display more energy. This can be achieved in the laboratory, by carefully observing the air in a chamber and by releasing the pollutants in it.

Appendix A Variance Co-Variance Calculation

A.1 MainForm.cs

```
  using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
//using System.Threading.Tasks;
using System.Windows.Forms;
using System.IO;
using System.Collections;
using System.Drawing.Drawing2D;
namespace VariantCoVariant
{
    public partial class MainForm : Form
    {
        public Image myImage;
        //public PDM ImagePDM;
        public MainForm()
        {
            InitializeComponent();
        }
```

```
private void FileOpenbtn_Click(object sender, EventArgs e)
{
    if (openMyFileDialog.ShowDialog(this) ==
        System.Windows.Forms.DialogResult.OK)
    {
        foreach (string fName in openMyFileDialog.FileNames)
        {
            FileInfo fInfo = new FileInfo(fName);
            if (fInfo.Exists)
            {
                myImage = Image.FromFile(fName);
            }
        }
    }
    PDM ImagePDM = new PDM(myImage);

    string text = "The average of Red Green and Blue Pixels : " +
        ImagePDM.RGB.Avrg_Red + ", " +
        ImagePDM.RGB.Avrg_Green + ",  " + ImagePDM.RGB.Avrg_Blue ;
    text = text + "\n The variantCoVariantMatrix is - \n";
    double[][] PDMMatrix = ImagePDM.GetPDM();
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            text = text + PDMMatrix[i][j] + "   ";
        }
        text = text + "\n";
    }
    text = text + "\n Corelation between Green and Blue : " +
        ImagePDM.CorelationGB;
    text = text + "\n Corelation between Red and Blue : " +
        ImagePDM.CorelationRB;
    text = text + "\n Corelation between Red and Green : " +
```

```
                ImagePDM.CorelationRG;
            richTextBox.Text = text;

        }
    }
}
```

A.2 RGBPixel.cs

```
 using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
//using System.Threading.Tasks;
using System.IO;
using System.Drawing;
using System.Windows.Forms;
namespace VariantCoVariant
{
    class RGBPixel
    {
        private int[][] m_RedPix;
        private int[][] m_GreenPix;
        private int[][] m_BluePix;
        private Size s_size;
        public double Avrg_Red, Avrg_Green, Avrg_Blue, totalPixel;
        public RGBPixel(Bitmap CurrentImage)
        {
            //filterRGB = new FilterRGB;
            s_size = CurrentImage.Size;

            Color[][] clr = null;

            // delete data in m_RedPixel
            m_BluePix = null;
```

```
        m_GreenPix = null;
        m_RedPix = null;
        int[][] RedPix = new int[s_size.Width][];
        int[][] GreenPix = new int[s_size.Width][];
        int[][] BluePix = new int[s_size.Width][];
        clr = new Color[s_size.Width][];
        for (int i = 0; i < s_size.Width; i++)
        {
            clr[i] = new Color[s_size.Height];
            RedPix[i] = new int[s_size.Height];
            GreenPix[i] = new int[s_size.Height];
            BluePix[i] = new int[s_size.Height];
        }//end of for
        if (CurrentImage != null)
        {
            for (int x = 0; x < s_size.Width; x++)
            {
                for (int y = 0; y < s_size.Height; y++)
                {
                    clr[x][y] = CurrentImage.GetPixel(x, y);
                    RedPix[x][y] = clr[x][y].R;
                    GreenPix[x][y] = clr[x][y].G;
                    BluePix[x][y] = clr[x][y].B;
                } // end for
            } // end for
        }// end of if (CurrentImage != null)
        m_RedPix = RedPix;
        m_BluePix = BluePix;
        m_GreenPix = GreenPix;
    }
    public int[][] GetBluePixel()
    {
        return m_BluePix;
    }
```

```
        public int[][] GetRedPixel()
        {
            return m_RedPix;
        }
        public int[][] GetGreenPixel()
        {
            return m_GreenPix;
        }
        public void ComputeAvrgRGB()
        {
            totalPixel = s_size.Width * s_size.Height;
            Avrg_Blue = Avrg_Green = Avrg_Red = 0;
            for (int i = 0; i < s_size.Width; i++)
            {
                for (int j = 0; j < s_size.Height; j++)
                {
                    Avrg_Red = Avrg_Red + m_RedPix[i][j];
                    Avrg_Green = Avrg_Green + m_GreenPix[i][j];
                    Avrg_Blue = Avrg_Blue + m_BluePix[i][j];
                }
            }
            Avrg_Red = Avrg_Red / totalPixel;
            Avrg_Green = Avrg_Green / totalPixel;
            Avrg_Blue = Avrg_Blue / totalPixel;
        }
    }
}
```

A.3 VCVMatrix.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
```

```
using System.Drawing;
namespace VarianceCoVariance
{
    // positive definite Matrix
    class VCVMatrix
    {
        /* public struct MatrixElements
         {
             public float Value;
             public Char Color;
         } */
         private double[][] PDMatrix;
         private int Matrixsize;

         private Image MyImage;
         private const int nPixelR = 0;
         private const int nPixelG = 1;
         private const int nPixelB = 2;
         public RGBPixel RGB;
         public double CorelationRB, CorelationRG, CorelationGB;
         public VCVMatrix(Image Image)
         {
             MyImage = Image;
             Bitmap OriginalImage = new Bitmap(MyImage);
             RGB = new RGBPixel(OriginalImage);
             RGB.ComputeAvrgRGB();
             Matrixsize = 3;
             PDMatrix = new double[Matrixsize][];
             for (int i = 0; i < Matrixsize; i++)
             {
                 PDMatrix[i] = new double[Matrixsize];
             }
           /*  for (int i = 0; i < Matrixsize; i++)
             {
```

```
            for (int j = 0; j < Matrixsize; j++)
            {
                PDMatrix[i][j].Color = 'R';
                PDMatrix[i][j+1].Color = 'G';
                PDMatrix[i][j + 2].Color = 'B';
            }
        }*/
    }
    public double[][] GetMatrix()
    {
        //TreeNode = MyImage;
        for (int i = 0; i < Matrixsize; i++)
        {
            for (int j = 0; j < Matrixsize; j++)
            {
                double Average_i = 0;
                double Avergae_j = 0;
                int[][] ColorValues_i = GetColor(i,ref Average_i);
                int[][] ColorValues_j = GetColor(j,ref Avergae_j);
                 PDMatrix[i][j] = ComputeSigma(ColorValues_i,
                     ColorValues_j,
                     Average_i,Avergae_j);

            }
         }
        CorelationRB = ComputeCorelation(nPixelR, nPixelB);
        CorelationRG = ComputeCorelation(nPixelR, nPixelG);
        CorelationGB = ComputeCorelation(nPixelG, nPixelB);
        return PDMatrix;
    }
    private int[][] GetColor(int Color, ref double average)
    {
        switch (Color)
        {
```

```
            case nPixelR :
                  average = RGB.Avrg_Red;
                  return RGB.GetRedPixel();
                 //break;
            case nPixelG:
                  average = RGB.Avrg_Green;
                  return RGB.GetGreenPixel();
            case nPixelB:
                  average = RGB.Avrg_Blue;
                  return RGB.GetBluePixel();
            default :
                  average = RGB.Avrg_Red;
                  return RGB.GetRedPixel();
        }
    }
     private double ComputeSigma(int[][] Color_i,
         int[][] Color_j,
         double Average_i,
         double Average_j)
     {
         double Sigma =0;
         double check1 = 0, check2 = 0;
         double temp1 = 0, temp2 = 0;
         int cnt =0;
        for (int i = 0; i < MyImage.Size.Width; i++)
        {
            for (int j = 0; j < MyImage.Size.Height; j++)
            {
                temp1 = Color_i[i][j] - Average_i;
                temp2 = Color_j[i][j] - Average_j;
                // [for debugging
                if (Color_j[i][j] < Average_j)
                    cnt++;
                check1 = check1 + temp1;
```

```
                check2 = check2 + temp2;
                //]
                // summation
                Sigma = Sigma + (temp1 * temp2);
            }
        }
        int check3 = (int)check2;
        return (Sigma / RGB.totalPixel - 1);
    }
     // Computer average of Red , blue and green

    public double ComputeCorelation(int Color1, int Color2)
    {
        double Corelation = 0;
        // Variance of color1 and color2
        double Color1Covariant = Math.Sqrt(PDMatrix[Color1][Color1]);
        double Color2Covariant = Math.Sqrt(PDMatrix[Color2][Color2]);
         Corelation = PDMatrix[Color1][Color2] /
             (Color1Covariant * Color2Covariant);

        return Corelation;
    }
  }
}
```

Appendix B Histogram and Haar Wavelet Implementation

B.1 MainForm.cs

```
  using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
//using System.Threading.Tasks;
using System.Windows.Forms;
using System.IO;
using System.Collections;
using System.Drawing.Drawing2D;
namespace ImageFilteringTry
{
    public partial class MainForm : Form
    {
        private FilterRGB m_SelectFilter, m_SelectFilter_Clean,
            m_SelectFilter_Polluted;
        //private Bitmap redComponent, greenComponent, blueComponent;
        private Bitmap m_OriginalImage, m_CleanImage, m_PolluteImage;
        private Histogram m_DataImage1, m_DataImage2;
        // [for 1st tab
```

