Comparison of Square and Hexagonal Pixels for Image Processing and Alternative Solutions to Linear Equations for Computed Tomography

By
Girish G. Tirunelveli

A Thesis submitted to the Faculty of Graduate Studies
in partial fulfillment of the requirements for the degree of

Master of Science
in Electrical and Computer Engineering

Department of Electrical and Computer Engineering
University of Manitoba
Winnipeg, Manitoba R3T 5V6

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

COMPARISON OF SQUARE AND HEXAGONAL PIXELS FOR IMAGE PROCESSING AND ALTERNATIVE SOLUTIONS TO LINEAR EQUATIONS FOR COMPUTED TOMOGRAPHY

BY

GIRISH G. TIRUNELVELI

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree of

Master of Science

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my deep sense of gratitude to Dr. Richard Gordon, without whose help not a single byte of this work would have been possible. His flexibility and ability to patiently repeat things for everyone's understanding is remarkable. I would have asked him about "Boxcar" function at least 5 times without realizing that I have asked him before. He would answer me patiently every time. With Dr. Gordon as my advisor, I had 30 years of knowledge and a zillion contacts at my disposal.

I am grateful to Dr. Stephen Pistorius who was a great source of inspiration for this thesis. The brainstorming sessions we had in his office every month contributed significantly to this project. His attention to detail and result-orientedness helped me keep the thesis on track. I hope to cultivate his inquisitiveness and "for-science" attitude during my career years.

I would like to thank Dr. Brian Hennen, Dean of Medicine, and the Organ Imaging Fund of the Department of Radiology, the University of Manitoba, for grant support. I would like to thank Dr. Miroslaw Pawlak for his critical comments.

I would call Cam Melvin and Jorge Edmundo Alpuche-Aviles whenever I had problems with my code. Just talking to them has helped me overcome a few obstacles. I would like to thank Eric Van Uytven for showing great interest in my work and providing tips

regarding the submission and presentation of the thesis. Thanks to Silvia Cenzano who volunteered in helping me whenever I needed it.

A big "Thank-You" goes to my mom and dad. They had come to Canada to have good time, but mom ended up cooking for me and dad ended up cleaning my apartment, organizing my paper affairs and proof-reading this thesis. Their sacrifice is truly magnificent.

ABSTRACT

In most applications of image processing data is collected and displayed in square pixels. Hexagonal pixels offer the advantage of greater rotational symmetry in addition to close packed structure without gaps and a more circular (isotropic) pixel. I compared the image quality of images using square pixels with that of images employing hexagonal pixels. The comparison was done using various images, each considering a different aspect of geometry (i.e., lines at different angles, curves, etc.). The square pixel images were constructed using the average of a square area of smaller square pixels. Hexagonal pixel images were constructed using two techniques. The first one was called the "two-template approach", wherein two different templates were used to create a close packed hexagonal image from smaller square pixels. The second approach was called the "six-neighbor approach" which creates a rectangular template using the six neighbors of a hexagonal pixel. Euclidean distance, Resemblity, Entropy and MTF were the image quality measures used to compare the square pixel and hexagonal pixel images. Based on the results obtained using the image quality measures employed, I conclude that contrary to my intuition and their widespread use in nature (retinas and ommatidia), hexagonal pixels do not appear to offer any significant advantage over conventional square pixels.

In case of reconstructions from highly underdetermined equations using iterative Algebraic Reconstruction Techniques (*ART including AART, MART, SIRT and SART), the reconstruction quality is dependent on many factors. In this thesis I implemented code for *ART and studied the effect of various seed (starting) images, projection angle ordering schemes and pixel weighting schemes on reconstruction quality. I used Euclidean distance for quantitative comparison of the reconstruction

quality. Based on the experiments performed I arrived at four results: i) the Euclidean distance measure is least (best) for SART, followed by ART, SIRT and MART; ii) the Euclidean distance measure is best for the seed image generated using Fourier backprojection technique, followed by the Matlab™ meshgrid and flat (constant) seeds; iii) of the various projection angle ordering schemes used for the experiments, the WDAS (Weighted Distance Access Scheme) gave the lowest Euclidean distance followed by the MLSAS (Multilevel resolution Select Access Scheme), FAAS (Fixed Angle Access Scheme), RAS (Random Access Scheme) and SAS (Sequential Access Scheme); and iv) of the different pixel weighting schemes used, the CONT (contribution made by pixel on adjacent ray) scheme, introduced here, gave the least Euclidean distance measure followed by BIN (Binary scheme), DIST (distance of center of pixel from center of ray) and INT (length of center of ray within a pixel) weighting scheme. These optimizations should help in the search for a computed tomography algorithm that yields the best image quality per x-ray dose used.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

1-D	One-Dimensional
2-D	Two-Dimensional
3-D	Three-Dimensional
ART	Algebraic Reconstruction Technique
BIN	Binary Pixel Weighting Scheme
CBP	Convoluted Backprojection (same as Fourier backprojection)
CFBP	Convoluted Filtered Backprojection (same as Fourier Filtered Backprojection)
CONT	Contribution made by pixel on adjacent rays weighting scheme
CT	Computed Tomography
DIST	Distance of center of ray from center of pixel weighting scheme
FAAS	Fixed Angle Access Scheme
FT	Fourier Transform
INT	Length of ray within pixel weighting scheme
MART	Multiplicative Algebraic Reconstruction Technique
MLSAS	Multi Level resolution Select Access Scheme
MTF	Modulation Transfer Function
PS	Power Spectrum
PSF	Point Spread Function
RAS	Random Access Scheme
SAS	Sequential Access Scheme
SNR	Signal to Noise Ratio
SIRT	Simultaneous Iterative Reconstruction Technique
SART	Simultaneous Algebraic Reconstruction Technique
WDAS	Weighted Distance Access Scheme

Chapter 1

Introduction, Motivation and Objective of the Thesis

Man, the crowning glory of creation has come a long way from the days where his only aim was survival. He has progressed steadily from the days of cave dwelling and coarse creature to the present refined sophisticated being. Of these periods none has been as stunning as the present era. One area where man has advanced leaps and bounds is medical science. One prominent fact that has underlined these developments is that it has been a collective achievement; no single profession (clinicians, radiologists or engineers) can claim the entire credit. This has been made possible with the collaboration of knowledge, ideas and ability. No other example would fit the bill as perfectly as the advancement in Computed Tomography.

1.1 What is Computed Tomography?

Computed Tomography (CT) is an imaging technique that has revolutionized the field of medical diagnostics. Computed Tomography is based on the x-ray principle: as x-rays pass through the body they are absorbed or attenuated (weakened) at differing levels creating a matrix or profile of x-ray beams of different strength. This x-ray profile is registered on film, thus creating an image. Each profile is then backwards reconstructed (i.e. back-projected) by a dedicated computer into a two-dimensional image of the slice that was scanned. This is shown in Figure 1-1. CT is used in many areas such as nondestructive evaluation of industrial and biological specimens, radio astronomy, electron microscopy, optical interferometry, X-ray crystallography, petroleum

engineering and geophysical exploration. Indirectly, it has also led to new developments in its predecessor techniques in radiographic imaging.

CT has the ability to image a combination of soft tissue, bone and blood vessels with high contrast. CT can be very useful in providing diagnostic information on several areas of the body including brain, eyes, heart, liver, kidney, woman's breast, etc.

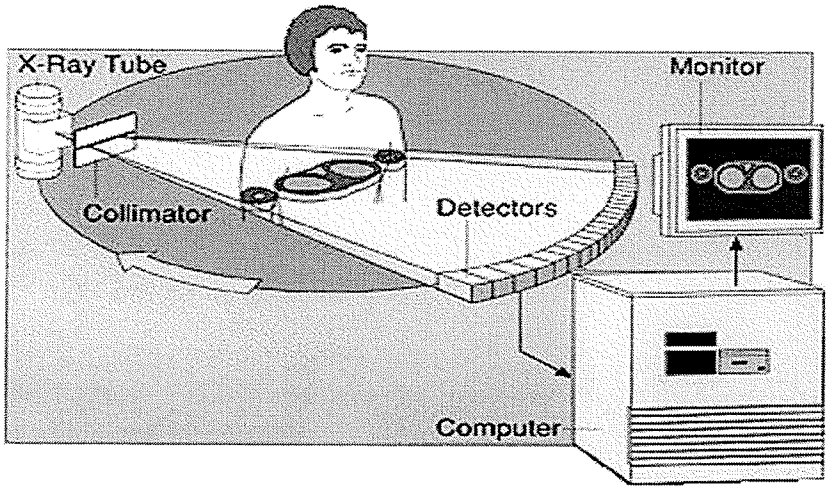


Figure 1-1. *Diagram showing relationship of x-ray tube, patient, detector, and image reconstruction computer and display monitor [24]. Copyright © 1997-2001 Imaginis Corporation All rights reserved.*

1.2 Mathematical Representation of CT

Mathematically, the main principle behind CT imaging is estimating an image (object) from its projections. In ordinary radiography, a two dimensional (2-D) shadow of a three-dimensional (3-D) body is produced on film by irradiating the body with X-ray photons as shown in Figure 1-2. But historically imaging a 3-D body is accomplished by reconstructing one 2-D section at a time through the use of one-dimensional (1-D) projections as shown in Figure 1-3. Exceptions to this are the Dynamic Spatial Reconstructor developed at the Mayo Clinic [40] [41], where a series of 2-D projection

images are obtained by irradiating the body of interest on a fluorescent screen, and single-photon emission computed tomography (SPECT), where a series of 2-D projection images are obtained using a gamma camera [45].

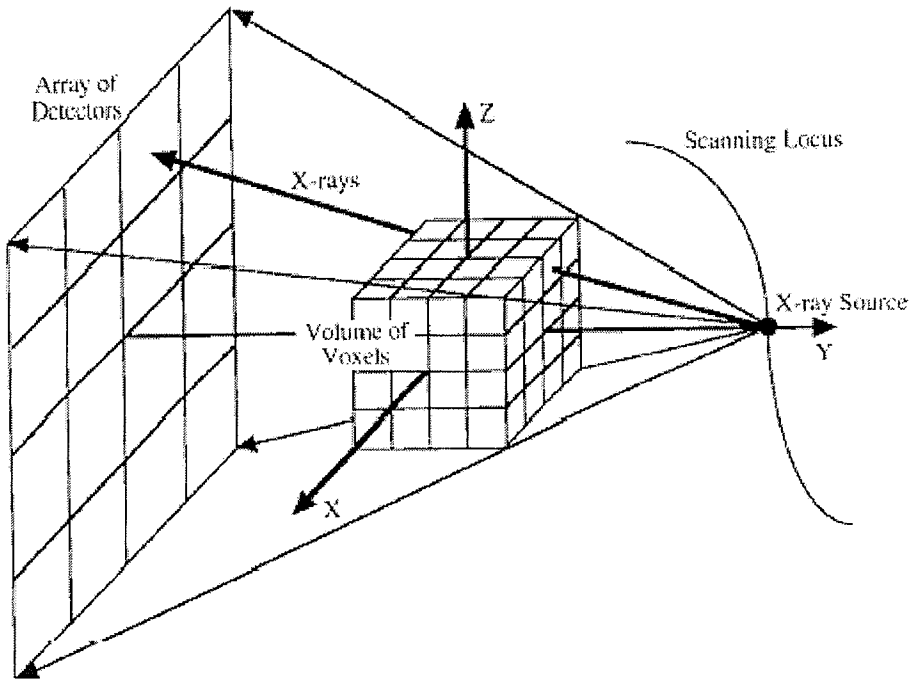


Figure 1.2. *Diagram Graphic/Mathematical model showing CT [17]. A 2-D shadow of a 3-D body is produced on film by irradiating the body with X-ray photons.*

Theoretically, every cone-beam datum is a linear integral along an X-ray path. To facilitate algorithmic implementation, the problem is cast in discrete domains. After the object and detection plate are made discrete, a continuous cone-beam projection frame is approximated as a set of values on a 2-D detection grid. Each of the values equals a sum of weighted values of those voxels that are in a neighborhood of the corresponding X-ray path.

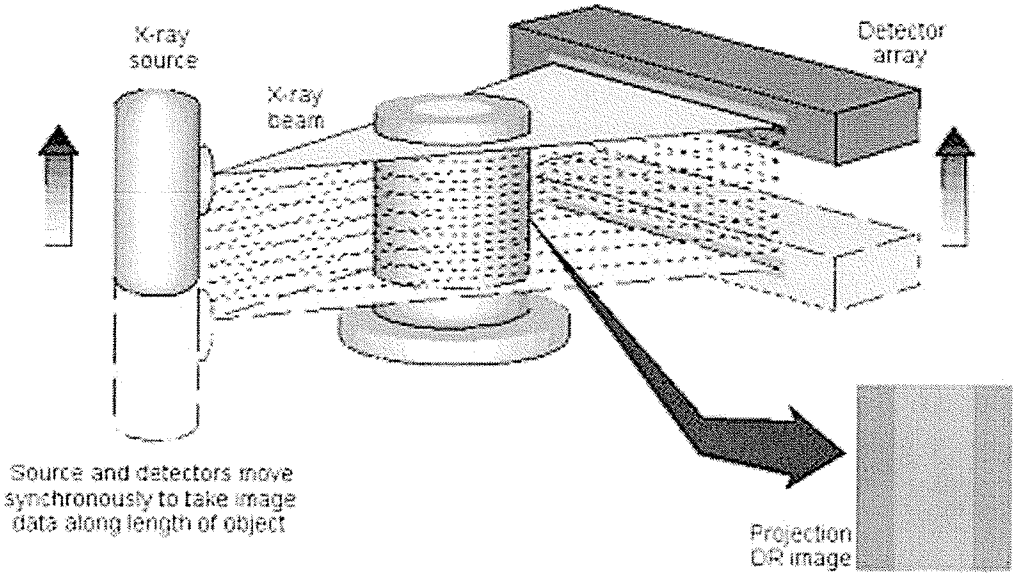


Figure 1-3. *A 3-D CT can be converted into multiple 2-D CT [24].*

Simplifying the 3-D problem of CT into 2-D, it can be expressed mathematically as below –

Given (the graphical representation is shown in Figure 1-4): -

At 0° angle, the projection values $P(\theta=0) = \{a_1, a_2, a_3, a_4, a_5\}$

At 90° angle, the projection values $P(\theta=90) = \{b_1, b_2, b_3, b_4, b_5\}$

Find $f(x,y)$.

Note that Figure 1-4 shows an x-ray source emitting parallel beams of x-rays. Different x-ray sources can be used. Some emit fan-beam x-rays. This thesis considers the CT problem with a parallel beam x-ray source. Also it deals in 2-D, since a 3-D problem can be reduced to a two-dimensional problem by recognizing that if a horizontal (planar) cross-section of the mass distribution is known at every height then all of the three-dimensional information is readily available.

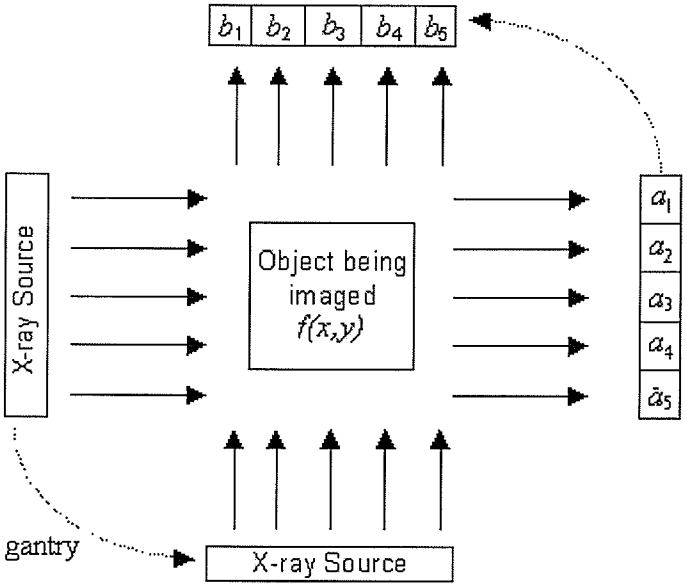


Figure 1-4. *Graphical representation of the mathematical model of forward CT. The X-ray source and detector are rotated around the object and the detector readings are recorded at each angle. These projection values are used to reconstruct back the image. Note that more the angles, the better the resolution of the reconstructed image, but higher would be the dose.*

1.3 Challenges of Computed Tomography

To obtain high quality tomograms, image reconstruction is essential. A reconstruction algorithm determines, along with the measured data, how accurately the linear attenuation coefficient can be calculated in medical x-ray CT. In clinical scanning the efficiency of the detectors is constrained both by techniques and costs. Since the patient's dose must be limited [23], the most convenient way to improve the accuracy is to optimize the reconstruction algorithm. High performance algorithms are sought to achieve reconstructions yielding more diagnostic information.

Several key factors characterize the performance of a CT reconstruction algorithm. The first and most important one is accuracy: how faithfully the precious diagnostic information can be reconstructed and presented in the tomogram. The image quality can be evaluated by different criteria, each characterizing a specific kind of information. Subjective image quality is also critical since most images are interpreted visually. Freedom from artifacts is crucial to avoid misleading diagnostic human interpretations. Another important factor is the computational speed. Fast reconstruction is always expected to reduce the diagnostic time. Other factors include how flexible the algorithm is, how easy it is to implement, etc. Improving the image quality when there is limited amount of projection data is also important in x-ray CT.

Radiation doses from CT scans are often higher than needed and may contribute to cancer later in life (according to studies published [24] in the American Journal of Roentgenology). Even though CT scans are very beneficial in detecting disease, researchers have found that many centers use the same CT settings on children as they do on adults, possibly exposing children to radiation levels approximately five times [24] higher than necessary to obtain a quality image. CT scans account for approximately 4% of medical imaging exams, however, research shows that CT scans contribute to 40% of the total amount of radiation received from diagnostic tests. In a study, conducted by Lane Donnelly, MD, a radiologist from the Children's Medical Center in Cincinnati, and his colleagues, the researchers found that approximately 600,000 abdominal and head CT examinations are annually performed in children under the age of 15 years, a rough estimate is that 500 of these individuals might ultimately die from cancer attributable to the CT radiation [24]. The number of CT scans performed in recent years has also risen

dramatically, further creating the need for minimizing radiation exposure during the test. In most cases, the benefits of finding disease with a CT scan outweigh the risks of X-Ray radiation exposure and/or injections of imaging contrast and use of sedatives during the scan. However, there is an important need to reduce the dose a patient receives in CT. From the forward-CT standpoint, the more the projection angles the higher is the dose. From CT image reconstruction standpoint, one cannot always get away with less projection data, since it will reduce the quality of the reconstructed image.

Reconstruction Tomography from a limited number of projections has always been of vital interest. Because of the need to protect the patient from an excessive dose, it is desirable to take as few projections as possible as is consistent with the goal to get a medically acceptable reconstruction [31]. In many applications of CT, the projection data from only a small number of viewing angles are available. Images reconstructed from a limited number of projections using the conventional image reconstruction algorithms, which are designed for 360° coverage of viewing angles, suffer from a systematic geometric distortion and severe streaking artifacts [31].