```
private Bitmap m_LowCompImage, m_HighCompImage;
int ProcessCountLow, ProcessCountHigh;
private List<BinaryTreeNode> m_LeafNodes =
    new List<BinaryTreeNode>();
private List<Bitmap> m_LeafNodesDisplayImage =
    new List<Bitmap>();
//]
// [for Clean image
private Bitmap m_LowCompImage_Clean, m_HighCompImage_Clean;
int m_ProcessCountLow_Clean, m_ProcessCountHigh_Clean;
private List<BinaryTreeNode> m_LeafNodes_Clean =
    new List<BinaryTreeNode>();
private List<Bitmap> m_LeafNodesDisplayImage_Clean =
    new List<Bitmap>();
//]
//[ for Polluted image
private Bitmap m_LowCompImage_Polluted, m_HighCompImage_Polluted;
int m_ProcessCountLow_Polluted, m_ProcessCountHigh_Polluted;
private List<BinaryTreeNode> m_LeafNodes_Polluted =
    new List<BinaryTreeNode>();
private List<Bitmap> m_LeafNodesDisplayImage_Polluted =
    new List<Bitmap>();
//]
public BinaryTree m_BT, m_BT_Clean, m_BT_Polluted;
ImageProcess IP;
public MainForm()
{
    m_OriginalImage = null;
    IP = new ImageProcess();
    ProcessCountLow = ProcessCountHigh = 0;
    m_ProcessCountLow_Clean = m_ProcessCountHigh_Clean =
    m_ProcessCountLow_Polluted = m_ProcessCountHigh_Polluted = 0;
    ImageProcess.CurrentProcessCntHigh =
        ImageProcess.CurrentProcessCntLow = 0;
```

```
        InitializeComponent();
    }
    private void openToolStripMenuItem_Click(object sender,
        EventArgs e)
    {
        if (ImageFilterTab.SelectedTab ==
            ImageFilterTab.TabPages["ImageTab"])
        {
            if (openMyFileDialog.ShowDialog(this) ==
                System.Windows.Forms.DialogResult.OK)
            {
                foreach (string fName in openMyFileDialog.FileNames)
                {
                    FileInfo fInfo = new FileInfo(fName);
                    if (fInfo.Exists)
                    {
                        Image image = Image.FromFile(fName);
                        /*if ((image.Size.Width * image.Size.Height) >
                          *(ClearImage.Size.Width * ClearImage.Size.Height))
                        {
                            Point point = new Point(0, 0);
                            Rectangle rec =
                             * new Rectangle(point, ClearImage.Size);
                            image = ImageProcess.cropImage(image, rec);
                            //image = resizeImage(image, ClearImage.Size);
                        }*/
                        MyImage.Image = image;
                        m_OriginalImage = new Bitmap(image);
                        ImageProcess.bitmap =
                            new Bitmap(m_OriginalImage.Width,
                                       m_OriginalImage.Height);
                        using (Bitmap bmp = new Bitmap(image))
                        {
                            m_BT = new BinaryTree(bmp);
```

```
                        //.Root.image = bmp;
                        //.Root.imageName = "Original";
                        m_SelectFilter = new FilterRGB(bmp);
                        ImageProcess.CurrentFilter = m_SelectFilter;
                        m_HighCompImage = (Bitmap)bmp.Clone();
                        m_LowCompImage = (Bitmap)bmp.Clone();
                        //flowLayoutPanel1.AutoScroll = true;
                        //flowLayoutPanel1.Controls.Clear();
                        ProcessCountLow = ProcessCountHigh = 0;
                    }
                } // end if (fInfo.Exists)
            } // end for
        } // end if
    }
    else if (ImageFilterTab.SelectedTab ==
        ImageFilterTab.TabPages["CompareTab"])
    {
    }
}
// End of void openToolStripMenuItem_Click(object sender, EventArgs e)
//Blue
private void checkBox3_CheckedChanged(object sender, EventArgs e)
{
    ImageProcess.ColorSelected = 'B';
    if (BluecheckBox.Checked)
    {
        GreencheckBox.Checked = false;
        RedcheckBox.Checked = false;
        ProcessCountHigh = 0;
        ProcessCountLow = 0;
        if (m_OriginalImage != null)
        {
            m_LowCompImage = (Bitmap)m_OriginalImage.Clone();
            m_HighCompImage = (Bitmap)m_OriginalImage.Clone();
```

```
        }
    }
}
//Green
private void checkBox2_CheckedChanged(object sender, EventArgs e)
{
    ImageProcess.ColorSelected = 'G';
    if (GreencheckBox.Checked)
        //&& (BluecheckBox.Checked || RedcheckBox.Checked))
    {
        BluecheckBox.Checked = false;
        RedcheckBox.Checked = false;
        ProcessCountHigh = 0;
        ProcessCountLow = 0;
        if (m_OriginalImage != null)
        {
            m_LowCompImage = (Bitmap)m_OriginalImage.Clone();
            m_HighCompImage = (Bitmap)m_OriginalImage.Clone();
        }
    }
}
//Red
private void RedcheckBox_CheckedChanged(object sender, EventArgs e)
{
    ImageProcess.ColorSelected = 'R';
    if (ImageFilterTab.SelectedTab ==
        ImageFilterTab.TabPages["ImageTab"])
    {
        if (RedcheckBox.Checked)
            // && (BluecheckBox.Checked || GreencheckBox.Checked))
        {
            //MessageBox.Show("You must check only one check box.",
            //"", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
            BluecheckBox.Checked = false;
```

```
            GreencheckBox.Checked = false;
            ProcessCountHigh = 0;
            ProcessCountLow = 0;
            if (m_OriginalImage != null)
            {
                m_LowCompImage = (Bitmap)m_OriginalImage.Clone();
                m_HighCompImage = (Bitmap)m_OriginalImage.Clone();
            }
        }
    }// end of if
    else if (ImageFilterTab.SelectedTab ==
        ImageFilterTab.TabPages["CompareTab"])
    {
        if (RedcheckBox.Checked)
            // && (BluecheckBox.Checked || GreencheckBox.Checked))
        {
            //MessageBox.Show("You must check only one check box.",
            //"", MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
            BluecheckBox.Checked = false;
            GreencheckBox.Checked = false;
            ProcessCountHigh = 0;
            ProcessCountLow = 0;
            if (m_CleanImage != null && m_PolluteImage != null)
            {
                m_LowCompImage_Clean =
                    (Bitmap)m_CleanImage.Clone();
                m_HighCompImage_Clean =
                    (Bitmap)m_CleanImage.Clone();
                m_LowCompImage_Polluted =
                    (Bitmap)m_PolluteImage.Clone();
                m_HighCompImage_Polluted =
                    (Bitmap)m_PolluteImage.Clone();
            }
        }
```

```
    }
}
/* // High compoenet
private void checkBox1_CheckedChanged(object sender, EventArgs e)
 {
     if (HighCompCheckBox.Checked)
     {
         LowCompCheckBox.Checked = false;
     }
 }
private void LowComponent_CheckedChanged(object sender, EventArgs e)
 {
     if (LowCompCheckBox.Checked)
     {
         HighCompCheckBox.Checked = false;
     }
 } */
private void ProcessImage_Click(object sender, EventArgs e)
{
    m_LeafNodes.Clear();
    m_LeafNodesDisplayImage.Clear();
    if (m_OriginalImage != null)
    {
        ImageProcess.CurrentImage = m_OriginalImage;
        //if (LowCompCheckBox.Checked)
        {
            ProcessCountLow++;
            // LowComponentProcess(m_LowCompImage);
        }
        //else if (HighCompCheckBox.Checked)
        {
            ProcessCountHigh++;
            ImageProcess.CurrentFilter = m_SelectFilter;
            ImageProcess.CurrentProcessCntLow = ProcessCountLow;
```

```
                ImageProcess.CurrentProcessCntHigh = ProcessCountHigh;
                //HighComponentProcess(m_HighCompImage);
                ImageProcess.LeafNodes_Current = m_LeafNodes;
                ImageProcess.LeafNodesDisplayImage_Current =
                    m_LeafNodesDisplayImage;
                m_BT.Pre_add();
                IP.GetLeafNode(m_BT.Root);
                MyImageProcessed.Image = IP.DisplayAllLeafNodes();
            }
        }
        else
        {
            MessageBox.Show("Please Select Image.", "Hey",
                MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
        }
    } // end of private void ProcessImage_Click
    private void flowLayoutPanel1_Paint(object sender, PaintEventArgs e)
    {
    }
    private void label2_Click(object sender, EventArgs e)
    {
    }
    private void Image_Click(object sender, EventArgs e)
    {
    }
    private void panel1_Paint(object sender, PaintEventArgs e)
    {
        if (m_BT_Clean != null )
        {
            //Boolean bCombine = chkCombine.Checked;
            Panel pan = sender as Panel;
            Rectangle rc = new Rectangle();

            int[] lpData = new int[256];
```

```
            Color clr;
            // draw the histograms
            // retrieve the dimensions of the drawing area
            if (pan != null)
                rc = pan.ClientRectangle;

            e.Graphics.Clear(Color.White);
            m_DataImage1.GetHistogram(Histogram.ColorType.RED_COLOR,
                lpData);
            clr = Color.FromArgb(150,
                Histogram.HistogramColor(Histogram.ColorType.RED_COLOR));
            drawHistogram(e.Graphics, lpData, rc, clr,
                m_DataImage1.GetMaxIndex(Histogram.ColorType.RED_COLOR));
            m_DataImage1.GetHistogram(Histogram.ColorType.GREEN_COLOR,
                lpData);
            clr = Color.FromArgb(150,
                Histogram.HistogramColor(Histogram.ColorType.GREEN_COLOR));
            drawHistogram(e.Graphics, lpData, rc, clr,
                m_DataImage1.GetMaxIndex(Histogram.ColorType.GREEN_COLOR));
            m_DataImage1.GetHistogram(Histogram.ColorType.BLUE_COLOR,
                lpData);
            clr = Color.FromArgb(150,
                Histogram.HistogramColor(Histogram.ColorType.BLUE_COLOR));
            drawHistogram(e.Graphics, lpData, rc, clr,
                m_DataImage1.GetMaxIndex(Histogram.ColorType.BLUE_COLOR));
        }
        else
        {
            e.Graphics.Clear(Color.White);
            MessageBox.Show("Please load the image 1", "Hey",
                MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
        }
    }
    private void drawHistogram(Graphics grp, int[] lpData, Rectangle rc,
```