From a mathematical standpoint, the challenge of limited dose CT can be expressed as reconstructing a n-by-n image using m projections with n equations each where $mn << n^2$. In this case the equations are highly underdetermined. If the equations are consistent, then there exists infinitude of possible solutions. If they are not consistent, solutions may be found within certain tolerances [18], again giving an infinite set.

The challenges of low dose CT can be summarized as below [37] –

a) Dose reduction: Projections need to be taken in as few angles and/or as few photons per angle as possible.

b) Accuracy: Reduce the number of false positives and false negatives in the reconstruction.

c) Spatial resolution of the radiation collector

d) Size of the radiation source

e) Speed of the reconstruction algorithm. Fast reconstruction is always expected to reduce the diagnostic time.

f) Density resolution of the radiation collector

g) Computer time and storage

h) Ability of the chosen reconstruction algorithm to handle noise

i) Resolution actually needed for a given type of diagnosis [37]

1.4 Motivation and Objective of the thesis

The reconstruction quality of CT for providing better diagnosis can be improved by the development of more efficient detectors, improving the display of the reconstruction or by improving the algorithms used for reconstruction. The work in this thesis addresses the latter two issues assuming that the detector quality has reached its limit.

The display of the reconstructed image is dependent on two main factors [46]:

a) Resolution of the display device

b) Pixel shape

The resolution of the display device depends on the display device hardware. While a larger number of pixels has led to higher image resolution so far, any further pixel

number increase is known to adversely affect the sensitivity, Signal to Noise (S/N) ratio and dynamic range, since the size of each pixel becomes increasingly smaller [46]. Even though the pixel shape seems to play a crucial role in the display of images, not much work has been done in the area of pixel shape. Hence in this thesis the task of comparing the image quality based on pixel shape is undertaken.

A circular pixel is the most suitable for an omni-directional image representation since almost all camera systems are based on perspective projection. The geodesic dome has been considered for spherical image representation [25]. However, in the geodesic dome, the connections of neighborhood pixels are complex. Further, it is difficult to represent the geodesic dome with a 2-D array. The problem is the same for 2-D. The pixel shape is usually square and not a circle.

This problem also exists in cell phone networks, where the shape of each cell (coverage area of one cellular tower) has to be a circle because the signal is transmitted omni-directionally [34]. However, a cell is typically hexagonal in shape, since it provides the ease of implementation and also resembles very closely to a circle.

Some of the properties of hexagons are as below -

a) A hexagon gives better rotational symmetry than a square. Rotational symmetry is important in CT, because the projections are taken at different angles. Also certain algorithms rotate the pixel [12]. These algorithms will become relatively easy if the unit of measure (pixel) were as rotationally symmetrical as possible.
b) Hexagons give a closed-packed structure without gaps.
c) A point in the hexagonal raster has the same distance to all its six neighbors.

d) Relatively easy math compared to a circular pixel.

In spite of the merits of hexagons as described above, images have almost always been represented in square pixels in CT. Researchers have preferred square pixels for the ease and speed of computation. With computer speed increasing at a rapid rate and sophisticated off-the-shelf software packages, it was time to investigate the quality of hexagonal pixels over the traditional square pixels.

In case of reconstructions from highly underdetermined equations, Algebraic Reconstruction Techniques (*ART like AART, MART, SIRT and SART) prove very helpful. The reconstruction quality of each of the techniques is dependent on various factors some of which are explained below –

1) **Nature of the original image:** The performance of the algorithm will differ based on the nature of the original image and what it contains in terms of objects, contrast and resolution.
2) **Ray Width:** Generally in CT, instead of considering the ray as a single straight line (no thickness), it is considered as a ray with thickness. The ray width refers to the thickness of the ray. See Figure 1-5 for the graphical illustration of ray width.
3) **Ray Gap:** The ray gap refers to the gap between two adjacent rays. The lower the ray gap the better the quality of the reconstructed image. Ray gaps could occur if there are collimators between the x-ray source and the patient. See Figure 1-5 for the graphical illustration of ray gap.
4) **Detector Width:** The detector width refers to the width of the detectors. See Figure 1-5 for the graphical illustration of detector width. The larger the detector

width the lower the resolution of the reconstructed image. On the other hand, the lower the detector width the lower would be the signal to noise ratio.

5) **Detector Gap:** In practical CT, the detectors are placed very close to one another. However there is a gap kept between two neighboring detectors to prevent energy transfer due to conduction between detectors. Also to reduce scattering noise caused by scattering photons, collimators are kept between two adjacent detectors. The gap between two adjacent detectors is referred to as the detector gap. See Figure 1-5 for the graphical illustration of detector gap.

6) **Pixel Width:** This parameter refers to the width of the pixel in the original and the seed image. See Figure 1-5 for the graphical representation of pixel width.

7) **Weighting Scheme:** The weight factors or the contribution that each pixel makes in a ray are calculated using four different approaches. "Binary", "Length of ray within a pixel", "Distance of center of pixel from center of ray" and "Contribution made by the pixel in adjacent rays" are the four weighting schemes presented in this thesis. The different schemes are explained in detail in section 4.5.

8) **Type of *ART:** AART, MART, SIRT and SART differ from one another in terms of the value of correction and the time at which it is applied.

9) **Seed image:** This is the initial estimate of the solution. [22] showed that the seed image plays a role in the convergence and reconstruction quality.

10) **Number of Cycles:** This refers to the number of times the code loops through all the angles and all the detectors for the entire image.

11) **Projection angles:** This parameter specifies the different angles at which the projections are taken.

12) **Projection Angle Ordering Scheme:** This refers to the order in which the projection angles are considered. "Sequential", "Fixed Angle", "Random", "Multilevel resolution access", "Weighted Distance" are the five projection angles ordering schemes discussed in this thesis. See section 4.6 for the discussion on the projection angle ordering schemes.

13) **Relaxation Factor:** The factor used for smoothing *ART (ART, MART, SIRT and SART) correction matrix.

The different parameters are explained graphically in Figure 1-5. This thesis talks about some of the factors affecting the performance and quality of the *ART reconstruction and discusses the results.

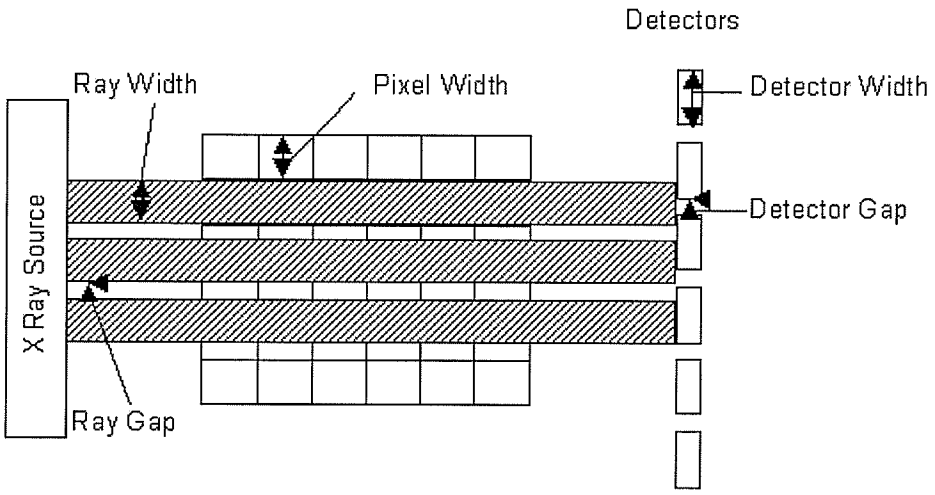


Figure 1-5. *Illustration of the different parameters that affect the convergence and quality of *ART reconstruction*

The first objective of the thesis is to compare hexagonal pixel resolution over square pixel resolution using different image quality measures. The second objective of the thesis is to study the effect of alternate seed images, projection angle ordering schemes and pixel weighting schemes on the reconstruction quality of *ART.

1.5 Organization of the thesis

Chapter 2 talks about the square vs. hexagonal pixel comparison experiment and shows the results, the third chapter of the thesis gives an introduction to the various reconstruction techniques. The fourth chapter discusses the *ART experiments and results. The fifth chapter talks about future work arising out of this thesis. Appendix A includes the Matlab™ code written for my experiments. Due to the nature of the experiments performed, the results are extensive and are summarized inside the chapters for illustrative purposes. Interested readers can see the tabulated results in Appendix B and C.

Chapter 2

Comparison of Square Pixel and Hexagonal Pixel Resolution

2.1 Overview

In most applications of image processing, data is collected and displayed in square pixels [16]. Hexagonal pixels offer the advantage of greater rotational symmetry in addition to close packed structure without gaps and a nearly circular pixel. We compared the image quality of images using square pixels with that of images employing hexagonal pixels. The comparison was done using various images, each considering a different aspect of geometry (i.e., lines at different angles, curves, etc.). The square pixel images were constructed using the average of a square area of smaller square pixels. Hexagonal pixel images were constructed using two techniques. The first one was called the "two-template approach", wherein two different templates were used to create a close packed hexagonal image from smaller square pixels. The second approach was called the "six neighbor approach" which creates a rectangular template using the six neighbors of a hexagonal pixel. Different image quality measurements such as Euclidean distance, resemblance measure, entropy and modulation transfer function (MTF) were used to compare the square pixel and hexagonal pixel images. Based on our results obtained using these quality measures, we conclude that contrary to our intuition and their widespread use in nature (retinas and ommatidia); hexagonal pixels appear to offer little or no significant advantage over conventional square pixels.

2.2 Introduction

For most modern display devices the shape of the pixels is square. It is due to this fact that in most applications of image processing, including computed tomography; data is gathered and arranged in square pixels [25]. The compound eye of insects and crustaceans is made of smaller, simple eye units, called ommatidia. The rhabdome is the common area where light is transmitted to the reticular cells. Each of these cells is connected to an axon and since each ommatidium consists of seven or eight reticular cells, there are these number of axons, which form a bundle from each ommatidium. Each ommatidium passes information about light from a single direction. The eyes of strepsipteran insects are very unusual among living insects. Externally they differ from the usual "insect plan" by presenting far fewer but much larger lenses. Beneath each lens is its own independent retina. Anatomical and optical measurements indicate that each of these units is image forming, so that the visual field is subdivided into and represented by "chunks," unlike the conventional insect compound eye that decomposes the visual image in a nearly point wise manner. This results in profound changes in the neural centers for vision and implies major evolutionary changes [13]. The total image formed therefore is a sum of the ommatidia fired. This resultant image can be thought of as a series of dots, just like a computer image is composed of a series of discreet pixels. The more pixels, the better the picture. Figure 2-1 shows the eyes of the mosquito. We can see that the ommatidia are more hexagonal than square shaped. It is this natural occurrence that motivated us to hypothesize that hexagonal pixels may provide a better image quality than square pixels.

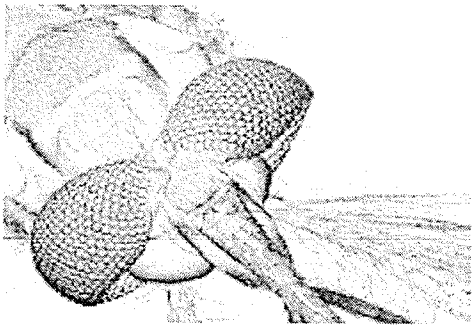


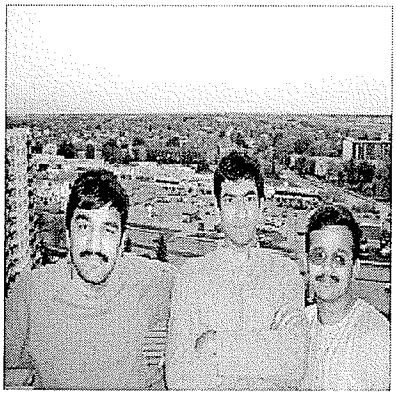
Figure 2-1. *The eyes of an insect such as a mosquito have hexagonally arranged ommatidia. [10]*

2.3 Methods and Materials

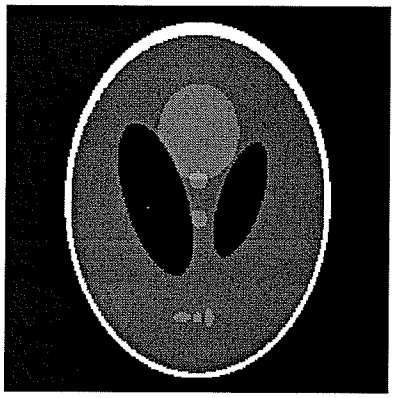
2.3.1 Platform

The experiments were done on a Windows98 PC with 256MB RAM and having a single AMD-K6 450MHz processor. The image processing programs were written in Matlab™. The images that were used for comparison purposes were assorted test patterns and not partial to any particular geometry. Some test patterns were mathematically created to observe and verify the accuracy of the image comparison algorithms. All test images were 256 by 256 pixels. Euclidean distance, resemblance measure, entropy and modulation transfer function (MTF) were the different image quality measures used for comparison. Figure 2-2 (a-h) show the different test patterns used for the experiment.

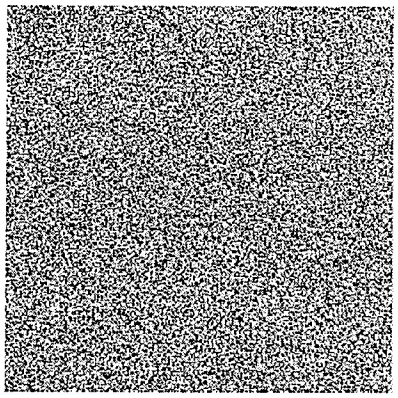
(a) admin256.bmp



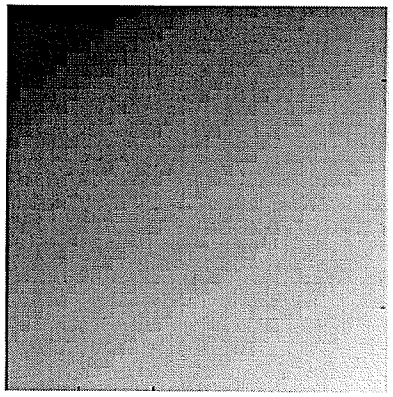
(b) balcony256.bmp



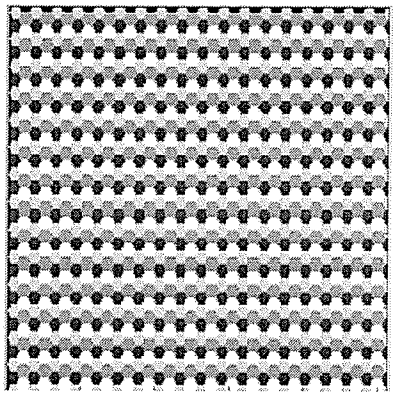
(c) phantom256.bmp



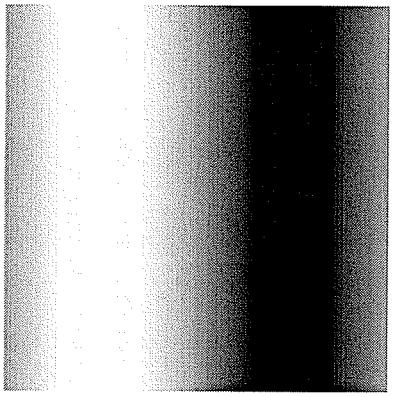
(d) rand256.bmp



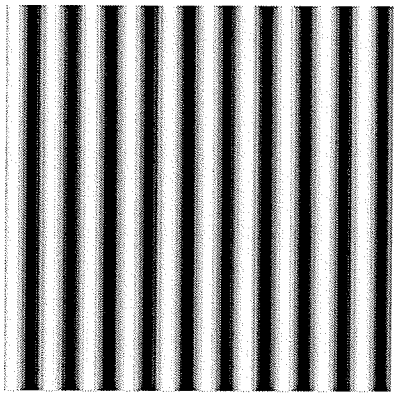
(e) square256.bmp



(f) hexagon256.bmp



(g) sinewave01_256.bmp



(h) sinewave10_256.bmp

Figure 2-2. *Test images used for comparing square and hexagonal pixel resolution.* **(a)** *University of Manitoba Administration Building (courtesy of Prof. W. Lehn, University of Manitoba, reproduced from his Digital Image Processing class).* **(b)** *Friends standing in balcony.* **(c)** *The Shepp-Logan brain Phantom image* **(d)** *Random image: Image created by uniformly distributed random numbers* **(e)** *Regular Square Image – This image is constructed by having one value for all pixels in a 8x8 square.* **(f)** *Regular Hexagon Image – This image is constructed by having one value for all pixels in a hexagon of length 4.5 pixels.* **(g)** *Sine wave image of frequency one across the width of the image* **(h)** *Sine wave image of frequency ten across the width of the image*

Note that the test images are referred to with their file names in this chapter. For example the University building image is stored in a file called admin256.bmp (Figure 2-2(a)). The "friends standing in a balcony" image is called friends256.bmp (Figure 2-2(b)). The Shepp-Logan brain phantom image is called phantom256.bmp (Figure 2-2(c)). The image created by uniformly distributed random numbers is called rand256.bmp (Figure 2-2(d)). The image created by 8 by 8 square blocks of uniform gray-level pixels is called square256.bmp (Figure 2-2(e)). The image created by assigning one pixel value to all pixels in a hexagon of length 4.5 is called hexagon256.bmp (Figure 2-2(f)). The image containing a horizontal sine wave of frequency one is called sinewave01_256.bmp (Figure 2-2(g)) and the image containing horizontal sine wave of frequency ten is called sinewave10_256.bmp (Figure 2-2(h)).