```
        Color color, int iMaxIndex)
    {
        //int m_iIntensity = -1;
        drawHistogram(grp, lpData, rc, color, iMaxIndex, -1);
    }
    private void drawHistogram(Graphics grp, int[] lpData, Rectangle rc,
        Color color, int iMaxIndex, int iIntensity)
    {
        // draw the data graph histogram for the sent
        // data (256 integers) into the sent Graphics canvas
        float fPixWidth, fPixHeight;
        int iArrMax, i;
        int ptArrSize = lpData.GetLength(0);
        Point[] pt = new Point[ptArrSize + 2];
        // create initial and final points for the histogram polygon
        pt[0].X = rc.Left;
        pt[0].Y = rc.Bottom;
        pt[pt.GetLength(0) - 1].X = rc.Right;
        pt[pt.GetLength(0) - 1].Y = rc.Bottom;
        // find max of data
        iArrMax = lpData[iMaxIndex];
        // scale the points
        fPixWidth = rc.Width / (float)ptArrSize;
        fPixHeight = rc.Height / (float)iArrMax;
        // generate the scaled points based on histogram data
        for (i = 0; i < ptArrSize; i++)
        {
            // help giving visual aid when there are pixels
            // in the value, but the
            // max pixels is so large that the smaller values
            //are not shown at scale
            int preY = (int)(lpData[i] * fPixHeight);
            if (preY <= 1 && lpData[i] > 0)
                preY = 2;
```

```
            pt[i + 1].Y = rc.Bottom - preY;
            pt[i + 1].X = (int)(i * fPixWidth) + rc.Left;
        } // end for
        // draw histogram
        grp.FillPolygon(new SolidBrush(color), pt);
        if (iIntensity >= 0 && iIntensity <= 255)
        {
            // draw the desired line
            grp.DrawLine(new Pen(Color.Black, fPixWidth),
                iIntensity * fPixWidth,
                0,
                iIntensity * fPixWidth,
                rc.Bottom);
        } // end if
        grp.DrawRectangle(Pens.Black, rc.X, rc.Y, rc.Width - 1,
            rc.Height - 1); // black border
    }
    private void tabPage1_Click(object sender, EventArgs e)
    {
    }
    private void Compare_Click(object sender, EventArgs e)
    {
        while (m_ProcessCountHigh_Clean < 2)
        {
            m_LeafNodes_Clean.Clear();
            m_LeafNodesDisplayImage_Clean.Clear();
            m_LeafNodes_Polluted.Clear();
            m_LeafNodesDisplayImage_Polluted.Clear();
            if (m_CleanImage != null && m_PolluteImage != null)
            {
                //if (LowCompCheckBox.Checked)
                {
                    m_ProcessCountLow_Clean++;
                    m_ProcessCountLow_Polluted++;
```

```
            // LowComponentProcess(m_LowCompImage);
        }
        //else if (HighCompCheckBox.Checked)
        {
            m_ProcessCountHigh_Polluted++;
            m_ProcessCountHigh_Clean++;
            //HighComponentProcess(m_HighCompImage);
            ImageProcess.CurrentImage = m_CleanImage;
            ImageProcess.CurrentFilter = m_SelectFilter_Clean;
            ImageProcess.CurrentProcessCntLow =
                m_ProcessCountLow_Clean;
            ImageProcess.CurrentProcessCntHigh =
                m_ProcessCountHigh_Clean;
            ImageProcess.LeafNodes_Current = m_LeafNodes_Clean;
            ImageProcess.LeafNodesDisplayImage_Current =
                m_LeafNodesDisplayImage_Clean;
            m_BT_Clean.Pre_add();
            IP.GetLeafNode(m_BT_Clean.Root);
            ImageProcess.bitmap = null;
            ImageProcess.bitmap = new Bitmap(ClearImage.Image);
//bitmap = new Bitmap(m_CleanImage.Width, m_CleanImage.Height);
            ClearImage.Image = IP.DisplayAllLeafNodes();
            // For polluted
            ImageProcess.CurrentImage = null;
            ImageProcess.CurrentImage =
                m_PolluteImage;
            ImageProcess.CurrentFilter =
                m_SelectFilter_Polluted;
            ImageProcess.CurrentProcessCntLow =
                m_ProcessCountLow_Polluted;
            ImageProcess.CurrentProcessCntHigh =
                m_ProcessCountHigh_Polluted;
            ImageProcess.LeafNodes_Current =
                m_LeafNodes_Polluted;
```

```
                ImageProcess.LeafNodesDisplayImage_Current =
                    m_LeafNodesDisplayImage_Polluted;
                m_BT_Polluted.Pre_add();
                IP.GetLeafNode(m_BT_Polluted.Root);
                ImageProcess.bitmap = null;
                ImageProcess.bitmap =
                    new Bitmap(PollutedImage.Image);
                //bitmap = new Bitmap(m_PolluteImage.Width,
                //m_PolluteImage.Height);
                PollutedImage.Image = IP.DisplayAllLeafNodes();
            }
        }
        else
        {
            MessageBox.Show("Please Select Image.", "Hey",
                MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
        }
    }// end of while (< 3)
    // Logic for computing Probability distribution
    if (m_BT_Clean.Root.High != null &&
        m_BT_Polluted.Root.High != null)
    {
        if (m_BT_Clean.Root.High.Low != null &&
            m_BT_Polluted.Root.High.Low != null)
        {
            BinaryTreeNode HighLow_Clean = m_BT_Clean.Root.High.Low;
            BinaryTreeNode HighLow_Polluted =
                m_BT_Polluted.Root.High.Low;
            ProbabiltyDistribution PD =
            new ProbabiltyDistribution(HighLow_Clean,
                                       HighLow_Polluted,
                                       ImageProcess.ColorSelected);
            double PDClean = 0, PDPolluted = 0;
            PD.GetAverage();
```

```
                PD.GetProbalityDist(ref PDClean, ref PDPolluted);
                textBox1.Text =
                "Result: Image 1 Probability distribution is - " +
                PDClean +
                ". Image 2 Probability distribution is - "
                + PDPolluted + ". Ratio is - " + PDClean / PDPolluted;
            }
        }
    } // end of private void COmapre_Click
    private void ClearImage_Click(object sender, EventArgs e)
    {
        if (openMyFileDialog.ShowDialog(this) ==
            System.Windows.Forms.DialogResult.OK)
        {
            foreach (string fName in openMyFileDialog.FileNames)
            {
                FileInfo fInfo = new FileInfo(fName);
                if (fInfo.Exists)
                {
                    Image image = Image.FromFile(fName);
                    /* if ((image.Size.Width * image.Size.Height) >
                     * (ClearImage.Size.Width * ClearImage.Size.Height))
                    {
                        Point point = new Point(0, 0);
                        Rectangle rec =
                     * new Rectangle(point, ClearImage.Size);
                        image = ImageProcess.cropImage(image, rec);
                        //image = resizeImage(image, ClearImage.Size);
                    } */
                    ClearImage.Image = image;
                    m_CleanImage = new Bitmap(image);
                    using (Bitmap bmp = new Bitmap(image))
                    {
                        m_DataImage1 = new Histogram(bmp);
```

```
                    m_BT_Clean = new BinaryTree(bmp);
                    //.Root.image = bmp;
                    //.Root.imageName = "Original";
                    m_SelectFilter_Clean = new FilterRGB(bmp);
                    ImageProcess.CurrentFilter =
                        m_SelectFilter_Clean;
                    m_HighCompImage_Clean = (Bitmap)bmp.Clone();
                    m_LowCompImage_Clean = (Bitmap)bmp.Clone();
                    //flowLayoutPanel1.AutoScroll = true;
                    //flowLayoutPanel1.Controls.Clear();
                    m_ProcessCountLow_Clean =
                        m_ProcessCountHigh_Clean = 0;
                }
            } // end if (fInfo.Exists)
        } // end for
    } // end if
}
private void PollutedImage_Click(object sender, EventArgs e)
{
    if (openMyFileDialog.ShowDialog(this) ==
        System.Windows.Forms.DialogResult.OK)
    {
        foreach (string fName in openMyFileDialog.FileNames)
        {
            FileInfo fInfo = new FileInfo(fName);
            if (fInfo.Exists)
            {
                Image image = Image.FromFile(fName);
                /* if ((image.Size.Width * image.Size.Height) >
                 * (ClearImage.Size.Width * ClearImage.Size.Height))
                {
                    Point point = new Point(0, 0);
                    Rectangle rec =
                 * new Rectangle(point, ClearImage.Size);
```

```
                    image = ImageProcess.cropImage(image, rec);
                    //image = resizeImage(image, ClearImage.Size);
                }*/
                PollutedImage.Image = image;
                m_PolluteImage = new Bitmap(image);
                using (Bitmap bmp = new Bitmap(image))
                {
                    m_BT_Polluted = new BinaryTree(bmp);
                    m_DataImage2 = new Histogram(bmp);
                    //.Root.image = bmp;
                    //.Root.imageName = "Original";
                    m_SelectFilter_Polluted = new FilterRGB(bmp);
                    ImageProcess.CurrentFilter = m_SelectFilter_Polluted;
                    m_HighCompImage_Polluted = (Bitmap)bmp.Clone();
                    m_LowCompImage_Polluted = (Bitmap)bmp.Clone();
                    //flowLayoutPanel1.AutoScroll = true;
                    //flowLayoutPanel1.Controls.Clear();
                    m_ProcessCountLow_Polluted =
                        m_ProcessCountHigh_Polluted = 0;
                }
            } // end if (fInfo.Exists)
        } // end for
    } // end if
    FillDataGridResult();
}
private void textBox1_TextChanged(object sender, EventArgs e)
{
}
private void CompareGreenBox_CheckedChanged(object sender,
    EventArgs e)
{
}
private void label2_Click_1(object sender, EventArgs e)
{
```

```
}
private void MyImage_Click(object sender, EventArgs e)
{
}
// For getting positive definite Matrix
private void PDMButton_Click(object sender, EventArgs e)
{
    VariantCoVariant ImagePDM = new VariantCoVariant();
   double [][]PDMMatrix = ImagePDM.GetPDM(m_BT.Root);
   double t = PDMMatrix[0][0];
}
private void HistogramTab_Click(object sender, EventArgs e)
{
    HistImage1.Invalidate();
    HistImage2.Invalidate();
    MessageBox.Show("Please load image.", "Hey",
        MessageBoxButtons.OK, MessageBoxIcon.Exclamation);

}
private void HistImage2_Paint(object sender, PaintEventArgs e)
{
    Hashtable PointCount = new Hashtable();
    if (m_BT_Polluted != null)
    {
        //Boolean bCombine = chkCombine.Checked;
        Panel pan = sender as Panel;
        Rectangle rc = new Rectangle();
        int[] lpData = new int[256];
        Color clr;
        // draw the histograms
        // retrieve the dimensions of the drawing area
        if (pan != null)
            rc = pan.ClientRectangle;
        e.Graphics.Clear(Color.White);
```

```
            // draw red color histogram
            m_DataImage2.GetHistogram(Histogram.ColorType.RED_COLOR,
                lpData);
            clr = Color.FromArgb(150,
            Histogram.HistogramColor(Histogram.ColorType.RED_COLOR));
            drawHistogram(e.Graphics, lpData, rc, clr,
            m_DataImage2.GetMaxIndex(Histogram.ColorType.RED_COLOR));
            // draw green color histogram
            m_DataImage2.GetHistogram(Histogram.ColorType.GREEN_COLOR, lpData);
            clr = Color.FromArgb(150,
            Histogram.HistogramColor(Histogram.ColorType.GREEN_COLOR));
            drawHistogram(e.Graphics, lpData, rc, clr,
            m_DataImage2.GetMaxIndex(Histogram.ColorType.GREEN_COLOR));
            // draw blue color histogram
            m_DataImage2.GetHistogram(Histogram.ColorType.BLUE_COLOR, lpData);
            clr = Color.FromArgb(150,
            Histogram.HistogramColor(Histogram.ColorType.BLUE_COLOR));
            drawHistogram(e.Graphics, lpData, rc, clr,
            m_DataImage2.GetMaxIndex(Histogram.ColorType.BLUE_COLOR));
        }
        else
        {
            e.Graphics.Clear(Color.White);
            MessageBox.Show("Please load image 2", "Hey",
                MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
        }
    }
    private void FillDataGridResult()
    {
        string[] lpsRow = new string[7];
        VariantCoVariant VcoVImage1 = new VariantCoVariant();
        VcoVImage1.GetPDM(m_BT_Clean.Root);
        VariantCoVariant VcoVImage2 = new VariantCoVariant();
        VcoVImage2.GetPDM(m_BT_Polluted.Root);
```

```
        //for (int i = 0; i < 2; i++)
        {
            lpsRow[0] = "Image1";
            lpsRow[1] = VcoVImage1.Avrg_Red.ToString();
            lpsRow[2] = VcoVImage1.Avrg_Green.ToString();
            lpsRow[3] = VcoVImage1.Avrg_Blue.ToString();
            lpsRow[4] = VcoVImage1.GetStandardDeviation(0).ToString();
            lpsRow[5] = VcoVImage1.GetStandardDeviation(1).ToString();
            lpsRow[6] = VcoVImage1.GetStandardDeviation(2).ToString();
            this.dataGridResult.Rows.Add(lpsRow);
            lpsRow[0] = "Image2";
            lpsRow[1] = VcoVImage2.Avrg_Red.ToString();
            lpsRow[2] = VcoVImage2.Avrg_Green.ToString();
            lpsRow[3] = VcoVImage2.Avrg_Blue.ToString();
            lpsRow[4] = VcoVImage2.GetStandardDeviation(0).ToString();
            lpsRow[5] = VcoVImage2.GetStandardDeviation(1).ToString();
            lpsRow[6] = VcoVImage2.GetStandardDeviation(2).ToString();
            this.dataGridResult.Rows.Add(lpsRow);
            String text;
            text = "Shift of red in polluted sky is : " +
                Math.Abs(VcoVImage2.Avrg_Red - VcoVImage1.Avrg_Red);
            text = text + "\nShift of green in polluted sky is : " +
                Math.Abs(VcoVImage2.Avrg_Green - VcoVImage1.Avrg_Green);
            text = text + "\nShift of blue in polluted sky is : " +
                Math.Abs(VcoVImage1.Avrg_Blue - VcoVImage2.Avrg_Blue);
            richTextBox_Shift.Text = text;
        }
    }
    private void label5_Click(object sender, EventArgs e){
    }
    private void Result_Click(object sender, EventArgs e){
    }
    private void tableLayoutPanel1_Paint(object sender,
        PaintEventArgs e){
```

```
        }
        private void dataGridView1_CellContentClick(object sender,
            DataGridViewCellEventArgs e){
        }
    }
}
```

B.2 RGBPixel.cs

```
  using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
//using System.Threading.Tasks;
using System.IO;
using System.Drawing;
using System.Windows.Forms;
namespace ImageFilteringTry
{
    class RGBPixel
    {
        private int[][] m_RedPix;
        private int[][] m_GreenPix;
        private int[][] m_BluePix;
        private float Avrg_Blue, Avrg_Green, Avrg_Red;
        public RGBPixel(Bitmap CurrentImage)
        {
            //filterRGB = new FilterRGB;
            Size s_size = CurrentImage.Size;

            Color[][] clr = null;
            // delete data in m_RedPixel
            m_BluePix = null;
            m_GreenPix = null;
            m_RedPix = null;
```

```
int [][] RedPix = new int[s_size.Width][];
int [][] GreenPix = new int[s_size.Width][];
int [][] BluePix = new int[s_size.Width][];
clr = new Color[s_size.Width][];
for (int i = 0; i < s_size.Width; i++)
{
    clr[i] = new Color[s_size.Height];
    RedPix[i] = new int[s_size.Height];
    GreenPix[i] = new int[s_size.Height];
    BluePix[i] = new int[s_size.Height];
}//end of for
if (CurrentImage != null)
{
    for (int x = 0; x < s_size.Width; x++)
    {
        for (int y = 0; y < s_size.Height; y++)
        {
            clr[x][y] = CurrentImage.GetPixel(x, y);
            RedPix[x][y] = clr[x][y].R;
            GreenPix[x][y] = clr[x][y].G;
            BluePix[x][y] = clr[x][y].B;
            // add all pixel value to compute average
            Avrg_Red = Avrg_Red + RedPix[x][y];
            Avrg_Green = Avrg_Green + GreenPix[x][y];
            Avrg_Blue = Avrg_Blue + BluePix[x][y];
        } // end for
    } // end for
}// end of if (CurrentImage != null)
m_RedPix = RedPix;
m_BluePix = BluePix;
m_GreenPix = GreenPix;
int totalPixel = s_size.Width * s_size.Height;
Avrg_Red = Avrg_Red / totalPixel;
Avrg_Green = Avrg_Green / totalPixel;
```

```
            Avrg_Blue = Avrg_Blue / totalPixel;
        }
        public int[][] GetBluePixel()
        {
            return m_BluePix;
        }
        public int[][] GetRedPixel()
        {
            return m_RedPix;
        }
        public int[][] GetGreenPixel()
        {
            return m_GreenPix;
        }
        public float GetRed_Avrg()
        {
            return Avrg_Red;
        }
        public float GetGreen_Avrg()
        {
            return Avrg_Green;
        }
        public float GetBlue_Avrg()
        {
            return Avrg_Blue;
        }
    }
}
```

B.3 BinaryTree.cs

```
  using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
//using System.Threading.Tasks;
using System.IO;
using System.Drawing;
using System.Windows.Forms ;
namespace ImageFilteringTry
{

    public class BinaryTree
    {
        public BinaryTreeNode Root;
        public MainForm MF;
        private BinaryTreeNode CurrentIterator;
        private List<BinaryTreeNode> Nodes = new List<BinaryTreeNode>();
        public BinaryTree(Bitmap OriginalImage)
        {
            Root = new BinaryTreeNode();
            //Root.image = (Bitmap)OriginalImage.Clone();
            MF = (MainForm)Application.OpenForms[0];
            Root.imageName = "Original";
            Root.ImageSize = OriginalImage.Size;
            RGBPixel RGB = new RGBPixel(OriginalImage);
            Root.RedPix = RGB.GetRedPixel();
            Root.GreenPix = RGB.GetGreenPixel();
            Root.BluePix = RGB.GetBluePixel();
            Nodes.Clear();
        }
        public int height(BinaryTreeNode t)
        {
            //int h = 1;
            if (t == null)
                return 0;
            int heightLeft = height(t.Low);
            int heightRight = height(t.High);
            if( heightLeft > heightRight )
```

```
        return heightLeft +1;
        else
        return heightRight +1;
       /* if (t.Parent == null)
            h = 1;
        else
        {
            while (t.Parent != null)
            {
                t = t.Parent;
                h++;
            }
        }
        return h; */
    }
    private void FindLeafNodes(BinaryTreeNode CurrentNode)
    {
        BinaryTreeNode leafNode = null;
        int h = height(Root);
        if (CurrentNode.Low != null)
        {
            if( CurrentNode.Low.Process != false)
                FindLeafNodes(CurrentNode.Low);

        }
        if (CurrentNode.High != null)
        {
            if ((height(Root) % 2 != 1) &&
                CurrentNode.High.Process != false)
            {
                FindLeafNodes(CurrentNode.High);
            }
            else if (CurrentNode.High.Process == true)
            {
```

```
                CurrentNode.High.Process = false;
                if (CurrentNode.High.Low != null)
                    CurrentNode.High.Low.Process = false;
                if (CurrentNode.High.High != null)
                    CurrentNode.High.High.Process = false;
            }
        }
        if (CurrentNode.Low == null && CurrentNode.High == null)
        {
            //CurrentNode.Process = true;
            leafNode = CurrentNode;
            Nodes.Add(leafNode);
        }