2.3.2 Euclidean Distance

Euclidean Distance is defined as the straight-line distance between two points. In a plane with point p_1 at (x_1, y_1) and point p_2 at (x_2, y_2), it is $((x_1 - x_2)^2 + (y_1 - y_2)^2)^{1/2}$ [34]. For comparing the difference between two images, the Euclidean Distance is calculated as the square root of the sum of the difference of the squares of pixels. For example if x_1, x_2, x_3... are the pixel values of image1 at position p_1, p_2, p_3... respectively and y_1, y_2, y_3... are the pixel values of image2 at the same positions then the per-pixel normalized Euclidean distance for an m by n picture is calculated by Equation 1. In equation 1, g indicates the maximum possible gray level of the image. The normalization is done by $g\sqrt{mn}$ to make the Euclidean distance dimensionless (independent of size of image) and gray scale independent.

$$\sqrt{\frac{1}{mn}\sum_{i=1}^{mn}\left(\frac{x_i - y_i}{g}\right)^2} \qquad (1)$$

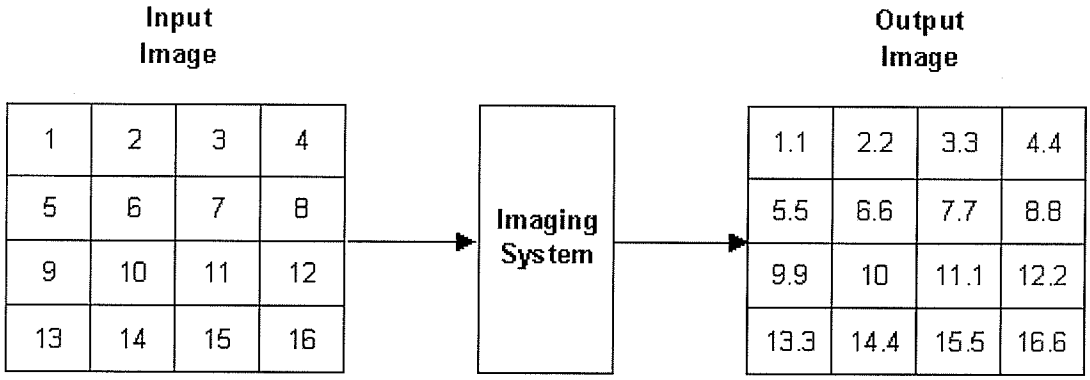


Figure 2-3. *Graphical/Mathematical Representation of an Imaging System. The numbers indicate pixel values.*

The Euclidean distance between Input Image and Output Image in Figure 2-3 with g=32 is given by

$$\frac{\sqrt{(1.1-1)^2 + (2.2-2)^2 + (3.3-3)^2 + \ldots..}}{32\sqrt{4*4}} = 0.0252$$

When two images are identical, the Euclidean distance measure will be 0. The maximum value that the Euclidean distance can have is 1, which indicates that the difference between the two images being compared is large.

2.3.3 Resemblity (Resemblance) as an Image Quality measure

The fundamental difficulty of Euclidean distance is that it calculates the quality of the image precisely, which could be different from human perception. To address this problem, Cornwell, Holdaway, and Uson introduced the resemblity measure [6] (for radio astronomy). The resemblity measure is defined as the ratio of the value of a pixel to the

error between the true sky distribution and the reconstructed image. In image processing, the ratio becomes the ratio of the value of a pixel to the error between the original image and the reconstructed image. In layman's terms this can also be described as the visual similarity between two images.

The resemblity measure [29] is given in equation 2.

$$XSD = \frac{\sum_{i=1}^{M}\sum_{j=1}^{M} g(i,j)f(i,j)}{\sqrt{\sum_{i=1}^{M}\sum_{j=1}^{N} f(i,j)^2}\sqrt{\sum_{i=1}^{M}\sum_{j=1}^{N} g(i,j)^2}} \tag{2}$$

where $g(i,j)$ and $f(i,j)$ are the two images whose resemblance to each other needs to be calculated. The resembility measure, like the normalized Euclidean Distance, is dimensionless (not dependent on the size of the image) and gray-scale independent. The resemblity measure of two identitical images will be 1. The resemblity is lowest if f is zero at all points where g is non-zero and vice versa. The lowest value of resemblity is 0. The resemblity measure cannot be greater than 1. It is interesting to note that the resemblity measure of two images where $f \propto g$ is 1. Also I calculated the resemblity of two uniformly distributed random images. The resemblity interestingly came close to 0.7 everytime.

The resemblity (resemblance) measure between the input $f(i,j)$ and output $g(i,j)$ image of Figure 2-3 when calculated will come to 0.99955.

2.3.4 Entropy as an Image Quality measure

To quantify the information of an image (the digital number of pixels) is similar to quantifying the information of communication. According to Shannon's assumption, one element of a large number of messages from an information source is just as likely as another, so the digital number of one pixel in an image is just as likely as another pixel. In any one image the number of pixels can be very large. In such cases, to quantify the information content of an image one can just satisfy the Shannon's assumption. Hence, it is reasonable to use Shannon's entropy in image analysis [30]. By applying Shannon's entropy in evaluating the information content of an image, the formula is modified as in equation 3:

$$H = -\sum_{i=1}^{G} d(i)\log_2[d(i)] \tag{3}$$

where G is the number of grey levels in the image's histogram ranging for a typical 8-bit image between 0 to 255 and $d(i)$ is the normalized frequency of occurrence of each grey level such that $\sum d(i)=1$. To sum up the self-information of each grey level from the image, the average information content is estimated in the units of bits per pixel. The entropy of an image is not an ideal measure for its information content [30]. It depends only on the probability of the elements of the image and totally disregards the spatial distribution of the pixel values. Therefore the image of a gray ramp can have the same entropy as random noise as long as the values have the same probability distribution. If entropy is used as the image quality measure for determining the quality of an imaging system, the ratio of the entropy of output image to the entropy of the input image is calculated. The imaging system is expected to preserve the amount of information in the

output. Hence the ratio is expected to be as close as possible to 1. Depending upon the nature of noise that the imaging system introduces, the output image could have more information than the input image. Entropy measure should be used only after careful consideration in these cases. Since entropy gives the probability distribution, it is never negative. The lowest value that the entropy could have is 0 where all the pixel values in the image are the same. Entropy is dimensionless i.e., it is not dependent on the size of the image. For the square and hexagonal pixel resolution comparison experiment, entropy is a good measure as both square and the hexagonal pixel output tends to smoothen the image and loses resolution and information content in the process. By comparing the entropy we can see by what factor each method preserves the information-content.

Algorithm for calculating image quality based on Entropy

a) Take the test image. Call it T.

b) Calculate the entropy of the test image using the formula given in equation 3. Call it E_t.

c) Pass the test image as input to the imaging system. Call the output image of the imaging system as O.

d) Calculate the entropy of the output image using the formula given in equation 3. Call it E_o.

e) The ratio of the entropy in the output image E_o to the entropy in the test image E_t gives us the entropy preservation of the imaging system.

The entropy ratio of the output to input in the square pixel and hexagonal pixel resolution comparison experiment will be less than one in most of the cases. However, they could be greater than one in some cases. One example is shown in Figure 2-4.

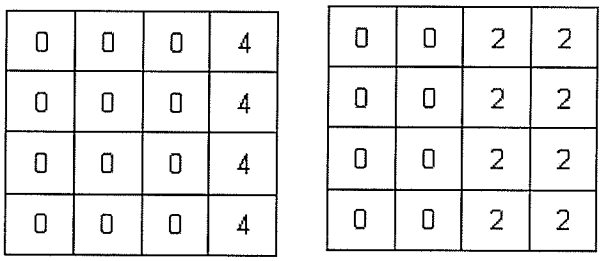


Figure 2-4: *Illustration of scenario where the ratio of output image entropy to input image entropy is greater than one. The numbers indicate pixel values.* **(a)** *Input Test image. Entropy of this image is 0.8113* **(b)** *Output image of the square pixel (explained later) counterpart. Entropy of this image is 1.*

2.3.5 Modulation Transfer Function (MTF) as an Image Quality measure

The modulation transfer function (MTF) characterizes the spatial resolution of an imaging system. MTF is the amplitude of a linear system's output in response to a sinusoidally varying input signal of unit amplitude. Equivalently, it is the magnitude of the Fourier transform of a system's response to an input signal that is a perfectly sharp, single point of light - the point-spread function of the system. Sine-wave frequencies, usually in units of cycles/mm are used as the metric for specifying detail in an MTF plot. These frequencies are always plotted as the independent variable on the x-axis. To complete the MTF metric, a measure of how well each sine-wave frequency is preserved after being imaged, i.e., transferred through an imaging device, is required. This measure, called modulation transfer is plotted along the y-axis for each available frequency.

Algorithm for calculating MTF

f) Create a test image that is made of a sine wave of frequency one. Measure the amplitude. Call it A_i

g) Pass this image through the imaging system. Measure the amplitude of the frequency one component of the output image. Call it A_o.

h) The ratio of A_o over A_i gives the modulation of the imaging system for frequency one.

i) Repeat steps a to c by for different frequencies. Plot the modulation function against frequency and you get the modulation transfer function. The typical plot of a imaging system's MTF is shown in Figure 2-5.

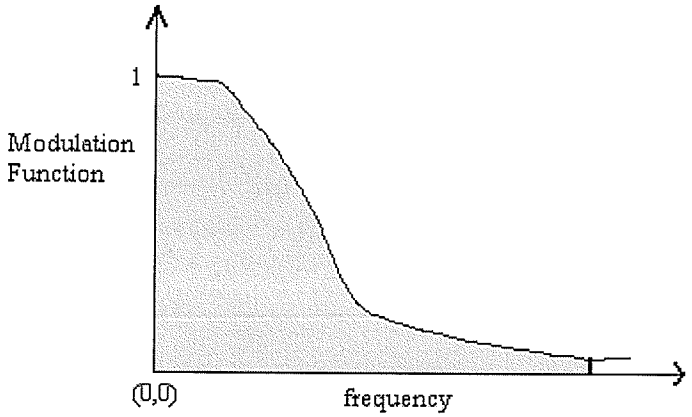


Figure 2-5. *Typical plot of MTF of an imaging system. The larger the shaded area, the better the MTF of the system.*

2.3.6 Test Images and Process Steps

The images that were used for carrying out the image quality analysis were shown in Figures 2-2(a-h). These images were reconstructed into squares of 64 pixels and hexagons of 62 pixels (average) in size. The manner in which this is done is explained below.

One test image from the ones shown in Figure 2-2 is taken at a time. This image is called the original image. The original image was broken into hexagons and each hexagonal pixel was given the average value of the pixels that fall in the hexagon. In order to ensure

that no two hexagons overlap and no gaps exist between two hexagons (see Figure 2-6) the hexagons were created using the following two approaches:

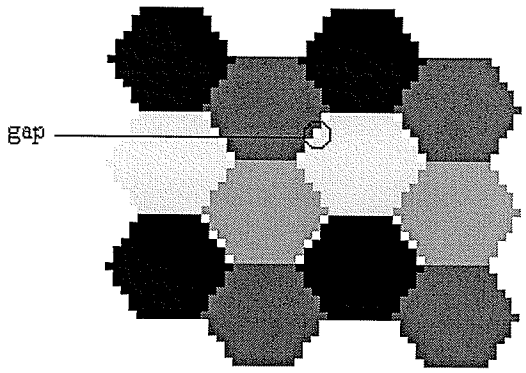


Figure 2-6. *Illustration of gaps, which can occur in hexagonal pattern created using a single definition of a hexagon.*

a) Two -Template Approach

In the two-template approach shown in Figure 2-7, the hexagons numbered 1 were created first, the hexagons numbered 3 were constructed later using the same formula as that of the hexagons numbered 1 and were vertically displaced by the height of the hexagon. The hexagons numbered 1 and 3 were called odd layered hexagons. Once the entire image was filled with odd layered hexagons, the hexagons numbered 2 were constructed such that they resemble very closely to the hexagons numbered 1 and do not include any pixel already taken by the odd layered hexagons and would include all pixels not considered by the odd layered hexagons. The hexagons numbered 4 were constructed similarly and were displaced by length equal to the hexagon's height from the hexagons numbered 2. The hexagons numbered 2 and 4 were called even layered hexagons. The

odd layered hexagons form one template, where as the even layered hexagons form another template. Hence this approach was called the two-template approach.

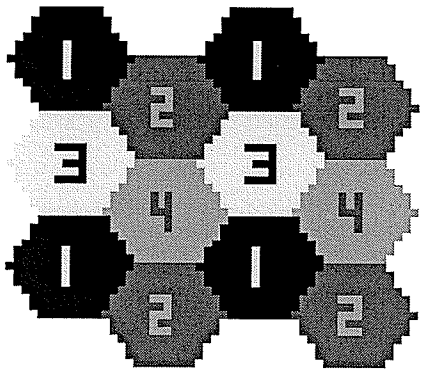


Figure 2-7. *Hexagonal Packed Structure using the two-template approach.*

b) Six -Neighbor Approach

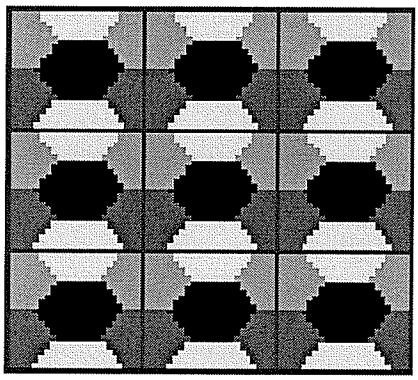


Figure 2 -8. *Hexagonal Packed Structure using the six-neighbor approach. Note that the grid is shown only to clearly identify the rectangular template.*

The six-neighbor approach used a rectangular template by considering one hexagon and part of its six neighbors as shown in Figure 2-8. The rectangular template was then replicated to tile the entire image. The two-template approach algorithm in itself ensured that no two hexagons overlapped and that no pixel was left out. However it was computationally cumbersome.

The six-neighbor approach on the other hand offered us the advantage of being computationally and programmatically efficient but required additional logic to ensure that all the pixels in the image were accounted exactly once. Using both the approaches described above the original images were converted into hexagonal pixels. The two-template approach gives the ability to count the number of pixels that lie in a hexagon, which is useful in comparing against the square pixel counterpart. Hence in most cases the Two-template approach has been used for the experiments illustrated in this chapter.

The original images (Figures 2-2(a-h)) were then converted into square pixels of size 8 x 8 and each square pixel was given a value equal to the average of the pixel values that fell in the square.

2.3.7 Evaluation of the Image Quality (Euclidean Distance, Resemblity and Entropy)

The image quality was evaluated using the different image quality measures.

Algorithm: -

a) Take a test image. Call it *T*.

b) Convert the test image into a square pixel image. Call it *S*.

c) Convert the test image into a hexagonal pixel image. Call it *H*.

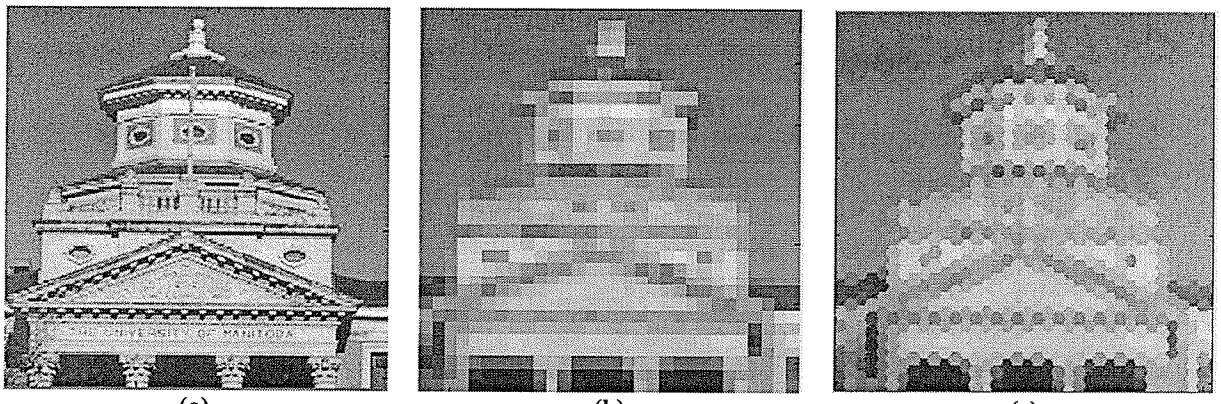
d) Calculate the image quality between *S* and *T*. This is Q_s.

e) Calculate the image quality between *H* and *T*. This is Q_h.

f) Plot $Q_h - Q_s$

g) Rotate T by 5 degrees and repeat steps b to f. The rotation is done to ensure that the results are not partial to any objects in the image. The rotation is carried out from 0 degrees to 360 degrees in increments of 5 degrees using bicubic interpolation. The graphical representation of the algorithm is represented in Figure 2-9. Note that even though these plots are shown for rotation angles of 0, 45 and 90 degrees, the actual experiment was done for rotation angles from 0 to 360 degrees in increments of 5 degrees. The graphs are shown in the following pages. The experiments were done for all test images, but only the admin256.bmp image is illustrated in this figure.

h) Image quality of the square pixel image at 0° is the difference between Figure 2-9(a) and Figure 2-9(b). The image quality of the hexagonal pixel image at 0° is the difference between Figure 2-9(a) and Figure2-9(c). Similarly the image quality of the square pixel image at 45° is the difference between Figure 2-9(d) and Figure 2-9(e) and the image quality of the hexagonal pixel image at 45° is the difference between Figure 2-9(d) and Figure 2-9(f) and so on.