    }
    public void Pre_add()
    {
        Nodes.Clear();
        BinaryTreeNode ImageNodeTobeProcessed;
        FindLeafNodes(Root);
        for (int i = 0; i < Nodes.Count ; i++)
        {
            ImageNodeTobeProcessed = Nodes[i];
            CurrentIterator = Nodes[i];
            if (ImageNodeTobeProcessed != null)
            {
              BinaryTreeNode imageProcessedLow =
               ImageProcess.LowComponentProcess(ImageNodeTobeProcessed);
              BinaryTreeNode imageProcessedHigh =
               ImageProcess.HighComponentProcess(ImageNodeTobeProcessed);
                Add(imageProcessedLow,
                    imageProcessedHigh,
                    CurrentIterator);
            }
```

```
            }
        }
        public void Add(BinaryTreeNode imageProcessedLow,
            BinaryTreeNode imageProcessedHigh, BinaryTreeNode Iterator)
        {
            BinaryTreeNode Lchild = new BinaryTreeNode();
             BinaryTreeNode Rchild = new BinaryTreeNode();
            Lchild = imageProcessedLow;
            //Lchild.imageName = bit_name;
            Rchild = imageProcessedHigh;
            if (Lchild.imageName == null)
                Lchild.imageName = Iterator.imageName + "Low";
            if (Rchild.imageName == null)
                Rchild.imageName = Iterator.imageName + "High";

            // Start from the root of the tree
            //BinaryTreeNode Iterator = Root;
            //while (true)
            {
                if (Iterator.Low == null && Iterator.High == null)
                {
                    Iterator.Low = Lchild;
                    Iterator.High = Rchild;
                    Iterator.Low.Parent = Iterator;
                    Iterator.High.Parent = Iterator;
                    //break;
                }
            }// end of while
        }
    }
}
```

B.4 BinaryTreeNode.cs

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
//using System.Threading.Tasks;
using System.IO;
using System.Drawing;
using System.Windows.Forms;
namespace ImageFilteringTry
{
    public class BinaryTreeNode
    {

            public int[][] RedPix;
            public int[][] GreenPix;
            public int[][] BluePix;
        public Size ImageSize;
        // Left is low componenet of image
        public BinaryTreeNode Low;
        // Right is high componenet of image
        public BinaryTreeNode High;
        public BinaryTreeNode Parent;
        public string imageName;
        //public Bitmap image;
        public bool Process;
        public BinaryTreeNode()
        {
            //ImageSize = null;
            RedPix = null;
            GreenPix = null;
            BluePix = null;
            Low = null;
            High = null;
            Parent = null;
            //image = null;
```

```
            Process = true;
        }
    }
}
```

B.5 FilterRGB.cs

```
  using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
//using System.Threading.Tasks;
using System.IO;
using System.Drawing;
using System.Windows.Forms ;
namespace ImageFilteringTry
{
    class FilterRGB
    {
        public int[] nPixelRGB;
        private RGBPixel RGB;
        private const int nPixelR = 0;
        private const int nPixelG = 1;
        private const int nPixelB = 2;
        private const int MAXRGB = 255;
        public Size size;
        public FilterRGB()
        {
           // size = new Size();
        }
        public FilterRGB(Bitmap bmp)
        {
            //int x, y, i;
            //Color[][] clr = null;
            nPixelRGB = new int[3] { 0, 0, 0 };
```

```
        size = new Size();
        RGB = new RGBPixel(bmp);
        size = bmp.Size;

    }//end of  public FilterRGB(Bitmap bmp)

    public int GetMaxPixelValue(int[][] arr)
    {
        int n_max = 0;
        for (int i = 0; i < arr.Length; i++)
        {
            int Max = arr[i].Max();
            if (Max > n_max)
                n_max = Max;
        }
        return n_max;
    }// end of int GetMaxPixelValue(int[][] arr)
    public int GetMinPixelValue(int[][] arr)
    {
        int n_min = 255;
        for (int i = 0; i < arr.Length; i++)
        {
            int Min = arr[i].Min();
            if (Min < n_min)
                n_min = Min;
        }
        return n_min;
    }
    public BinaryTreeNode GetHighComponent(BinaryTreeNode CurrentImage,
        char C_ColorSelected, int ProcessCount)
    {
        BinaryTreeNode HighBmp = new BinaryTreeNode();
        size.Height = CurrentImage.ImageSize.Height;
        size.Width = CurrentImage.ImageSize.Width;
```

```
        //SetRGBArrayPixel(CurrentImage);
        switch (C_ColorSelected)
        {
            case 'R':
                HighBmp = ProcessHighComp(RGB.GetRedPixel(),
                    nPixelR,
                    ProcessCount);
                break;
            case 'G':
                HighBmp = ProcessHighComp(RGB.GetGreenPixel(),
                    nPixelG,
                    ProcessCount);
                break;
            case 'B':
                HighBmp = ProcessHighComp(RGB.GetBluePixel(),
                    nPixelB,
                    ProcessCount);
                break;
        }
        return HighBmp;
    }// end of public Bitmap GetHighComponent

    // Get Red high component of Image
    private BinaryTreeNode ProcessHighComp(int[][] pixel,
        int nPosition, int ProcessCount)
    {
        BinaryTreeNode finalImage = new BinaryTreeNode();
        if (ProcessCount % 2 == 0)
        {
            finalImage = PerformHighVerticalPass(pixel, nPosition);
        }
        else
        {
            finalImage = PerformHighHorizontalPass(pixel, nPosition);
```

```
    }
    return finalImage;
}
BinaryTreeNode PerformHighHorizontalPass(int[][] pixel,
    int nPosition)
{
    BinaryTreeNode finalImage = new BinaryTreeNode();
    int [][] HighPixels = new int [size.Width/2][];
    //finalImage = new Bitmap((size.Width/2),
    //size.Height,
    //System.Drawing.Imaging.PixelFormat.Format24bppRgb);
    for (int i = 0; i < size.Width/2; i++)
    {
        HighPixels[i] = new int[size.Height];
    }
    for (int i = 0; i < size.Height; i++)
    {
        int k = 0;
        for (int j = 0; (j + 1) < size.Width; j = j + 2)
        {
            HighPixels[k][i] = (pixel[j][i] - pixel[j + 1][i]);

            //finalImage.SetPixel(k, i,
            //Color.FromArgb((byte)nPixelRGB[nPixelR],
            //(byte)nPixelRGB[nPixelG],
            //(byte)nPixelRGB[nPixelB]));
            k++;
        }
    }
     Size ArraySize = new Size();
    ArraySize.Height = size.Height;
    ArraySize.Width = size.Width / 2;
    finalImage = SetBitmapImage(finalImage,
        nPosition,
```

```
            HighPixels,
            ArraySize);

        return finalImage;
    }// end of private Bitmap ProcessHighComp
    BinaryTreeNode PerformHighVerticalPass(int[][] pixel, int nPosition)
    {
        BinaryTreeNode finalImage = new BinaryTreeNode();
        int[][] HighPixels = new int[size.Width][];
        //finalImage = new Bitmap((size.Width), (size.Height/2),
        //System.Drawing.Imaging.PixelFormat.Format24bppRgb);
        for (int i = 0; i < size.Width; i++)
        {
            HighPixels[i] = new int[size.Height/2];
        }
        for (int i = 0; i < size.Width; i++)
        {
            int k = 0;
            for (int j = 0; (j + 1) < size.Height; j = j + 2)
            {
                HighPixels[i][k] = (pixel[i][j] - pixel[i][j+1]);
                //finalImage.SetPixel(k, i,
                //Color.FromArgb((byte)nPixelRGB[nPixelR],
                //(byte)nPixelRGB[nPixelG], (byte)nPixelRGB[nPixelB]));
                k++;
            }
        }
         Size ArraySize = new Size();
         ArraySize.Height = size.Height / 2;
        ArraySize.Width = size.Width;
        //its a raw image
        finalImage = SetBitmapImage(finalImage,
            nPosition,
            HighPixels,
```

```
            ArraySize);
        return finalImage;
    }// end of private Bitmap PerformHighVerticalPass
    BinaryTreeNode SetBitmapImage(BinaryTreeNode finalImage,
        int nPosition, int[][] ProcessedPixels,Size ArraySize)
    {
        int width = ArraySize.Width;
        int height = ArraySize.Height;
       // set other values
       finalImage.RedPix = new int[width][];
       finalImage.BluePix = new int[width][];
       finalImage.GreenPix = new int[width][];
       for (int i = 0; i < width; i++)
       {
            finalImage.GreenPix[i] = new int[height];
            finalImage.BluePix[i] = new int[height];
            finalImage.RedPix[i] = new int[height];
       }//end of for
        switch (nPosition)
        {
            case 0:
                finalImage.RedPix = ProcessedPixels;
                break;
            case 1:
                finalImage.GreenPix = ProcessedPixels;
                break;
            case 2:
                finalImage.BluePix = ProcessedPixels;
                break;
        }
        finalImage.ImageSize.Width = width;
        finalImage.ImageSize.Height = height;
        return finalImage;
    }//end of SetBitmapVerticalImage
```

```
public BinaryTreeNode GetLowComponent(BinaryTreeNode CurrentImage,
    char C_ColorSelected, int ProcessCount)
{
    BinaryTreeNode LowBmp = new BinaryTreeNode();
    size.Height = CurrentImage.ImageSize.Height;
    size.Width = CurrentImage.ImageSize.Width;
    switch (C_ColorSelected)
    {
        case 'R':
            LowBmp = ProcessLowComp(CurrentImage.RedPix,
                nPixelR,
                ProcessCount);
            break;
        case 'G':
            LowBmp = ProcessLowComp(CurrentImage.GreenPix,
                nPixelG,
                ProcessCount);
            break;
        case 'B':
            LowBmp = ProcessLowComp(CurrentImage.BluePix,
                nPixelB,
                ProcessCount);
            break;
    }
    return LowBmp;
}// end of public Bitmap GetHighComponent
// Get Red high component of Image
private BinaryTreeNode ProcessLowComp(int[][] pixel,
    int nPosition, int ProcessCount)
{
    BinaryTreeNode finalImage = new BinaryTreeNode();
    if (ProcessCount % 2 == 0)
    {
        // divide image vertically
```

```
            finalImage = PerformLowVerticalPass(pixel, nPosition);
        }
        else
        {
            // divide image horizontally
            finalImage = PerformLowHorizontalPass(pixel, nPosition);
        }
        return finalImage;
    }
    // Processing vertical Low compoenent
    private BinaryTreeNode PerformLowHorizontalPass(int[][] pixel,
        int nPosition)
    {
        BinaryTreeNode finalImage = new BinaryTreeNode();
        //finalImage = new Bitmap((size.Width / 2),
        //size.Height,
        //System.Drawing.Imaging.PixelFormat.Format24bppRgb);
        int[][] LowPixels = new int[size.Width / 2][];
        for (int i = 0; i < (size.Width / 2); i++)
        {
            LowPixels[i] = new int[size.Height];
        }
        for (int i = 0; i < size.Height; i++)
        {
            int k = 0;
            for (int j = 0; (j + 1) < size.Width; j = j + 2)
            {
                LowPixels[k][i] = pixel[j][i] + pixel[j + 1][i];
                //finalImage.SetPixel(k, i,
                //Color.FromArgb((byte)nPixelRGB[nPixelR],
                //(byte)nPixelRGB[nPixelG], (byte)nPixelRGB[nPixelB]));
                k++;
            }
        }
```

```
        Size ArraySize = new Size();
        ArraySize.Height = size.Height;
        ArraySize.Width = size.Width / 2;

        // its a raw image
        finalImage = SetBitmapImage(finalImage, nPosition,
            LowPixels, ArraySize);
        return finalImage;
    }// end of private Bitmap ProcessHighComp
    // Processing vertical Low compoenent
    private BinaryTreeNode PerformLowVerticalPass(int[][] pixel,
        int nPosition)
    {
        BinaryTreeNode finalImage = new BinaryTreeNode();
        //finalImage = new Bitmap((size.Width), (size.Height/2),
        //System.Drawing.Imaging.PixelFormat.Format24bppRgb);
        int[][] LowPixels = new int[size.Width][];
        for (int i = 0; i < (size.Width); i++)
        {
            LowPixels[i] = new int[size.Height/2];
        }
        for (int i = 0; i < size.Width; i++)
        {
            int k = 0;
            for (int j = 0; (j + 1) < size.Height ; j = j + 2)
            {
                LowPixels[i][k] = pixel[i][j] + pixel[i][j+1];
                //finalImage.SetPixel(k, i,
                //Color.FromArgb((byte)nPixelRGB[nPixelR],
                //(byte)nPixelRGB[nPixelG], (byte)nPixelRGB[nPixelB]));
                k++;
            }
        }
        Size ArraySize = new Size();
```

```
            ArraySize.Height = size.Height / 2;
            ArraySize.Width = size.Width;
            // its a raw image
            finalImage = SetBitmapImage(finalImage, nPosition,
                LowPixels, ArraySize);
            return finalImage;
        }// end of private Bitmap ProcessHighComp
    }
}
```

B.6 ImageProcess.cs

```
  using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Drawing;
using System.IO;
using System.Collections;
using System.Windows.Forms;
namespace ImageFilteringTry
{
    class ImageProcess
    {
        private const int nPixelR = 0;
        private const int nPixelG = 1;
        private const int nPixelB = 2;
        private const int MAXRGB = 255;
        public static FilterRGB CurrentFilter;
        public static char ColorSelected;
        public static List<BinaryTreeNode> LeafNodes_Current =
            new List<BinaryTreeNode>();
        public static List<Bitmap> LeafNodesDisplayImage_Current =
            new List<Bitmap>();
        public static int CurrentProcessCntLow, CurrentProcessCntHigh;
```

```
public static Bitmap bitmap;
public static Bitmap CurrentImage;
public void GetLeafNode(BinaryTreeNode CurrentNode)
{
    Bitmap leafNodeImage = null;
    if (CurrentNode.Low != null)
    {
        GetLeafNode(CurrentNode.Low);
    }
    if (CurrentNode.High != null)
    {
        GetLeafNode(CurrentNode.High);
    }
    if (CurrentNode.Low == null && CurrentNode.High == null)
    {
        if (CurrentNode == CurrentNode.Parent.Low)
            leafNodeImage = ConvertToDisplayImage_Low(CurrentNode);
        if (CurrentNode == CurrentNode.Parent.High)
            leafNodeImage = ConvertToDisplayImage_High(CurrentNode);
        LeafNodes_Current.Add(CurrentNode);
        // Display
        LeafNodesDisplayImage_Current.Add(leafNodeImage);
    }
}
Bitmap SetBitmapImage(Bitmap finalImage, int nPosition,
    int[][] ProcessedPixels, Size size)
{
    for (int i = 0; i < size.Width; i++)
    {
        int k = 0;
        for (int j = 0; j < size.Height; j++)
        {
            for (int m = 0; m < nPixelB + 1; m++)
            {
```

```
                    if (m == nPosition)
                        CurrentFilter.nPixelRGB[m] =
                            ProcessedPixels[i][k];
                    else
                        CurrentFilter.nPixelRGB[m] = 0;
                }
                finalImage.SetPixel(i,
                    k,
                    Color.FromArgb((byte)CurrentFilter.nPixelRGB[nPixelR],
                    (byte)CurrentFilter.nPixelRGB[nPixelG],
                    (byte)CurrentFilter.nPixelRGB[nPixelB]));
                k++;
            }
        }
        return finalImage;
    }//end of SetBitmapVerticalImage
    private Bitmap ConvertToDisplayImage_Low(BinaryTreeNode RawImage)
    {
        Bitmap DisplayImage = null;
        Size size = new Size();
        size.Width = RawImage.ImageSize.Width;
        size.Height = RawImage.ImageSize.Height;
        int[][] pixels = new int[size.Width][];
        for (int i = 0; i < size.Width; i++)
        {
            pixels[i] = new int[size.Height];
        }
        DisplayImage = new Bitmap((size.Width),
            size.Height,
            System.Drawing.Imaging.PixelFormat.Format24bppRgb);
        int nPosition = 0;
        switch (ColorSelected)
        {
            case 'R':
```

```
            CopyValueOfArray(RawImage.RedPix, ref pixels, size);
            nPosition = 0;
            break;
        case 'G':
            CopyValueOfArray(RawImage.GreenPix, ref pixels, size);
            nPosition = 1;
            break;
        case 'B':
            CopyValueOfArray(RawImage.BluePix, ref pixels, size);
            nPosition = 2;
            break;
        default:
            CopyValueOfArray(RawImage.RedPix, ref pixels, size);
            nPosition = 0;
            break;
    }
    //get max value of pixel
    int n_max = CurrentFilter.GetMaxPixelValue(pixels);
    // check if max pixel value is greater than 255
    if (n_max > MAXRGB)
    {
        for (int i = 0; i < size.Height; i++)
        {
            for (int j = 0; j < size.Width; j++)
            {
                pixels[j][i] = (pixels[j][i] * MAXRGB) / n_max;
            }
        }
    }
    else
    {
        //[check to see
        //get min value of pixel
        int n_min = CurrentFilter.GetMinPixelValue(pixels);
```

```
            int abs_min = Math.Abs(n_min);
            // check if min pixel value is less than 0
            if (n_min < 0)
            {
                for (int i = 0; i < size.Height; i++)
                {
                    for (int j = 0; j < size.Width; j++)
                    {
                        pixels[j][i] = pixels[j][i] + abs_min;
                    }
                }
            }
            //]
        }//end of else
        DisplayImage = SetBitmapImage(DisplayImage,
            nPosition,
            pixels,
            size);
        return DisplayImage;
    }
    private static void CopyValueOfArray(int[][] Source,
        ref int[][] destination,
        Size size)
    {
        for (int i = 0; i < size.Width; i++)
        {
            for (int j = 0; j < size.Height; j++)
            {
                destination[i][j] = Source[i][j];
            }
        }
    }
    private Bitmap ConvertToDisplayImage_High(BinaryTreeNode RawImage)
    {
```

```
Bitmap DisplayImage = null;
Size size = new Size();
size.Width = RawImage.ImageSize.Width;
size.Height = RawImage.ImageSize.Height;
int[][] pixels = new int[size.Width][];
for (int i = 0; i < size.Width; i++)
{
    pixels[i] = new int[size.Height];
}
DisplayImage = new Bitmap((size.Width),
    size.Height,
    System.Drawing.Imaging.PixelFormat.Format24bppRgb);
int nPosition = 0;
switch (ColorSelected)
{
    case 'R':
        CopyValueOfArray(RawImage.RedPix, ref pixels, size);
        nPosition = 0;
        break;
    case 'G':
        CopyValueOfArray(RawImage.GreenPix, ref pixels, size);
        nPosition = 1;
        break;
    case 'B':
        CopyValueOfArray(RawImage.BluePix, ref pixels, size);
        nPosition = 2;
        break;
    default:
        CopyValueOfArray(RawImage.RedPix, ref pixels, size);
        nPosition = 0;
        break;
}
//get min value of pixel
int n_min = CurrentFilter.GetMinPixelValue(pixels);
```

```
        int abs_min = Math.Abs(n_min);
        // check if min pixel value is less than 0
        if (n_min < 0)
        {
            for (int i = 0; i < size.Height; i++)
            {
                for (int j = 0; j < size.Width; j++)
                {
                    pixels[j][i] = pixels[j][i] + abs_min;
                }
            }
        }
        DisplayImage = SetBitmapImage(DisplayImage,
            nPosition,
            pixels,
            size);
        return DisplayImage;
    }
    public Bitmap DisplayAllLeafNodes()
    {
        // Bitmap bitmap =
        //new Bitmap(OriginalImage.Width, OriginalImage.Height);
        int x = 0, y = 0;
        for (int i = 0; i + 1 < LeafNodes_Current.Count; i = i + 2)
        {
            if (LeafNodes_Current[i].Process == false)
                break;
            using (Graphics g = Graphics.FromImage(bitmap))
            {
                //draw first image
                g.DrawImage(LeafNodesDisplayImage_Current[i], x, y);
                //for sencond image
                if (LeafNodesDisplayImage_Current[i].Height ==
                    LeafNodes_Current[i].Parent.ImageSize.Height)
```

```
                g.DrawImage(LeafNodesDisplayImage_Current[i + 1],
                    LeafNodesDisplayImage_Current[i].Width,
                    y);
            else
            {
                g.DrawImage(LeafNodesDisplayImage_Current[i + 1],
                    x,
                    LeafNodesDisplayImage_Current[i].Height);
                x = x + LeafNodesDisplayImage_Current[i].Width;
            }
        }
    }
    return bitmap;
}
public static BinaryTreeNode LowComponentProcess(
    BinaryTreeNode LowImage)
{
    //BinaryTreeNode LowImage = null;
    if (CurrentImage == null)
    {
        MessageBox.Show("Please Select Image.", "Hey",
            MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
    }
    else
    {
        LowImage = CurrentFilter.GetLowComponent(LowImage,
            ColorSelected,
            CurrentProcessCntLow);
    }
    //LowCompImage = LowImage;
    return LowImage;
}
public static BinaryTreeNode HighComponentProcess(
    BinaryTreeNode HighImage)
```

```
{
    //Bitmap HighImage = null;
    if (CurrentImage == null)
    {
        MessageBox.Show("Please Select Image.", "Hey",
            MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
    }
    else
    {
        HighImage = CurrentFilter.GetHighComponent(HighImage,
            ColorSelected,
            CurrentProcessCntHigh);
    }
    // HighCompImage = HighImage;
    return HighImage;
}
// Image resize
public static Image resizeImage(Image imgToResize, Size size)
{
    int sourceWidth = imgToResize.Width;
    int sourceHeight = imgToResize.Height;
    float nPercent = 0;
    float nPercentW = 0;
    float nPercentH = 0;
    nPercentW = ((float)size.Width / (float)sourceWidth);
    nPercentH = ((float)size.Height / (float)sourceHeight);
    if (nPercentH < nPercentW)
        nPercent = nPercentH;
    else
        nPercent = nPercentW;
    int destWidth = (int)(sourceWidth * nPercent);
    int destHeight = (int)(sourceHeight * nPercent);
    Bitmap b = new Bitmap(destWidth, destHeight);
    Graphics g = Graphics.FromImage((Image)b);
```

```
            //g.InterpolationMode = InterpolationMode.HighQualityBicubic;
            g.DrawImage(imgToResize, 0, 0, destWidth, destHeight);
            g.Dispose();
            return (Image)b;
        }
        // cropping image
        public static Image cropImage(Image img, Rectangle cropArea)
        {
            Bitmap bmpImage = new Bitmap(img);
            Bitmap bmpCrop = bmpImage.Clone(cropArea,
            bmpImage.PixelFormat);
            return (Image)(bmpCrop);
        }
    }
}
```