(a) **(b)** **(c)**

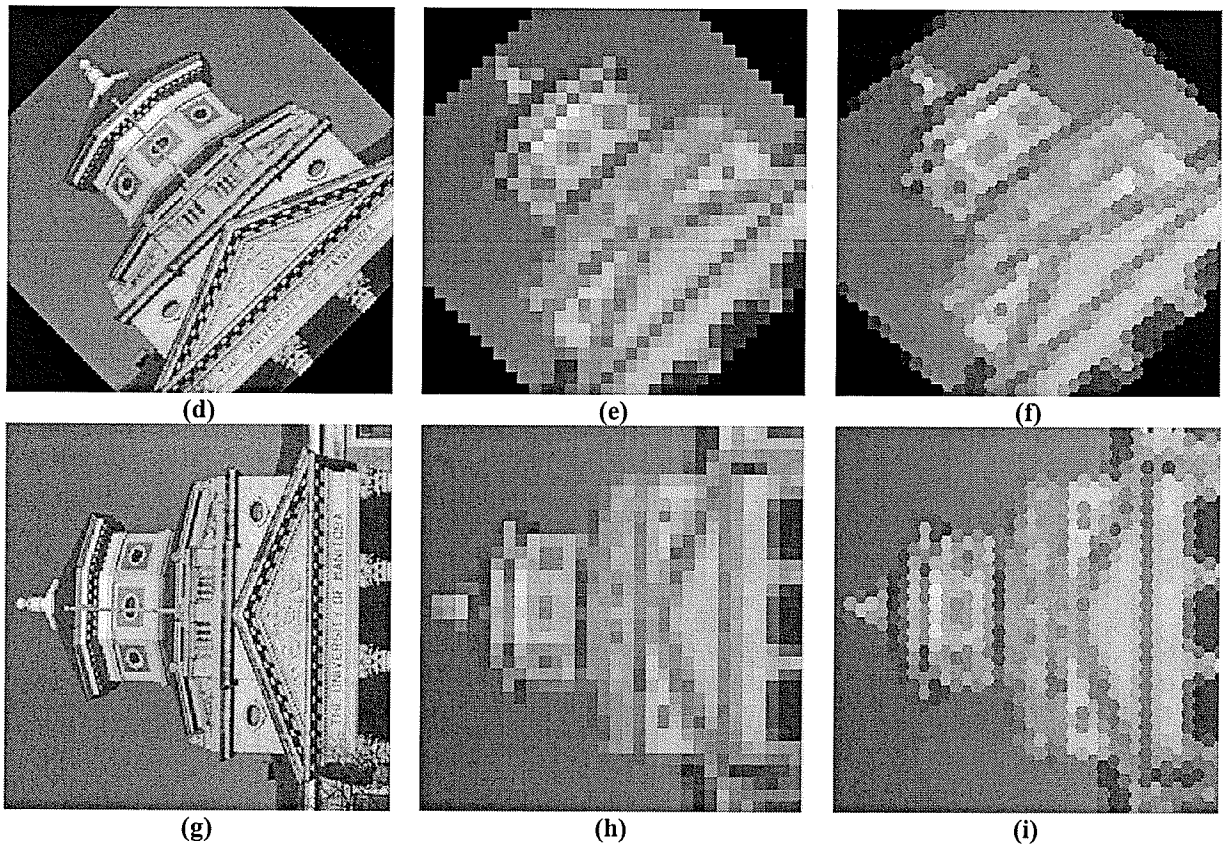
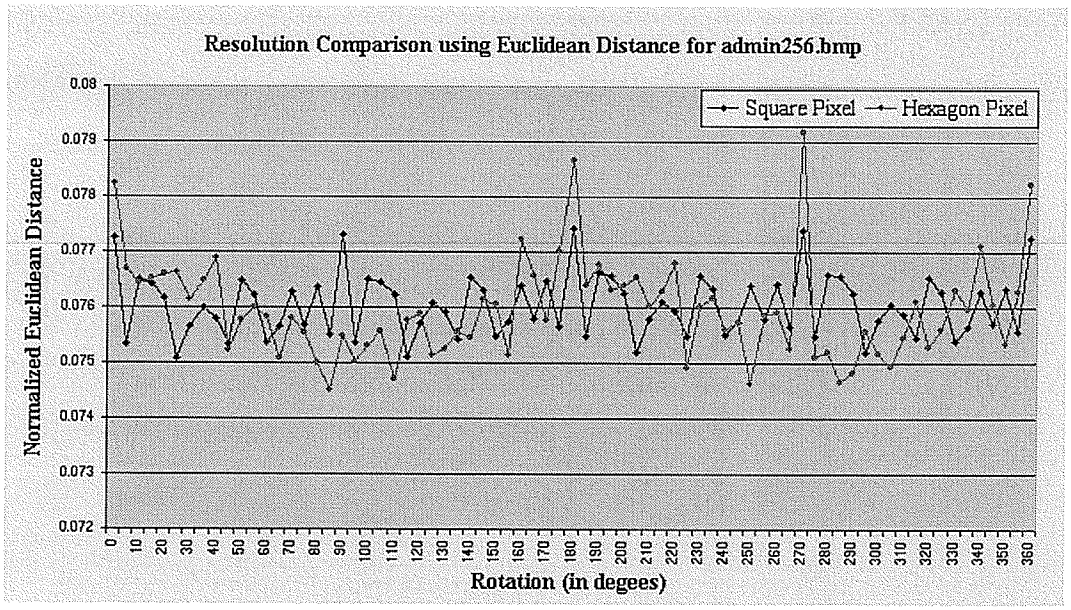


Figure 2-9. *Square vs Hexagon resolution comparison experiment process for evaluating Euclidean distance, resemblity and entropy image quality measures* **(a)** *University of Manitoba Administration Building (courtesy of Prof. W. Lehn, University of Manitoba, reproduced from his Digital Image Processing class)* **(b)** *The 256 x 256 test image shown in (a) is broken into square pixels each of size 8 x 8 (64 pixels). The entire 256 by 256 image would be filled with 32 by 32 such squares.* **(c)** *The 256 x 256 test image shown in Figure (a) is broken into hexagonal pixels each of length 4.5 (62 pixels).The entire 256 by 256 image is filled with 1057 hexagons. (b) and (c) are constructed using the two-template approach but the six-neighbor approach gives the same result.***(d)** *Image in (a) is rotated by 45 degrees.* **(e)** *Rotated Image in (d) is converted into square pixel image of size 8 x 8 (64 pixels)* **(f)** *Rotated image in (d) is converted into hexagonal pixel image of size 4.5 (62 pixels).* **(g)** *Image in (a) is rotated by 90 degrees.* **(h)** *Rotated image in (g) is converted into square pixel image of size 8 x 8 (64 pixels).* **(i)** *Rotated image in (g) is converted into hexagonal pixel image of size 4.5 (62 pixels).*

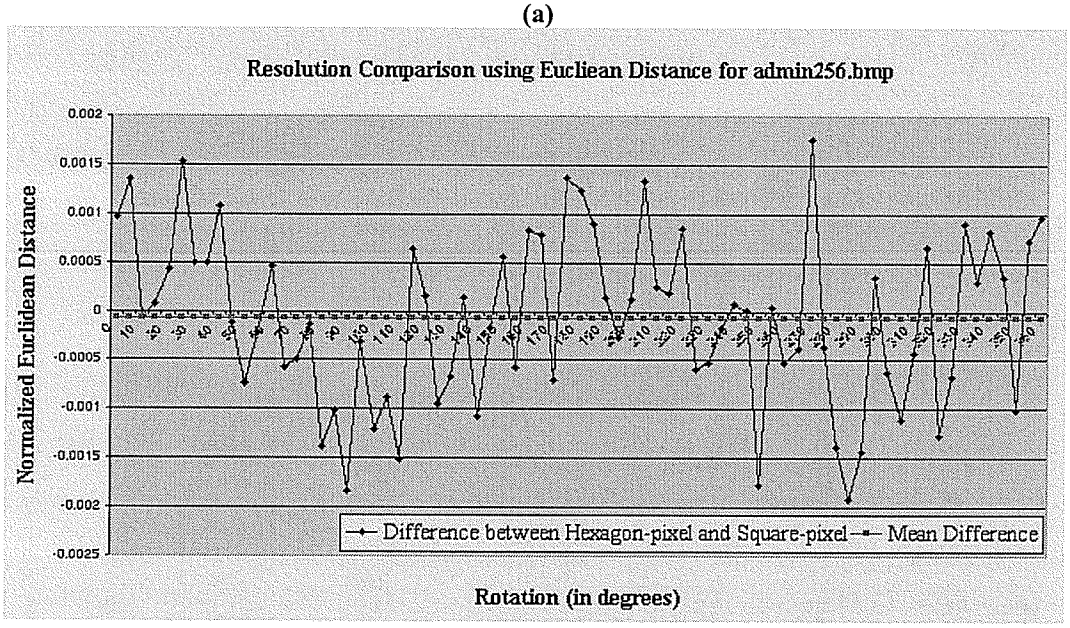
Rounding and interpolation errors are introduced when an image is rotated [28]. These rotation errors can be determined by rotating an image by θ and rotating it back by $-\theta$. The difference between the original image and the image obtained by rotating the image by θ and then by $-\theta$ gives an estimate of the rotation error. Also a major portion of rotation error occurs at the edges [28]. To minimize the rotation error, one should ignore the edges and just consider the center portion of the image. The rotation error gets

complemented if the rotation is done in increments. To avoid this the original image is rotated by the angle of rotation for each successive angle. The results presented in this chapter do not ignore the rotation error. Future work is warranted to estimate the effect of rotation error on the overall results.

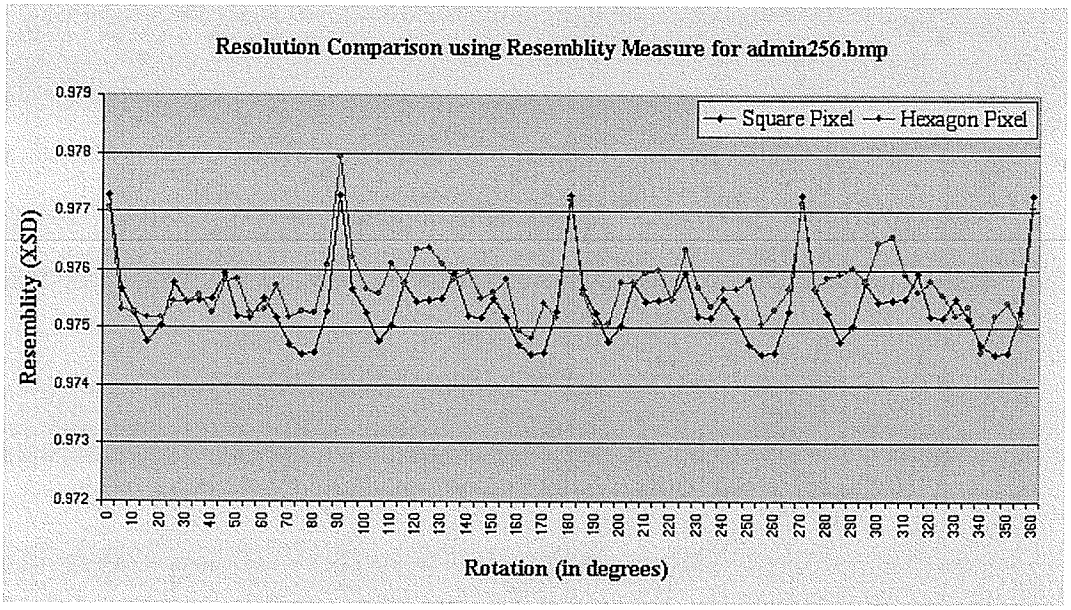
Instead of displaying the plots of the image quality measures for both square pixel and hexagonal pixel images as shown in Figure 2-10 (a, c, e), in the results section, I only show the plot of the difference in the error margin of the image-quality-measure between hexagon and square pixel images. This makes the graph much easier to read. The mean of the difference is represented as a continuous dotted line as can be seen in Figure 2-10 (b, d, f). If this line is above the horizontal axis, then it means that the square pixel method is better for that particular quality measure, if the dark line is below the horizontal axis, then it indicates that the hexagonal pixel method is better for that particular quality measure. 2 standard deviation (σ) is calculated for all experiments. From Figures 2-10 (a,c,e) it is difficult to interpret which pixel-method is better, but Figures 2-10 (b,d,f) simplifies this task.


Resolution Comparison using Euclidean Distance for admin256.bmp
Square Pixel
Hexagon Pixel
Normalized Euclidean Distance
Rotation (in degees)

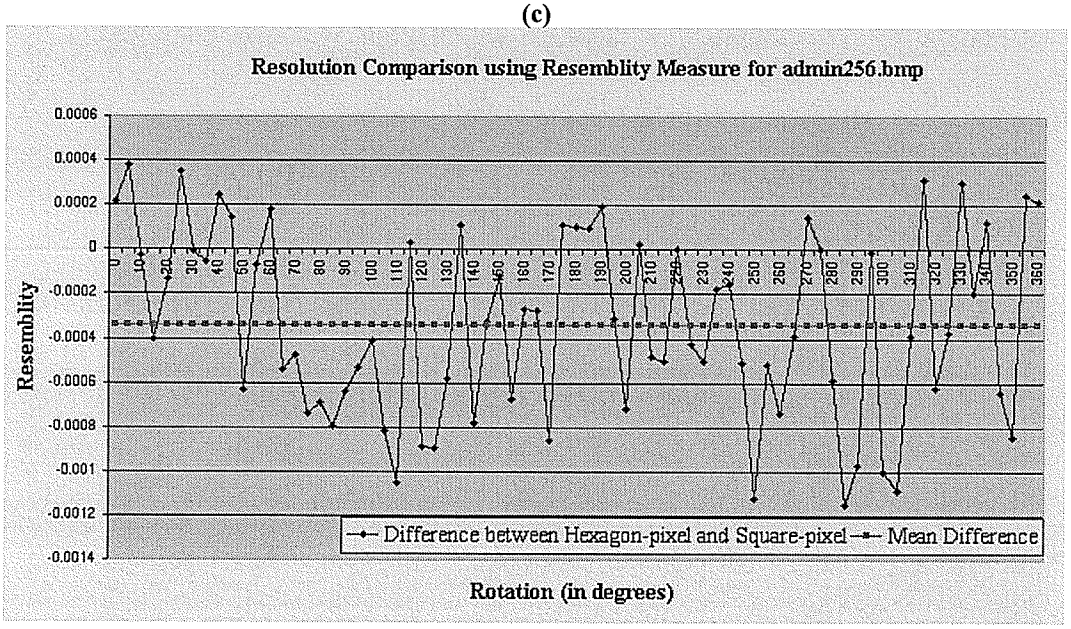
(a)


Resolution Comparison using Eucliean Distance for admin256.bmp
Difference between Hexagon-pixel and Square-pixel
Mean Difference
Normalized Euclidean Distance
Rotation (in degrees)

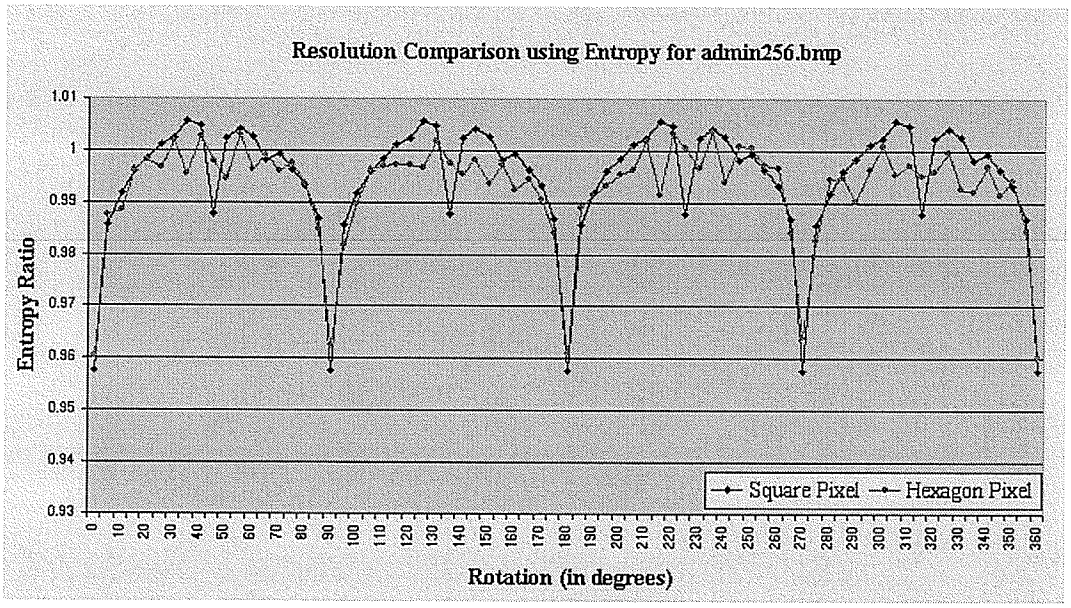
(b)


Resolution Comparison using Resemblity Measure for admin256.bmp
Square Pixel
Hexagon Pixel
Resemblity (XSD)
Rotation (in degrees)

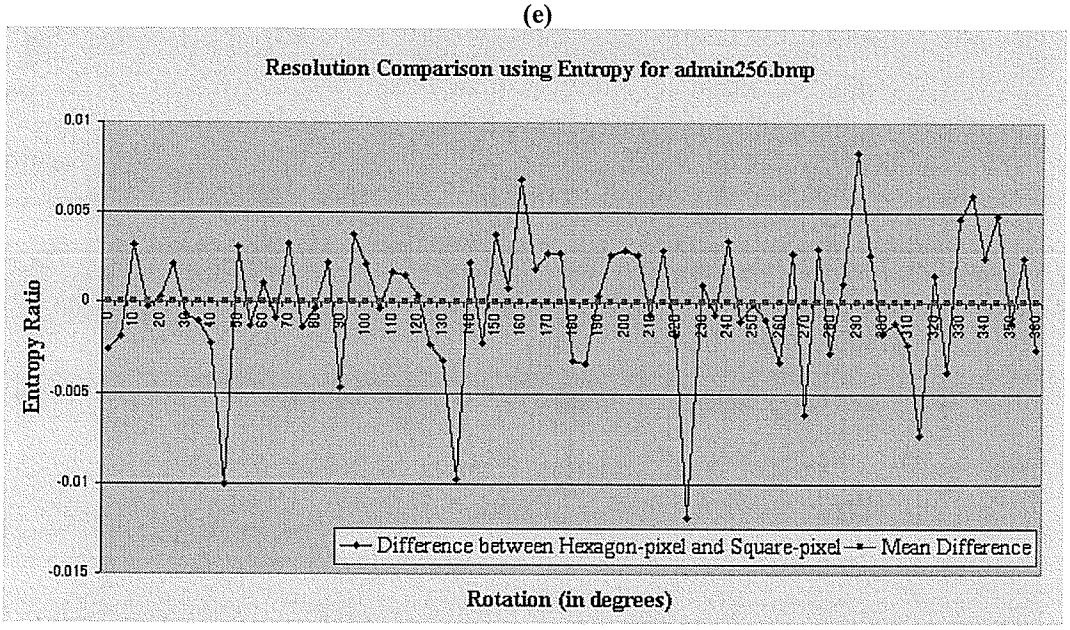
(c)


Resolution Comparison using Resemblity Measure for admin256.bmp
Resemblity
Difference between Hexagon-pixel and Square-pixel
Mean Difference
Rotation (in degrees)

(d)



(e)



(f)

Figure 2-10. *Euclidean distance, resemblity and entropy plots for comparison between square pixel and hexagonal pixel and how they are simplified by only showing the difference.* **(a)** *The Euclidean distance is plotted as a function of rotation in degrees for both the square and hexagonal pixel images.* ***(b)*** *The difference between the Euclidean distance of the hexagonal pixel image and the square pixel image is plotted as a function of rotation in degrees. This graph gives an easy representation as to which (square pixel or hexagonal pixel) is better in terms of Euclidean distance* **(c)** *The resemblity measure is plotted as a function of rotation in degrees for both the square and hexagonal pixel images.* **(d)** *The difference between the resemblity of the hexagonal pixel image and the square pixel image is plotted as a function of rotation in degrees. This graph is for easy graphical comparison of resemblity between the two pixel-resolution methods.* **(e)** *The entropy is plotted as a function of rotation in degrees for both the square and hexagonal pixel images.* **(f)** *The difference between the entropy of the hexagonal pixel image and the square pixel image is plotted as a function of rotation in degrees. This graph gives an easy graphical comparison of entropy between the two pixel-resolution methods.*

2.3.8 Evaluation of the Image Quality using MTF

The image quality was evaluated using MTF using the algorithm below –

Algorithm: -

a) Create a sine wave image of frequency one. Call it *T*.

b) Calculate the FFT of *T* and get the amplitude at frequency one.

c) Convert the test image into square pixel image. Call it *S*. Calculate the FFT of *S* and get the amplitude for frequency one. Call it A_s.

d) Convert the test image into hexagonal pixel image. Call it *H*. Calculate the FFT of *H* and get the amplitude at frequency one. Call it A_h

e) Calculate the ratio of A_s/A_t. This is the modulation factor of the square pixel image at frequency one.

f) Calculate the ratio of A_h/A_t. This is the modulation factor of the hexagonal pixel image at frequency one.

g) Increase the frequency gradually and repeat steps a to f.

h) Plot a graph of the modulation function at the different frequencies as a function of frequency.

Graphical procedure of calculating MTF is shown in Figure 2-11.

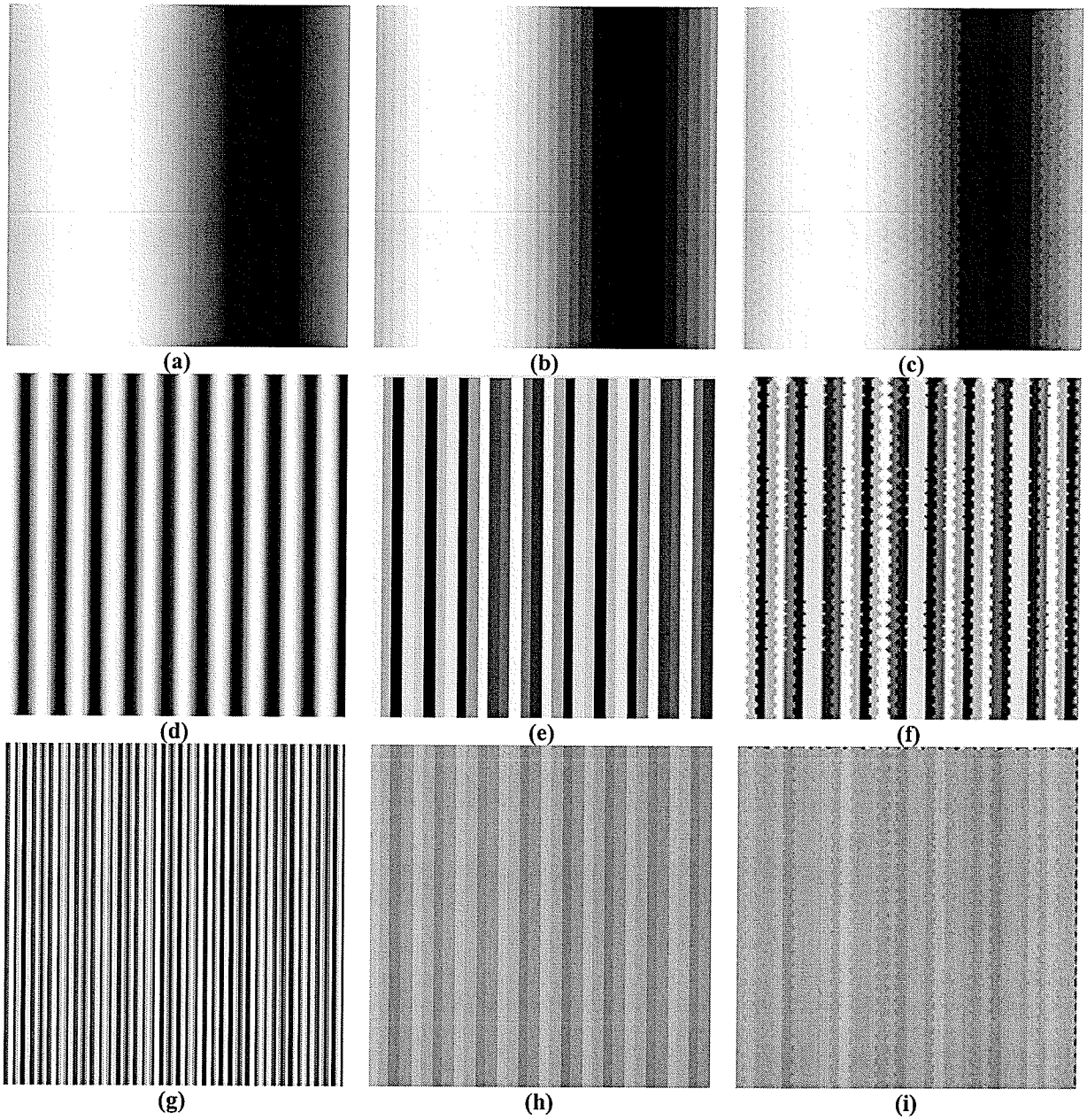


Figure 2-11. *Square vs Hexagon resolution comparison experiment process for evaluating MTF quality measure* **(a)** *Sinewave of frequency one.* **(b)** *The 256 x 256 test image shown in (a) is broken into square pixels each of size 8 x 8 (64 pixels)* **(c)** *The 256 x 256 test image shown in Figure (a) is broken into hexagonal pixels each of length 4.5 (62 pixels). (c) is constructed using the two-template approach but even the six-neighbor approach gives the same result.* **(d)** *Sinewave of frequency ten cycles across the width of the image* **(e)** *Image in (d) is converted into square pixel image of size 8 x 8 (64 pixels)* **(f)** *Image in (d) is converted into hexagonal pixel image of size 4.5 (62 pixels).* **(g)** *Sinewave of frequency forty across the width of the image.* **(h)** *Image in (g) is converted into square pixel image of size 8 x 8 (64 pixels).* **(i)** *Image in (g) is converted into hexagonal pixel image of size 4.5 (62 pixels).*

Note that even though these plots are shown for frequencies 1, 10 and 40, the actual experiment was done for frequencies from 1 to 128 in increments of one. The MTF plot for the square pixel, hexagonal pixel comparison experiment is shown in Figure 2-20.

2.4 Results and Discussion

The results of the Euclidean distance, resemblity and entropy evaluation are shown for all test images in Figures 2-13 to 2-20. The graphs show the difference in image quality measures between hexagonal pixels and square pixels. The difference is p lotted in the graphs as it gives a clearer picture of the behavior of the two pixel methods. The mean difference is also plotted in the same graphs. The mean difference gives an easy interpretation as to which pixel method is better at a quick glance for the entire rotation. If the mean difference line is above the horizontal axis then it indicates that the square-pixel is better. If the mean difference line is below the horizontal axis then it indicates that the hexagonal pixel is better. The distance of this mean difference line from the horizontal axis indicates the extent of the superiority of one pixel-method over the other for the particular image quality measure.

Figures 2-13a to 2-20a shows the Euclidean distance plots. Note that the Euclidean distance is based on hexagonal pixels generated using the two-template approach. Figures 2-13b to 2-20b shows the Resemblity plots for all the test images. Figures 2-13c to 2-20c shows the Entropy plots for the test images. Figures 2-13d to 2-20d tabulate the summary of the plots including the 2σ.

2.4.1 Results for the admin256.bmp (University of Manitoba building) image

Figure 2-13 shows the image quality comparison for the admin256.bmp image. This image is a real life picture and has objects partial to both square and circles. However, since the square pixel can lie exactly over the square objects, but the hexagonal pixels

cannot over the circular object, it was expected and seen that the Euclidean distance at 0° for the square pixel image is less than the hexagonal pixel image. As soon as the image is rotated by a small angle of 5°, some of the lines in the image got aligned with the square and hexagonal pixel counterpart, thereby bringing the Euclidean distance down at 5°. These 5° lines are the ones that exist in the top arch of the admin building. However, there are no objects that are at a 10° angle. Hence when I rotated the image by 10°, the Euclidean distance of the square-pixel image increased. However, at 10° rotation the rectangular pellets that exist just above the "The University of Manitoba" label overlapped one another, which brought the hexagonal pixel resolution down. For each increment of rotation, certain objects add up and certain objects move away from the standard square and hexagonal pixel geometry thus making the plot very volatile. An image when rotated at a certain angle using bicubic interpolation will have its own rounding precision error. However, when the image is rotated at 90°, 180° and 270°, these errors are small. Hence I paid special attention to these rotation angles. At 90°, 180° and 270° rotations the square pixel will fit in exactly the same way as the 0° (original) image. Hence the square-pixel Euclidean distance at these rotations angles is nearly same. One would have expected the image quality measures to be identical at 0° and 60° for a hexagonal pixel. However, since the center of the image is (128.5, 128.5) and is not the center of the centermost hexagon, the rotation of the image is not symmetrical at the center. Hence the results did not match my expectation. One way to work around this problem is to rotate the image at the center of the centermost hexagon and see the result. The explanation of the resemblity measure follows along the same lines. Resemblity gives the degree of resemblance between two images. The closer the resemblity measure

to 1, the better the resemblance between the two pictures. At 0^{o} the background of the admin image finds a very good match with the square pixel image than the hexagonal pixel counterpart and hence the resemblance of the square-pixel image to the original image is better than the resemblance of the hexagonal pixel image to the original image. This is true even at 180^{o}. However, when the angle increases above 0^{o}, the difference between the square and hexagon image quality based on resemblity measure is not very different. At 90^{o} rotation, the bottom part of the image coincides with the hexagonal pixel image. Hence the resemblity measure is good for the hexagonal pixel image compared to square pixel image. When I did a summary of the resemblity measure between square and hexagonal pixel image for all the rotation angles, I found that the resemblity measure of the hexagonal pixel is 0.0345% better than the resemblity measure of square pixel. But this difference could also be because in my experiment there are 62 pixels in one hexagon compared to 64 pixels in one square. I did the experiment for a larger hexagonal pixel size (one hexagon having 69 pixels) and found that the resemblity decreases for the larger-hexagon hexagonal pixel image. The entropy measure is a measure of how well the hexagon and square pixel preserve information content. The entropy ratio will be less than 1 in most cases, but scenarios still exist where the entropy ratio can be greater than 1. See section 2.3 for the specific scenario. Since the number of pixels in the hexagon image is more than the number of pixels in the square image, it was expected that the hexagonal pixel would retain more information that its square pixel counterpart. This was found to be the case.

Summary (for admin256.bmp)

Quality	% Difference	2σ	Which is better?
Euclidean Distance	0.1034%	0.0315%	Hexagon

Resemblity	0.0345%	0.0012%	Hexagon
Entropy	0.0008%	0.0100%	Square

2.4.2 Results for the balcony256.bmp (Friends standing in the balcony of a high rise building) image

The balcony test image was used more for interest than intelligence. The results are shown in Figure 2-14. Since this image does not have lot of edges (straight lines), I expected the hexagon image to be better than the square image. However, the degree of difference between the square and hexagon image is much less than my anticipated difference. The ground seems to be tilted at -20°; hence rotating the image by 20° aligns it with the square pixel, thereby bringing the Euclidean distance down. The explanation follows on similar lines as that for the admin256 image. Since this image has more objects that are a little partial to circular symmetry, the hexagonal pixel did find a better match thereby bringing the overall Euclidean distance down.

As far as the resemblity measure is concerned, I thought the resemblity measure for the hexagon image would be better than square image, since I visually found the hexagon image more pleasing and resembling more closely to the original image compared to the square image. This came out correct.

The entropy measure of the square-pixel came out better than hexagonal pixel. This is rather surprising. I currently have no explanation for this.

Summary (for balcony256.bmp)

Quality	% Difference	2σ	Which is better?
Euclidean Distance	0.6181%	0.0403%	Hexagon
Resemblity	0.0483%	0.0010%	Hexagon
Entropy	0.1431%	0.0101%	Square

2.4.3 Results for the phantom256.bmp (Shepp-Logon brain phantom) image

Figure 2-15 shows the image quality comparison for the phantom256.bmp file. Since this image has got more circular objects, I was expecting the hexagon image to have better resolution than the square image. Since the borders of this image are 0, this is a good image to test. In this case the noise introduced by the rotation algorithm is less. Hence the result tells us the difference between square and pixel image more correctly. Even though this image has more circles, the border white circle resembles very closely to a line and the square pixel image fell exactly on this line. Hence the Euclidean distance became less for the square image compared to hexagon image. Since the image is centered in the middle, the noise introduced by the rotation algorithm is less, hence the entropy ratio falls in the same range at 0° and any other rotation angle (unlike other images where the ratio is minimum at 0° rotation). Since at 45°, the square pixel image overlaps exactly with the border of the circular objects, the preservation of information at this angle for a square image is large. The distance of the pixel values between the original image and the pixel values in the square-pixel image are closer than the distance of the original image pixel values to the hexagon image. This is a strange result and needs more exploration. I expect that if the center of the centermost hexagon had been the center of the image, we would have got slightly different results.

Summary (for phantom256.bmp)

Quality	% Difference	2σ	Which is better?
Euclidean Distance	0.2351%	0.0723%	Square
Resemblity	0.1264%	0.0251%	Hexagon
Entropy	0.2723%	0.0463%	Hexagon

2.4.4 Results for the rand256.bmp image

Figure 2-16 shows the image quality comparison for the rand256.bmp file. This test image is impartial to any geometry; it has all frequency components (as can be seen from its FFT in Figure 2-12b), and has a uniformly distributed histogram (as can be seen from its histogram plot in 2-12c).

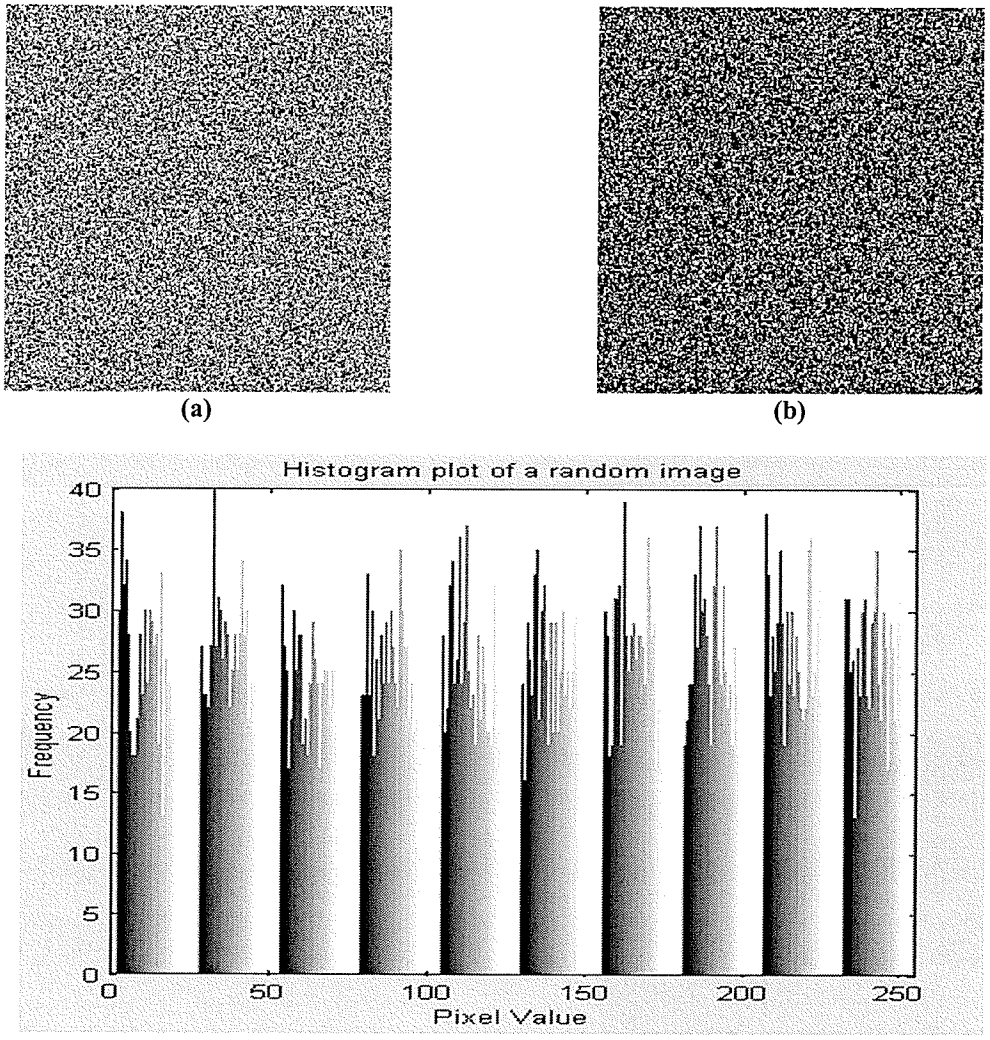


Figure 2-12. (a) *Random Image generated by a uniformly distributed random number generator* **(b)** *FFT of the image shown in (a).* **(c)** *Histogram of the image shown in (a).*

The Euclidean distance is 0.0021% better for square image as compared to hexagon image. The resemblity of the hexagon image came out better than square. The biggest difference was noticed in entropy. Entropy is the ratio of information content in the target image to that of the original image. Since the hexagon image has more number of pixels than the square image (62 pixels in one hexagon compared to 64 pixels in one square), there is a greater likelihood of the hexagon image retaining the information than the square image. Local peaks were observed in all the quality measures at rotation angles of 0°, 90°, 180° and 270°. These were caused because in other rotation angles, the small local homogeneous blocks that exist in the random image get busted up during rotation, whereas at 0, 90, 180 and 270 they are not altered.

Summary (for rand256.bmp)

Quality	% Difference	2σ	Which is better?
Euclidean Distance	0.0021%	0.0020%	Square
Resemblity	0.1331%	0.0011%	Hexagon
Entropy	0.7402%	0.0254%	Hexagon

2.4.5 Results for the square256.bmp (Mathematically created to fit the 8 by 8 square pixel-image)

Figure 2-17 shows the image quality comparison for the square256.bmp file. This test image was created for two reasons – 1) to check the accuracy of the code and 2) to check the behavior of the pixel-methods to images that fit its geometry.

Since the image was partial to square pixels, I expected that the Euclidean distance of square pixels would be better than the Euclidean distance of its hexagonal pixel counterpart. Even though my hypothesis turned out to be correct, the extent of the difference is small. The square-pixel method gave a 6.8704% improvement in the

Euclidean distance. The square-pixel image is exactly similar to the squar256.bmp image at the rotation angles of 0°, 90°, 180° and 270° and hence the square-pixel Euclidean distance at these angles is zero. Hence the plot shows the peak at these angles. However, as soon as the image is rotated by a small angle, the square-pixel does not fit the rotated image any more. Hence the Euclidean distance changes rapidly and becomes comparable with the Euclidean distance of the hexagonal pixel image.

One interesting fact was that the resemblity of the hexagonal pixel image came out better than the square-pixel image.

Summary (for square256.bmp)

Quality	% Difference	2σ	Which is better?
Euclidean Distance	6.8704%	0.7570%	Square
Resemblity	0.0390%	0.0009%	Hexagon
Entropy	0.0823%	0.0608%	Square

2.4.6 Results for the hexagon256.bmp (Mathematically created to fit the hexagon of length 4.5)

Figure 2-18 shows the image quality comparison for the hexagon256.bmp file. This test image was created for two reasons – 1) to check the accuracy of the code and 2) to check the behavior of the pixel-methods to images that fit its geometry.