B.7 Variance-covariance.cs

```
  using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Drawing;
namespace ImageFilteringTry
{
    // positive definite Matrix
    class VariantCoVariant
    {
       /* public struct MatrixElements
        {
            public float Value;
            public Char Color;
        } */
        private double[][] VVMatrix;
```

```
private int Matrixsize;
public double Avrg_Red, Avrg_Green, Avrg_Blue, totalPixel;
private BinaryTreeNode TreeNode;
private const int nPixelR = 0;
private const int nPixelG = 1;
private const int nPixelB = 2;
public VariantCoVariant()
{
    Matrixsize = 3;
    VVMatrix = new double[Matrixsize][];
    for (int i = 0; i < Matrixsize; i++)
    {
        VVMatrix[i] = new double[Matrixsize];
    }
  /*  for (int i = 0; i < Matrixsize; i++)
    {
        for (int j = 0; j < Matrixsize; j++)
        {
            PDMatrix[i][j].Color = 'R';
            PDMatrix[i][j+1].Color = 'G';
            PDMatrix[i][j + 2].Color = 'B';
        }
    }*/
}
public double[][] GetPDM(BinaryTreeNode MyImage)
{
    TreeNode = MyImage;
    totalPixel = TreeNode.ImageSize.Width * TreeNode.ImageSize.Height;
    ComputeAvrgRGB();

    for (int i = 0; i < Matrixsize; i++)
    {
        for (int j = 0; j < Matrixsize; j++)
        {
```

```
                double Average_i = 0;
                double Avergae_j = 0;
                int[][] ColorValues_i = GetColor(i,ref Average_i);
                int[][] ColorValues_j = GetColor(j,ref Avergae_j);
                VVMatrix[i][j] = ComputeSigma(ColorValues_i,
                    ColorValues_j, Average_i, Avergae_j);

            }
        }
        double CorelationRB = ComputeCorelation(nPixelR, nPixelB);
        double CorelationRG = ComputeCorelation(nPixelR, nPixelG);
        return VVMatrix;
    }
    private int[][] GetColor(int Color, ref double average)
    {
        switch (Color)
        {
            case nPixelR :
                average = Avrg_Red;
                return TreeNode.RedPix;
                //break;
            case nPixelG:
                average = Avrg_Green;
                return TreeNode.GreenPix;
            case nPixelB:
                average = Avrg_Blue;
                return TreeNode.BluePix;
            default :
                average = Avrg_Red;
                return TreeNode.RedPix;
        }
    }
    private double ComputeSigma(int[][] Color_i, int[][] Color_j, double//
        Average_i, double Average_j)
```

```
{
    double Sigma =0;
    double check1 = 0, check2 = 0;
    double temp1 = 0, temp2 = 0;
    int cnt =0;
    for (int i = 0; i < TreeNode.ImageSize.Height; i++)
    {
        for (int j = 0; j < TreeNode.ImageSize.Width; j++)
        {
            temp1 = Color_i[j][i] - Average_i;
            temp2 = Color_j[j][i] - Average_j;
            if (Color_j[j][i] < Average_j)
                cnt++;
            check1 = check1 + temp1;
            check2 = check2 + temp2;
            Sigma = Sigma + (temp1 * temp2);
        }
    }
    int check3 = (int)check2;
    return (Sigma / totalPixel -1);
}
// Computer average of Red , blue and green
private void ComputeAvrgRGB()
{
    Avrg_Blue = Avrg_Green = Avrg_Red = 0;
    for (int i = 0; i < TreeNode.ImageSize.Height; i++)
    {
        for (int j = 0; j < TreeNode.ImageSize.Width; j++)
        {
            Avrg_Red = Avrg_Red + TreeNode.RedPix[j][i];
            Avrg_Green = Avrg_Green + TreeNode.GreenPix[j][i];
            Avrg_Blue = Avrg_Blue + TreeNode.BluePix[j][i];
        }
    }
```

```
            Avrg_Red = Avrg_Red / totalPixel;
            Avrg_Green = Avrg_Green / totalPixel;
            Avrg_Blue = Avrg_Blue / totalPixel;
        }
        public double ComputeCorelation(int Color1, int Color2)
        {
            double Corelation = 0;
            double Color1Covariant = Math.Sqrt(VVMatrix[Color1][Color1]);
            double Color2Covariant = Math.Sqrt(VVMatrix[Color2][Color2]);
            Corelation = VVMatrix[Color1][Color2] /
                (Color1Covariant * Color2Covariant);

            return Corelation;
        }
        public double GetStandardDeviation(int Color1)
        {
            return (Math.Sqrt(VVMatrix[Color1][Color1]) );
        }
    }
}
```