Since the image was partial to hexagonal pixels, I expected that the Euclidean distance of hexagonal pixels would be better than the Euclidean distance of its hexagonal pixel counterpart. This was found to be the case. The hexagonal pixel method gave a 3.2687% improvement in the Euclidean distance. The hexagonal pixel image is exactly similar to the hexagon256.bmp image only at the rotation angle of 0°. As soon as the rotation starts the hexagonal pixel image does not match with the rotated image. We could have

expected this to happen at least at the rotation degrees of 60, 120, 180, 240, 300, but as the center of the centermost hexagon is not the center of rotation, there is no pattern across these angles. The hexagonal pixel shows the best quality match over its square-pixel counterpart at 0° since it matches exactly with the test image. As soon as a slight rotation is applied, the hexagonal pixel image no longer matches with the rotated image making the image quality measure comparable with its square-pixel counterpart. When the image is rotated 180°, because of the two different templates used in construction of the hexagon, the hexagonal pixel image does not exactly overlap with that of the rotated image but the overlap area is increased. This explains the small peak that we see at 180° angle.

Summary (for hexagon256.bmp)

Quality	% Difference	2σ	Which is better?
Euclidean Distance	3.2687%	0.4801%	Hexagon
Resemblity	0.2986%	0.0309%	Hexagon
Entropy	4.5298%	0.7157%	Hexagon

2.4.7 MTF Results

The modulation transfer function describes the quality of an imaging system by giving the amplitude response of the system to different spatial frequencies. The blurring caused by the system causes the amplitude to be reduced as spatial frequency increases. For the square-pixel and hexagonal pixel comparison experiment, the square-pixel and the hexagonal pixel can be considered as the imaging system in itself as they take a test image as input, execute their respective algorithm and produce an output image. Also since both the methods take the average over an area of pixels of the original image, they essentially are blurring the image.

The plot of MTF results obtained for the square-pixel and hexagonal pixel comparison experiment is shown in Figure 2-21. Since the area under the curve is more for the hexagonal pixel image one can say that the hexagonal pixel image gives a better MTF than its square-pixel counterpart. Note that the square-pixel MTF becomes zero at frequencies 32, 64, 96 and 128. This is because at these frequencies, the square-pixel image averages out evenly across a 8 by 8 block thereby making it lose the frequency component. The square-pixel image becomes a flat image at these frequencies and therefore has only the DC component. The peak occurring in the MTF of the hexagonal pixel image at frequency 18 is rather curious and needs further analysis.

Summary

Quality	% Difference	2σ	Which is better?
MTF	3.1003%	0.1066%	Hexagon

2.5 Conclusion

I averaged the results of all the test images. They summarize as below

Euclidean Distance: 0.9594% ± 0.0391% better for hexagonal pixel

Resemblity: 2.0483% ± 0.0317% better for hexagonal pixel

Entropy: 8.8201% ± 0.2989% better for hexagonal pixel. Note that this averaged value looks higher only because of the increase in entropy for the hexagon256.bmp image at 0° rotation.

MTF: 3.1003% ± 0.1066% better for hexagon.

Since in the experiments each hexagon has 62 pixels compared to 64 pixels in a square, there were approximately 1057 hexagons compared to 1024 squares for the test images (Note that the test images are 256 by 256 pixels). Hence I was expecting an improvement

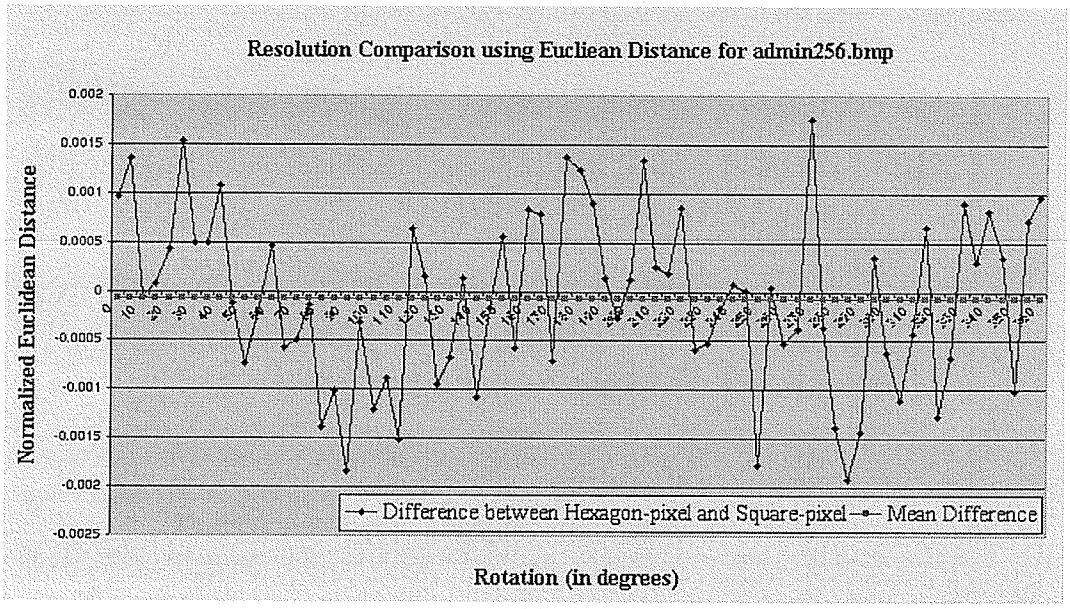
in hexagonal pixel. I estimated this improvement by plotting a graph of the image quality measure as a function of number of pixels that fit the 256 by 256 image. This plot is shown in Figure 2-22. Based on this plot, I calculated the image quality of the reconstruction that will have 1057 square pixels in the 256 by 256 image. A 62 pixel square image shows an improvement over a 64 pixel square image as below –

Euclidean distance: 2.45%

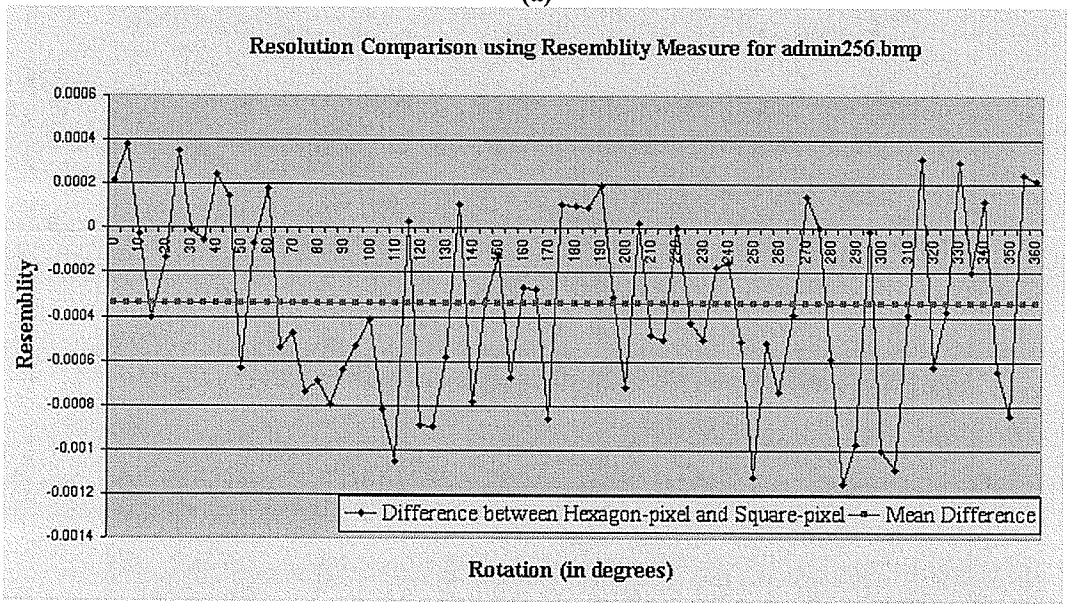
Resemblity: 7.56%

Entropy: 2.84%

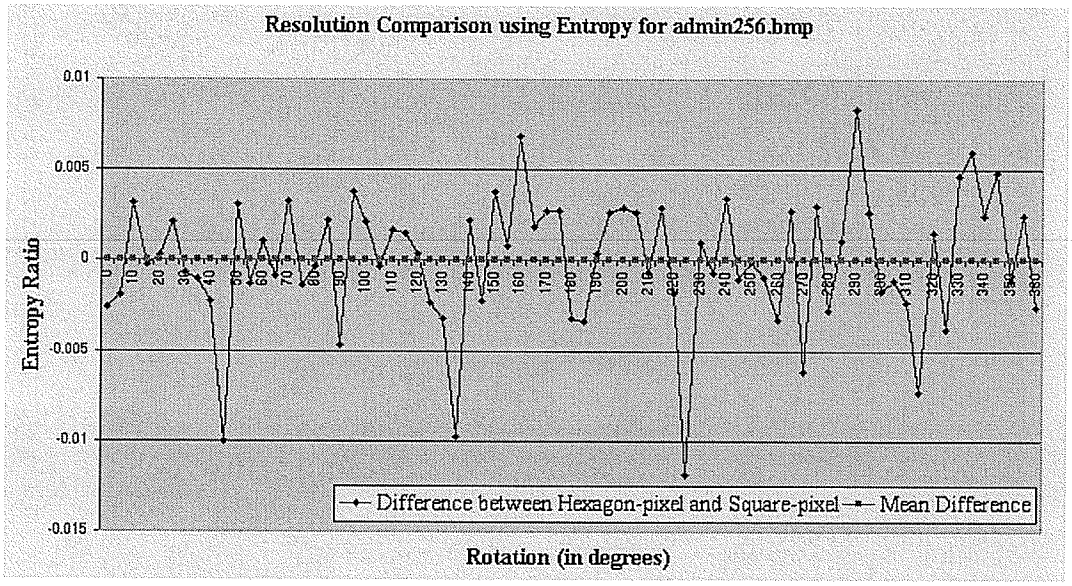
Also it cannot be generalized that the hexagonal pixel is always better than square-pixel as we have seen from the experiments that the nature of the test image plays an important part in determining which one is better (compare results of phantom256.bmp and balcony256.bmp). Due to the closeness of the quality measures between the two pixel-methods I conclude based on my experiments and the image quality measures employed, that hexagonal pixels do not appear to offer any significant advantage over conventional square pixels.


Resolution Comparison using Eucliean Distance for admin256.bmp
Normalized Euclidean Distance
0.002
0.0015
0.001
0.0005
0
-0.0005
-0.001
-0.0015
-0.002
-0.0025
Difference between Hexagon-pixel and Square-pixel
Mean Difference
Rotation (in degrees)

(a)


Resolution Comparison using Resemblity Measure for admin256.bmp
Resemblity
0.0006
0.0004
0.0002
0
-0.0002
-0.0004
-0.0006
-0.0008
-0.001
-0.0012
-0.0014
Difference between Hexagon-pixel and Square-pixel
Mean Difference
Rotation (in degrees)

(b)



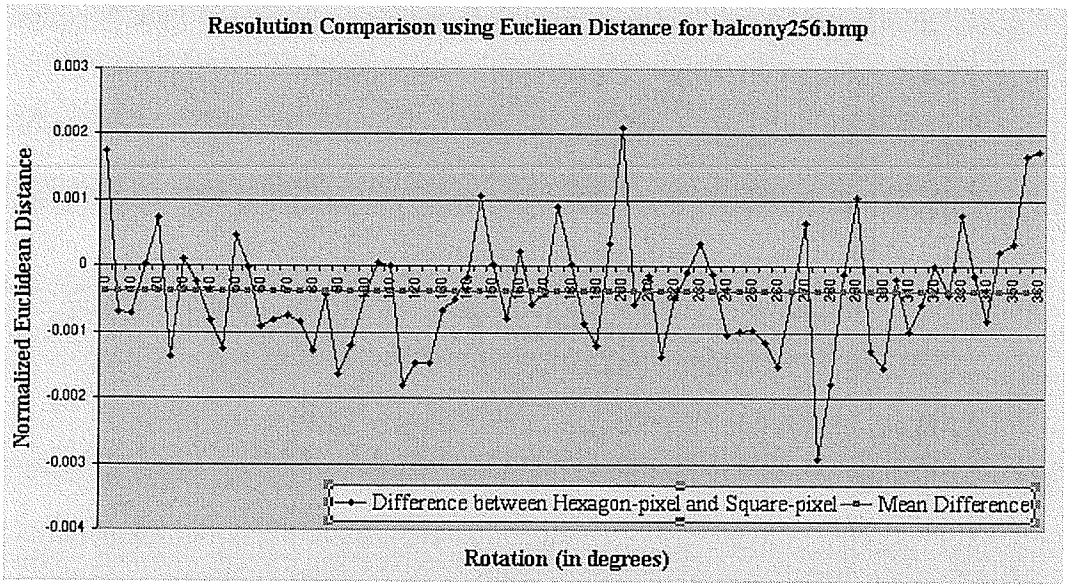
(c)

Summary (for admin256.bmp)

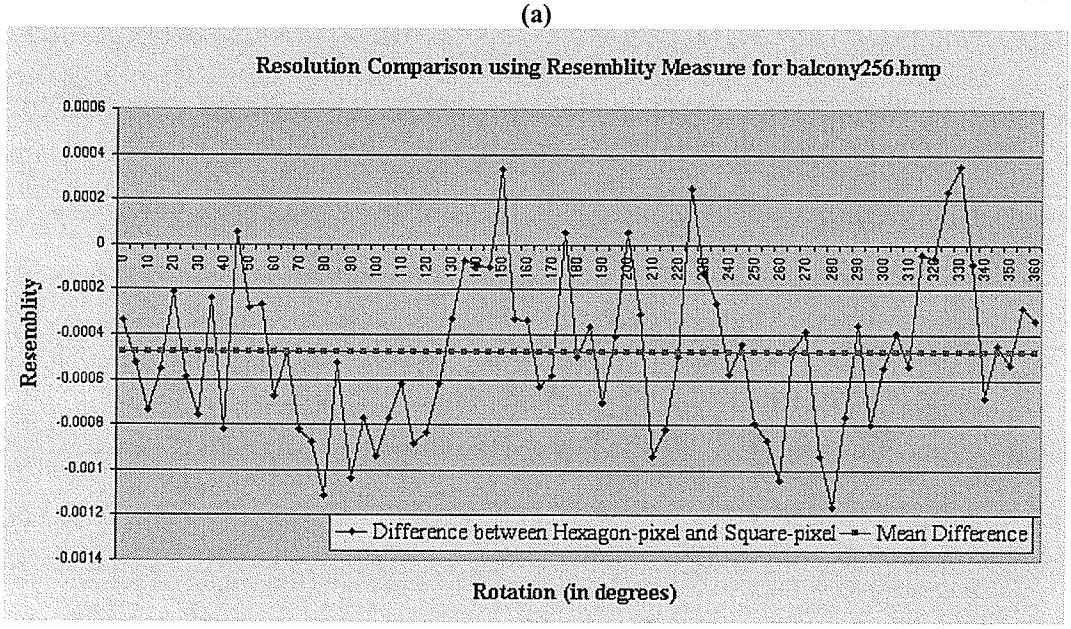
Quality	% Difference	2σ	Which is better?
Euclidean Distance	0.1034%	0.0315%	Hexagon
Resemblity	1.3768%	0.0012%	Hexagon
Entropy	0.1008%	0.0100%	Square

(d)

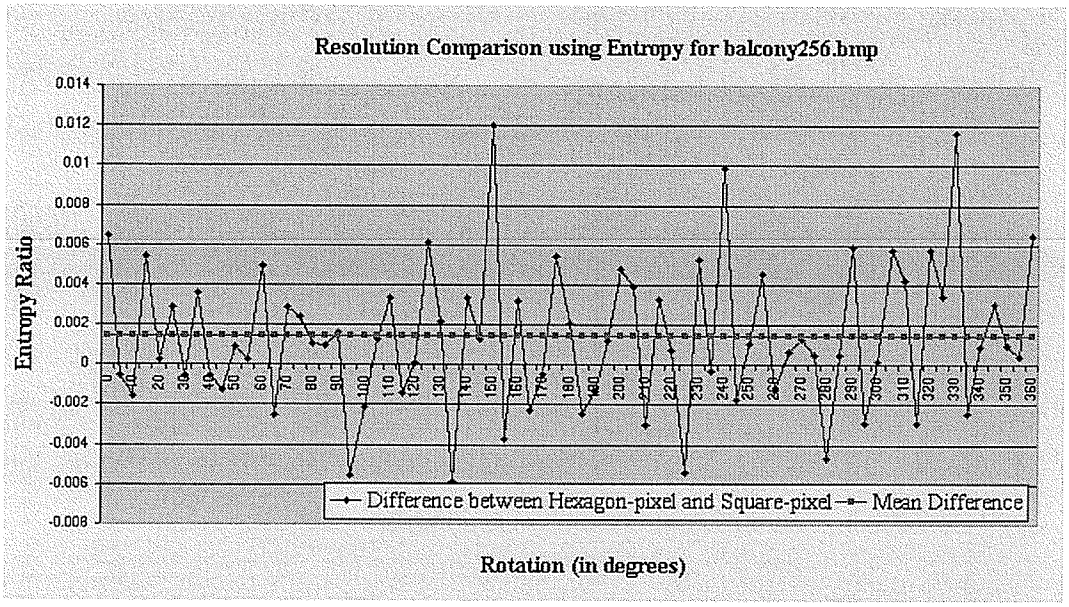
Figure 2-13. *Plots are shown for admin256.bmp image* ***(a)*** *Plot of Euclidean Distance Measure* ***(b)*** *Plot of Resemblity Measure* ***(c)*** *Plot of Entropy* ***(d)*** *Summary*


Resolution Comparison using Euclean Distance for balcony256.bmp
Normalized Euclidean Distance
0.003
0.002
0.001
0
-0.001
-0.002
-0.003
-0.004
Difference between Hexagon-pixel and Square-pixel
Mean Difference
Rotation (in degrees)

(a)


Resolution Comparison using Resemblity Measure for balcony256.bmp
Resemblity
0.0006
0.0004
0.0002
0
-0.0002
-0.0004
-0.0006
-0.0008
-0.001
-0.0012
-0.0014
Difference between Hexagon-pixel and Square-pixel
Mean Difference
Rotation (in degrees)

(b)



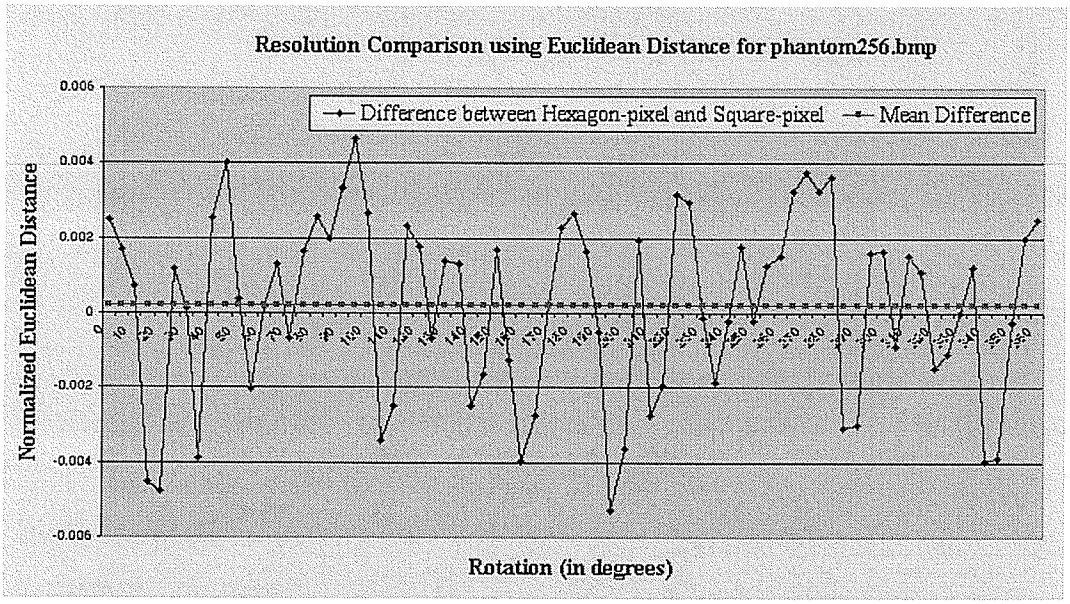
(c)