B.8 Histogram.cs

```
  using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
//using System.Threading.Tasks;
using System.IO;
using System.Drawing;
using System.Windows.Forms ;
using System.Collections;
namespace ImageFilteringTry
{
    class Histogram
```

```
{

    private int[] m_All; // histogram of the grayscale
    private int[] m_Red;
    private int[] m_Green;
    private int[] m_Blue;
    public enum ColorType { MIX_COLORS = -1, ALL_COLOR = 0, RED_COLOR, GREEN_COLO
    private static Color[] histogramColor = { Color.Gray, Color.Red, Color.Lime,

    /// <summary>
    /// Total number of gray pixels.
    /// </summary>
    public int TotalAll { get; private set; }
    /// <summary>
    /// Total number of pixels containing red.
    /// </summary>
    public int TotalRed { get; private set; }
    /// <summary>
    /// Total number of pixels containing green.
    /// </summary>
    public int TotalGreen { get; private set; }
    /// <summary>
    /// Total number of pixels containing blue.
    /// </summary>
    public int TotalBlue { get; private set; }
    /// <summary>
    /// Total number of pixels analyzed.
    /// </summary>
    public int TotalPixels { get; private set; }
    /// <summary>
    /// Index of the maximum element of the grayscale histogram data array.
    /// </summary>
    public int MaxAllIndex { get; private set; }
    /// <summary>
```

```
/// Index of the maximum element of the red histogram data array.
/// </summary>
public int MaxRedIndex { get; private set; }
/// <summary>
/// Index of the maximum element of the green histogram data array.
/// </summary>
public int MaxGreenIndex { get; private set; }
/// <summary>
/// Index of the maximum element of the blue histogram data array.
/// </summary>
public int MaxBlueIndex { get; private set; }
/// <summary>
/// Returns the color for a histogram based on its type.
/// </summary>
/// <param name="type">
/// Type of histogram. This is one of the constants:
/// ALL_HISTOGRAM, RED_HISTOGRAM, GREEN_HISTOGRAM, BLUE_HISTOGRAM.
/// </param>
/// <returns>
/// The color for the specified histogram.
/// </returns>
public static Color HistogramColor(ColorType type)
{
    return histogramColor[(int)type];
}
/// <summary>
/// Creates the histograms for the specified bitmap.
/// </summary>
/// <param name="bmp">
/// Bitmap to analyze.
/// </param>
public Histogram(Bitmap bmp)
{
    int x, y, i;
```

```
        Color[][] clr = null;
        Size size = new Size();
        m_All = new int[256];
        m_Red = new int[256];
        m_Green = new int[256];
        m_Blue = new int[256];
        // generate the Histogram data
        if (bmp != null)
        {
            size = bmp.Size;
            TotalPixels = size.Width * size.Height;
            clr = new Color[size.Width][];
            for (i = 0; i < size.Width; i++)
                clr[i] = new Color[size.Height];
            for (x = 0; x < size.Width; x++)
                for (y = 0; y < size.Height; y++)
                {
                    clr[x][y] = bmp.GetPixel(x, y);
                    m_Red[clr[x][y].R]++;
                    m_Green[clr[x][y].G]++;
                    m_Blue[clr[x][y].B]++;
                    m_All[(clr[x][y].R + clr[x][y].G + clr[x][y].B) / 3]++;
                } // end for
        } // end if
        initMaxAndTotal();
    }
    private void initMaxAndTotal()
    {
        // REQUIRES contructor
        // determine the max levels of each color
        MaxAllIndex = 0;
        MaxRedIndex = 0;
        MaxGreenIndex = 0;
        MaxBlueIndex = 0;
```

```
        TotalAll = m_All[0];
        TotalRed = m_Red[0];
        TotalGreen = m_Green[0];
        TotalBlue = m_Blue[0];
        for (int i = 1; i < m_All.GetLength(0); i++)
        {
            TotalAll += m_All[i];
            TotalRed += m_Red[i];
            TotalGreen += m_Green[i];
            TotalBlue += m_Blue[i];
            if (m_All[MaxAllIndex] < m_All[i])
                MaxAllIndex = i;
            if (m_Red[MaxRedIndex] < m_Red[i])
                MaxRedIndex = i;
            if (m_Green[MaxGreenIndex] < m_Green[i])
                MaxGreenIndex = i;
            if (m_Blue[MaxBlueIndex] < m_Blue[i])
                MaxBlueIndex = i;
        } // end for
    }
    /// <summary>
    /// Retrieves the grayscale histogram data.
    /// </summary>
    /// <param name="lpResult">
    /// Array where to put the data.
    /// </param>
    public void GetAllHistogram(int[] lpResult)
    {
        copyArray(m_All, lpResult);
    }
    /// <summary>
    /// Retrieves the red histogram data.
    /// </summary>
    /// <param name="lpResult">
```

```
/// Array where to put the data.
/// </param>
public void GetRedHistogram(int[] lpResult)
{
    copyArray(m_Red, lpResult);
}
/// <summary>
/// Retrieves the green histogram data.
/// </summary>
/// <param name="lpResult">
/// Array where to put the data.
/// </param>
public void GetGreenHistogram(int[] lpResult)
{
    copyArray(m_Green, lpResult);
}
/// <summary>
/// Retrieves the blue histogram data.
/// </summary>
/// <param name="lpResult">
/// Array where to put the data.
/// </param>
public void GetBlueHistogram(int[] lpResult)
{
    copyArray(m_Blue, lpResult);
}
/// <summary>
/// Retrieves the specified histogram data. Return true if success.
/// </summary>
/// <param name="type">
/// The histogram to retrieve. This is one of the constants:
/// ALL_COLOR, RED_COLOR, GREEN_COLOR, BLUE_COLOR.
/// </param>
/// <param name="lpResult">
```

```
/// Array where to put the data.
/// </param>
/// <returns>
/// true - success: lpResult contains the data.
/// false - error: contents of lpResult are unknown.
/// </returns>
public bool GetHistogram(ColorType type, int[] lpResult)
{
    bool retval = true;
    switch (type)
    {
        case ColorType.ALL_COLOR:
            GetAllHistogram(lpResult);
            break;
        case ColorType.RED_COLOR:
            GetRedHistogram(lpResult);
            break;
        case ColorType.GREEN_COLOR:
            GetGreenHistogram(lpResult);
            break;
        case ColorType.BLUE_COLOR:
            GetBlueHistogram(lpResult);
            break;
        default:
            retval = false;
            break;
    } // end switch
    return retval;
}
/// <summary>
/// Returns the index of the maximum element of the specified histogram data
/// </summary>
/// <param name="type">
/// The histogram to retrieve. This is one of the constants:
```

```
/// ALL_HISTOGRAM, RED_HISTOGRAM, GREEN_HISTOGRAM, BLUE_HISTOGRAM.
/// </param>
/// <returns>
/// The index of the maximum element of the specified histogram data array, o
/// </returns>
public int GetMaxIndex(ColorType type)
{
    int retval = -1;
    switch (type)
    {
        case ColorType.ALL_COLOR:
            retval = MaxAllIndex;
            break;
        case ColorType.RED_COLOR:
            retval = MaxRedIndex;
            break;
        case ColorType.GREEN_COLOR:
            retval = MaxGreenIndex;
            break;
        case ColorType.BLUE_COLOR:
            retval = MaxBlueIndex;
            break;
    } // end switch
    return retval;
}
private void copyArray(int[] lpFrom, int[] lpTo)
{
    // copy the array "from" into "to"
    int iSize;
    if (lpFrom != null && lpTo != null)
    {
        iSize = Math.Min(lpFrom.GetLength(0), lpTo.GetLength(0));
        for (int i = 0; i < iSize; i++)
            lpTo[i] = lpFrom[i];
```

```
            } // end if
        }
        private void copyArray(float[] lpFrom, float[] lpTo)
        {
            // copy the array "from" into "to"
            int iSize;
            if (lpFrom != null && lpTo != null)
            {
                iSize = Math.Min(lpFrom.GetLength(0), lpTo.GetLength(0));
                for (int i = 0; i < iSize; i++)
                    lpTo[i] = lpFrom[i];
            } // end if
        }
    }
}
```

References

[1] R. M. GOODY and Y. L. YUNG Atmospheric Radiation. *Oxford University Press*, 2:558–565, 1989.

[2] K.Dogras, P.Ioannidou, P.Chrissoulidis Analytical study of the changes in the color of daylight due to sulfate droplets and soot grains in the atmosphere. *Journal of Quantitative Spectroscopy and Radiative Transfer*, 84:223–238, 2004.

[3] C. Bohren and A. Fraser Colors of the Sky. 1985.

[4] Nicolas Hautire, Raouf Babari, ric Dumont, Roland Brmond, and Nicolas Paparoditis Estimating Meteorological Visibility using Cameras: a Probabilistic Model-Driven Approach.

[5] Nicole Pauly Hyslop Impaired visibility: the air pollution people see. *Atmospheric Environment journal homepage: www.elsevier.com/locate/atmosenv*, 43:182–195, 2009.

[6] Tsay, G. Stephens and T. Greenwald An Investigation of Aerosol Microstructure on visual air quality. *Atmospheric Environment*, 25a(5/6):1039-1053,1991.

[7] P. E. Haralabidis and Christodoulos Pilinis Skylight Color Shifts due to Variations of Urban-Industrial Aerosol Properties: Observer Color Difference Sensitivity Compared to a Digital Camera. *Aerosol Science and Technology*, 2008.

[8] Baumer, D., Versick S. and Vogel, B. Determination of the Visibility Using a Digital Panorama Camera. *Atmos. Environ. Doi:10.1016/J.Atmosenv.2007.06.24.*, 2007.

[9] Lo‘pez-A lvarez, M. A., Hernandez-Andre‘s, J., Romero J., and Lee, R. L., Jr. Designing a Practical System for Spectral Imaging of Skylight. *Appl. Opt.*, 44(27):5688-5695, 2005.

[10] Bre on, F.-M. CLIMATE: how do aerosols affect cloudiness and climate?. *doi:10.1126/science.1131668*, 313:623-624, 2006.

[11] Kokkola, H., Romakkaniemi, S., Laaksonen, A. On the formation of radiation fogs under heavily polluted conditions. *Atmospheric Chemistry and Physics Discussions*, 3:389-411, 2003.

[12] Malm, W.C. Introduction to Visibility. *Cooperative Institute for Research in the Atmosphere, Fort Collins, CO.*, 1: 27-40, 1999.

[13] Malm,W., MacFarland, K.K., Molenar, J., Daniel, T. Introduction to Visibility. *Managing Air Quality and Scenic Resources at National Parks and Wilderness Areas. Westview Press, Boulder, CO*, 1:27-40, 1983.

[14] Frdric Patin An Introduction to Digital Image Processing. *Atmospheric Environment*, , 2003.

[15] Robi Polikar, The Wavelet Tutorial, http://engineering.rowan.edu/ polikar/WAVELETS/WT-tutorial.html.

[16] Eric J. Stollnitz ,Tony D. DeRose, David H. Salesin Wavelets for Computer Graphics: A Primer Part 1 University of Washingtonl.

[17] Eric J. Stollnitz ,Tony D. DeRose, David H. Salesin Wavelets for Computer Graphics: A Primer Part 2 University of Washingtonl.

[18] http://iwritearticle.com/environmental-problems-in-ahmadabad/

[19] http://www.stagespot.com/colorchoice.html

[20] http://www.gamedev.net/page/resources/technical/graphics-programming-and-theory/an-introduction-to-digital-image-processing-r2007

[21] http://www.severewx.com/Radiation/scattering.html

Vita

Graduate College
University of Nevada, Las Vegas

Amrita N. Amritphale

Degrees:

Bachelor of Engineering in Information Technology Science 2008
University of Pune, India

Thesis Title: A Digital Image Processing Method for Detecting Pollution in the Atmosphere from Camera Video

Thesis Examination Committee:

Chairperson, Dr. Evangelos Yfantis, Ph.D.
Committee Member,Dr. John Minor, Ph.D.
Committee Member, Dr. Hal Berghel, Ph.D.
Graduate Faculty Representative, Dr. Jacimaria R. Batista, Ph.D.