Summary (for balcony256.bmp)

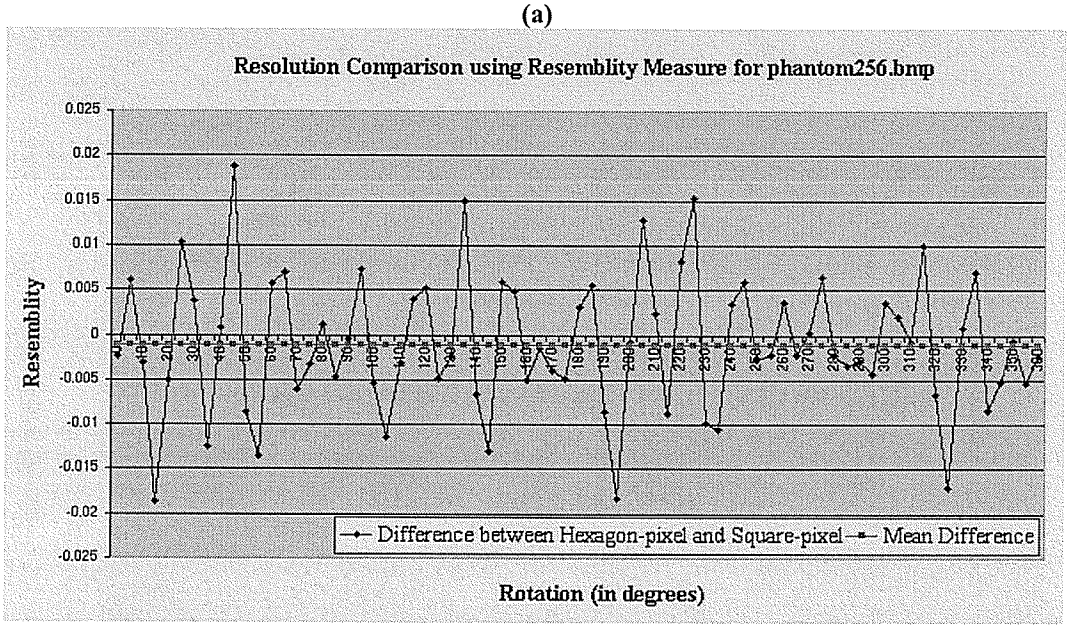
Quality	% Difference	2σ	Which is better?
Euclidean Distance	0.6181%	0.0403%	Hexagon
Resemblity	0.0483%	0.0010%	Hexagon
Entropy	0.1431%	0.0101%	Square

(d)

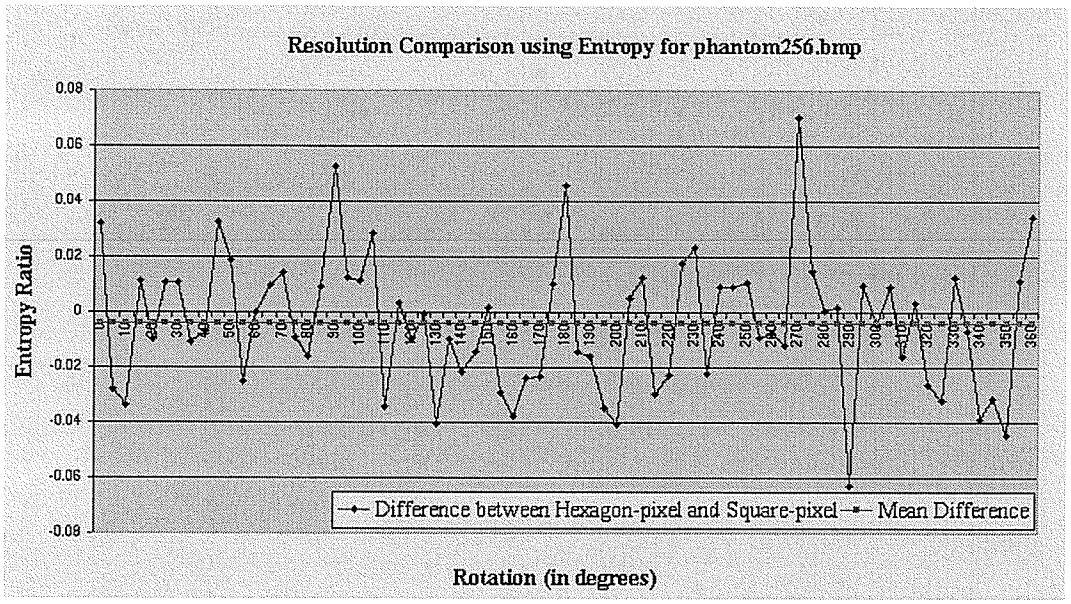
Figure 2-14. *Plots are shown for balcony256.bmp image* ***(a)*** *Plot of Euclidean Distance Measure* ***(b)*** *Plot of Resemblity Measure* ***(c)*** *Plot of Entropy* ***(d)*** *Summary*


Resolution Comparison using Euclidean Distance for phantom256.bmp
Difference between Hexagon-pixel and Square-pixel
Mean Difference
Normalized Euclidean Distance
0.006
0.004
0.002
0
-0.002
-0.004
-0.006
Rotation (in degrees)

(a)


Resolution Comparison using Resemblity Measure for phantom256.bmp
Resemblity
0.025
0.02
0.015
0.01
0.005
0
-0.005
-0.01
-0.015
-0.02
-0.025
Difference between Hexagon-pixel and Square-pixel
Mean Difference
Rotation (in degrees)

(b)



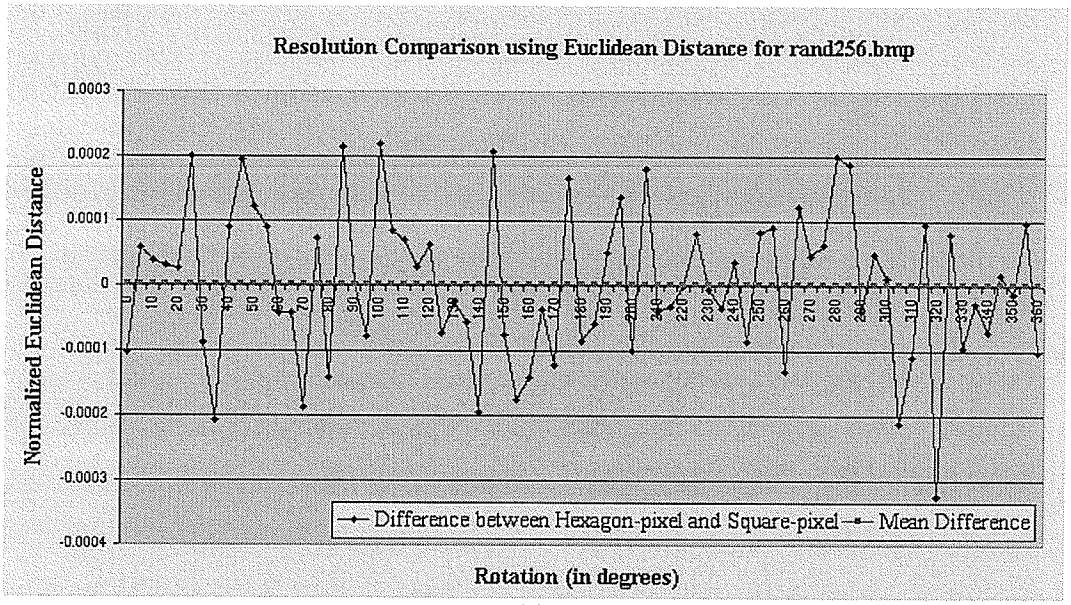
(c)

Summary (for phantom256.bmp)

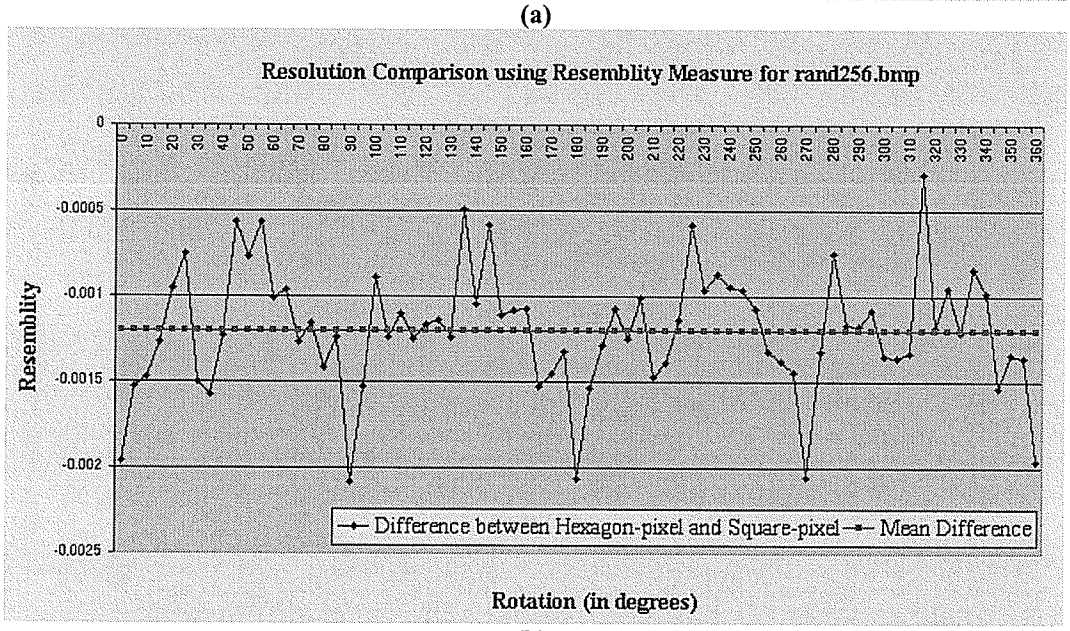
Quality	% Difference	2σ	Which is better?
Euclidean Distance	0.2351%	0.0723%	Square
Resemblity	0.1264%	0.0251%	Hexagon
Entropy	0.2723%	0.0463%	Hexagon

(d)

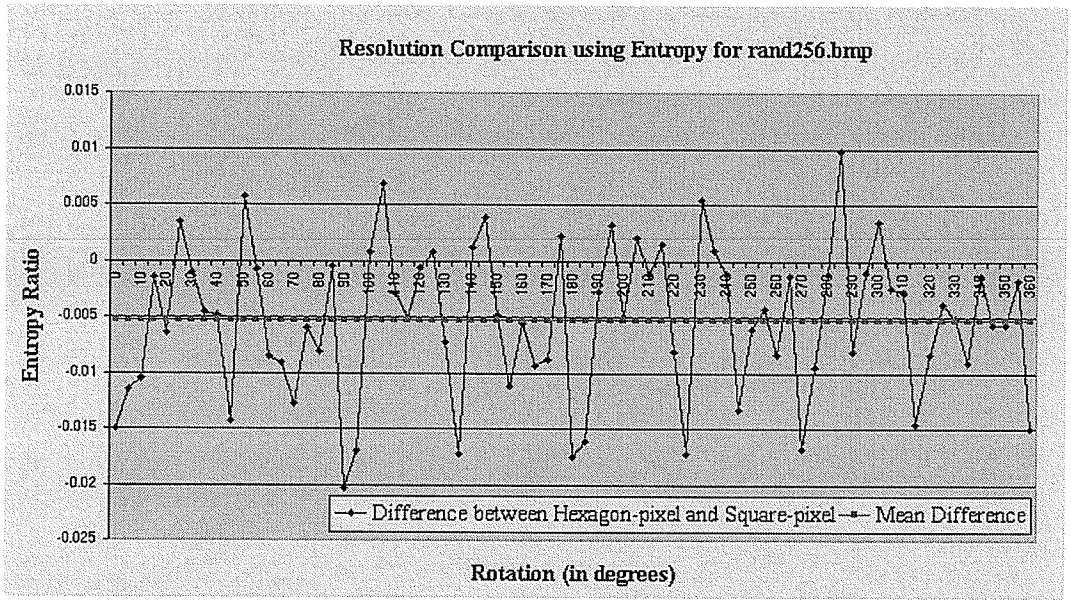
Figure 2-15. *Plots are shown for phantom256.bmp image* ***(a)*** *Plot of Euclidean Distance Measure* ***(b)*** *Plot of Resemblity Measure* ***(c)*** *Plot of Entropy (d) Summary*


Resolution Comparison using Euclidean Distance for rand256.bmp
Normalized Euclidean Distance
0.0003
0.0002
0.0001
0
-0.0001
-0.0002
-0.0003
-0.0004
Difference between Hexagon-pixel and Square-pixel
Mean Difference
Rotation (in degrees)

(a)


Resolution Comparison using Resemblity Measure for rand256.bmp
Resemblity
0
-0.0005
-0.001
-0.0015
-0.002
-0.0025
Difference between Hexagon-pixel and Square-pixel
Mean Difference
Rotation (in degrees)

(b)



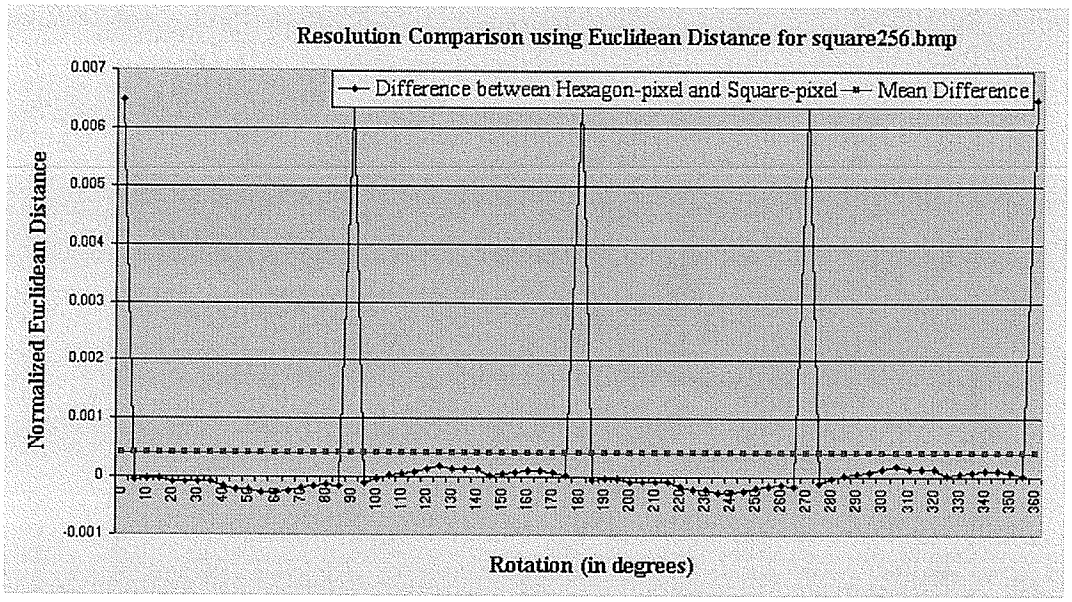
(c)

Summary (for rand256.bmp)

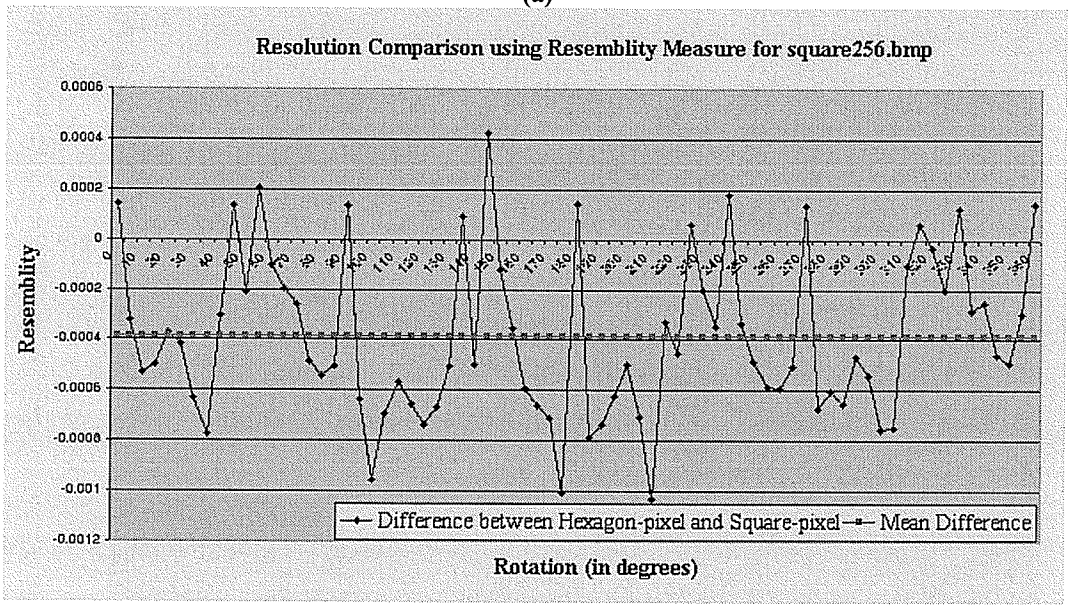
Quality	% Difference	2σ	Which is better?
Euclidean Distance	0.0021%	0.0020%	Square
Resemblity	0.1331%	0.0011%	Hexagon
Entropy	0.7402%	0.0254%	Hexagon

(d)

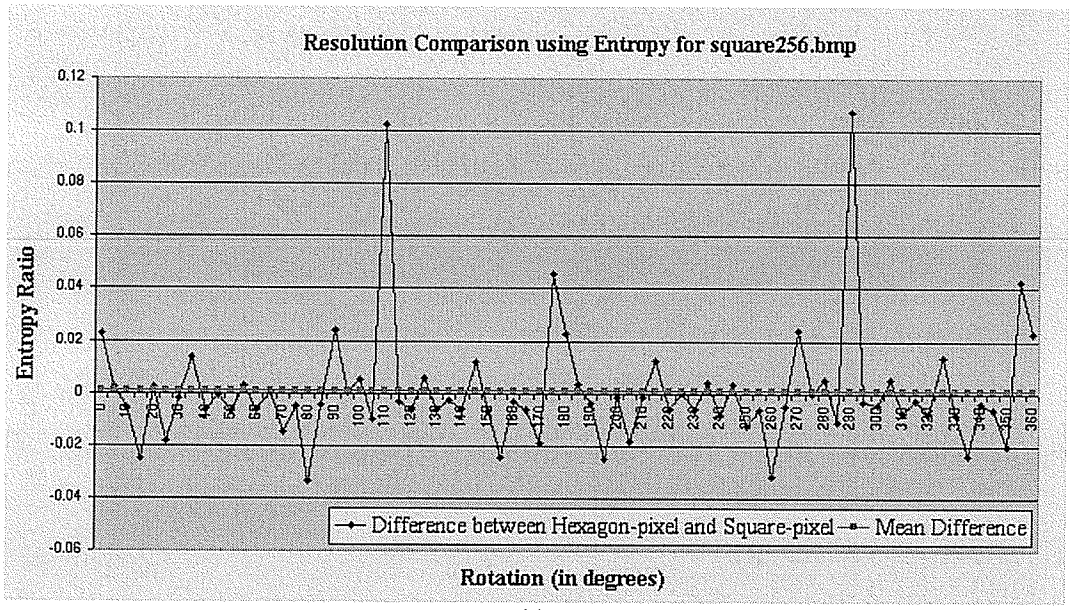
Figure 2-16. *Plots are shown for rand256.bmp image* ***(a)*** *Plot of Euclidean Distance Measure* ***(b)*** *Plot of Resemblity Measure* ***(c)*** *Plot of Entropy* ***(d)*** *Summary*


Resolution Comparison using Euclidean Distance for square256.bmp
Difference between Hexagon-pixel and Square-pixel
Mean Difference
Normalized Euclidean Distance
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
-0.001
Rotation (in degrees)

(a)


Resolution Comparison using Resemblity Measure for square256.bmp
Resemblity
0.0006
0.0004
0.0002
0
-0.0002
-0.0004
-0.0006
-0.0008
-0.001
-0.0012
Difference between Hexagon-pixel and Square-pixel
Mean Difference
Rotation (in degrees)

(b)



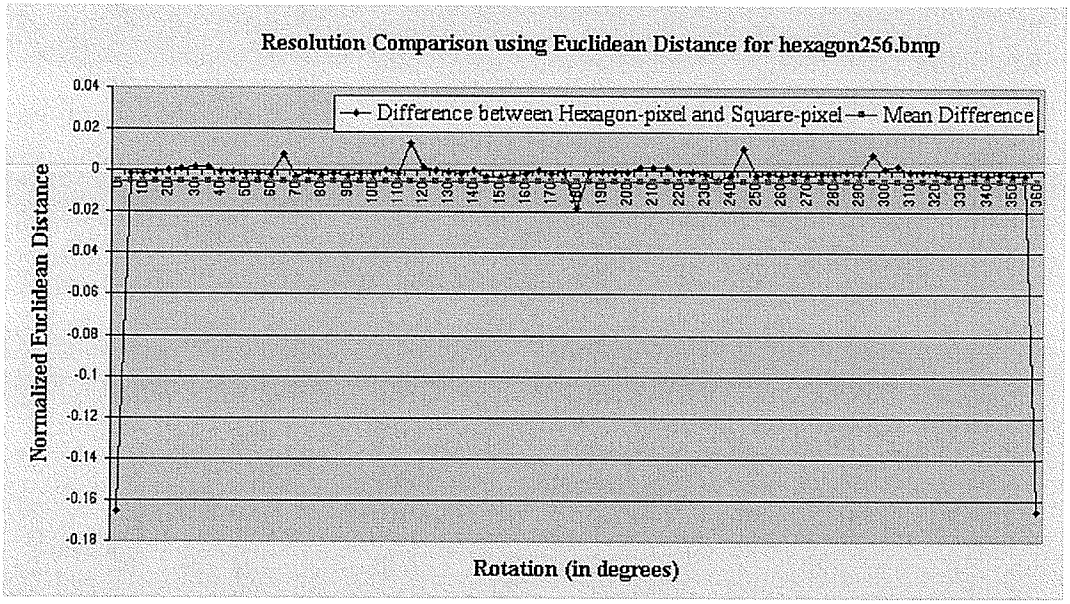
(c)

Summary (for square256.bmp)

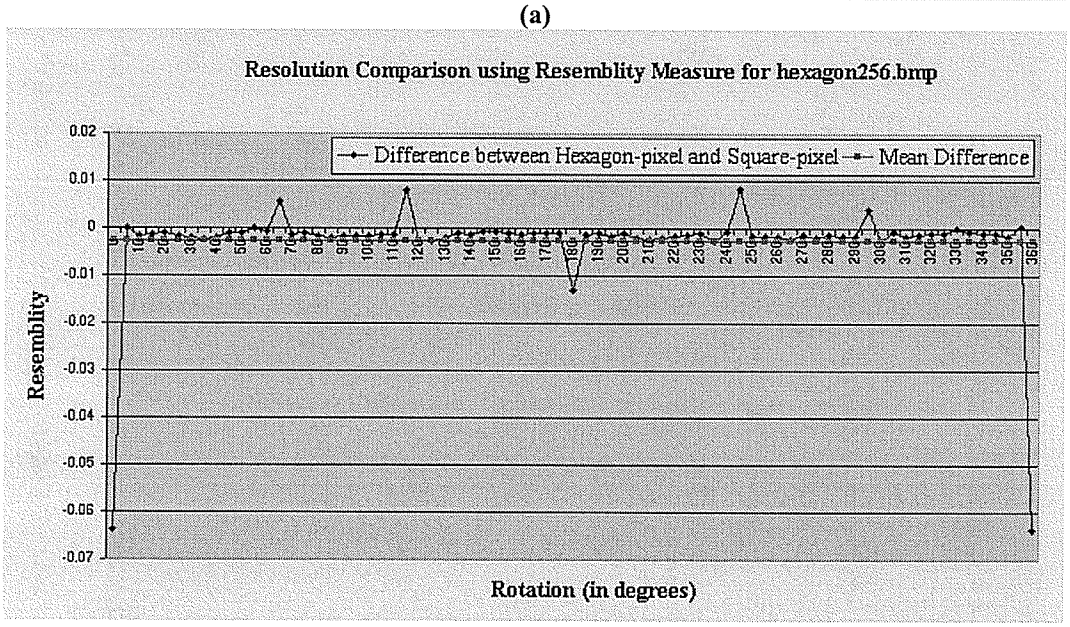
Quality	% Difference	2σ	Which is better?
Euclidean Distance	6.8704%	0.7570%	Square
Resemblity	0.0390%	0.0009%	Hexagon
Entropy	0.0823%	0.0608%	Square

(d)

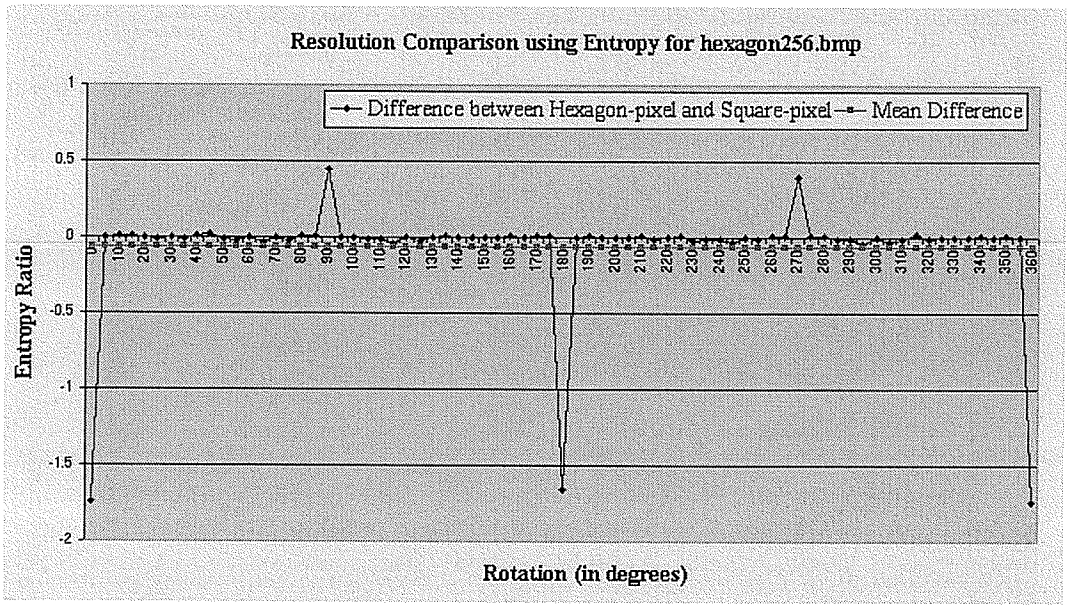
Figure 2-17. *Plots are shown for square256.bmp image* ***(a)*** *Plot of Euclidean Distance Measure* ***(b)*** *Plot of Resemblity Measure* ***(c)*** *Plot of Entropy* ***(d)*** *Summary*


Resolution Comparison using Euclidean Distance for hexagon256.bmp
Difference between Hexagon-pixel and Square-pixel
Mean Difference
Normalized Euclidean Distance
Rotation (in degrees)

(a)


Resolution Comparison using Resemblity Measure for hexagon256.bmp
Difference between Hexagon-pixel and Square-pixel
Mean Difference
Resemblity
Rotation (in degrees)

(b)



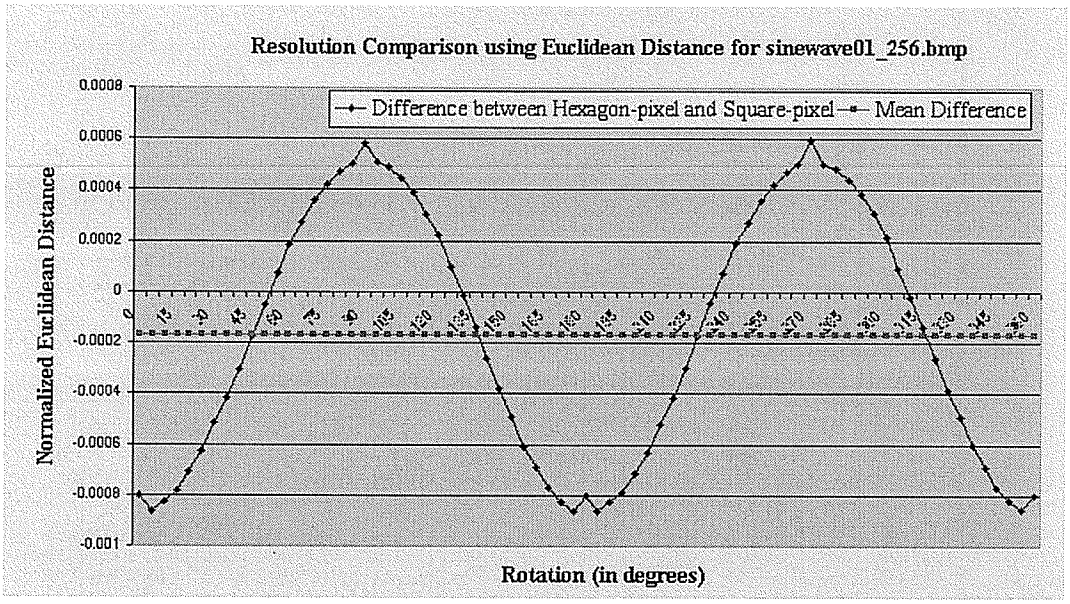
(c)

Summary (for hexagon256.bmp)

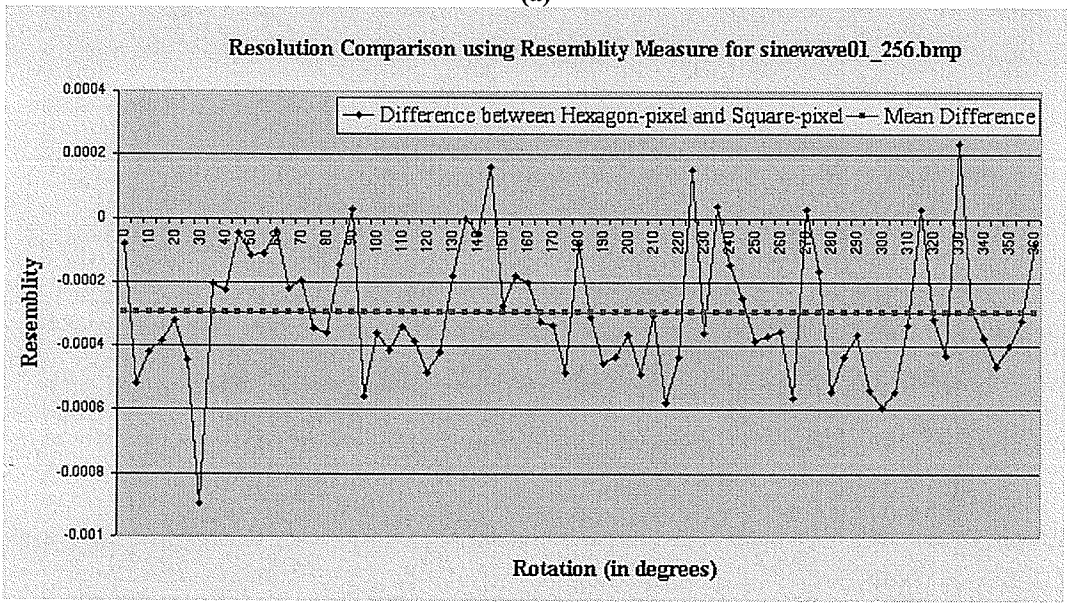
Quality	% Difference	2σ	Which is better?
Euclidean Distance	3.2687%	0.4801%	Hexagon
Resemblity	0.2986%	0.0309%	Hexagon
Entropy	4.5298%	0.7157%	Hexagon

(d)

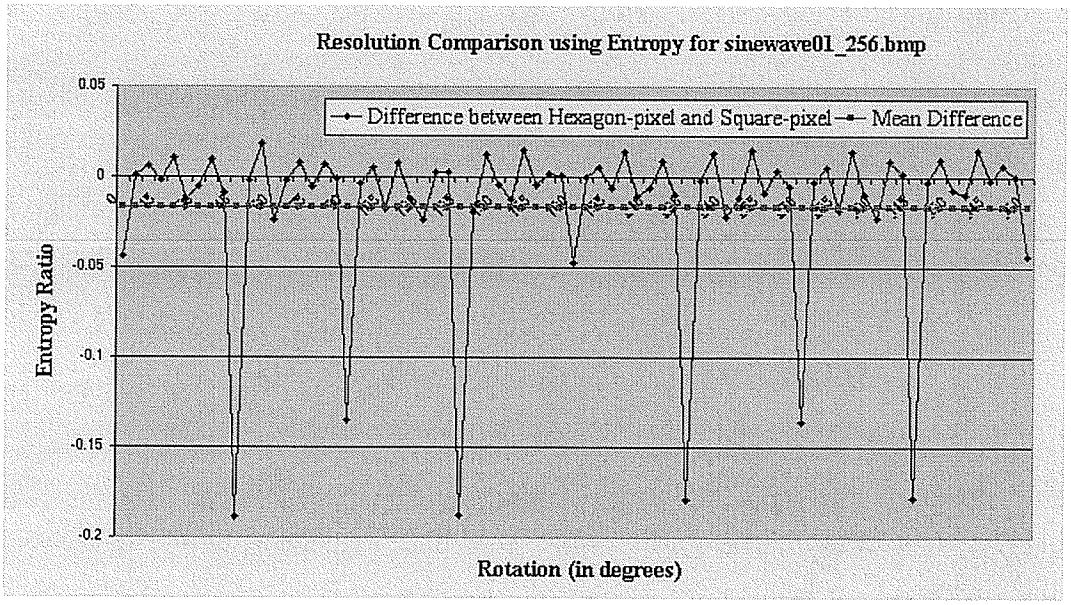
Figure 2-18. *Plots are shown for hexagon256.bmp image* ***(a)*** *Plot of Euclidean Distance Measure* ***(b)*** *Plot of Resemblity Measure* ***(c)*** *Plot of Entropy* ***(d)*** *Summary*


Resolution Comparison using Euclidean Distance for sinewave01_256.bmp
Difference between Hexagon-pixel and Square-pixel
Mean Difference
Normalized Euclidean Distance
0.0008
0.0006
0.0004
0.0002
0
-0.0002
-0.0004
-0.0006
-0.0008
-0.001
Rotation (in degrees)

(a)


Resolution Comparison using Resemblity Measure for sinewave01_256.bmp
Difference between Hexagon-pixel and Square-pixel
Mean Difference
Resemblity
0.0004
0.0002
0
-0.0002
-0.0004
-0.0006
-0.0008
-0.001
Rotation (in degrees)

(b)



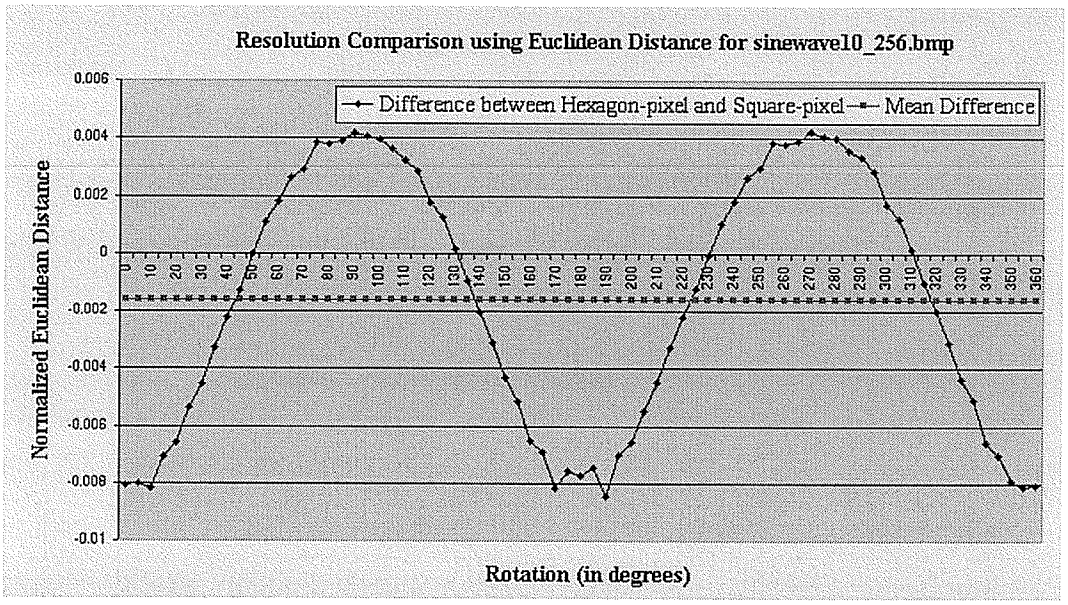
(c)

Summary (for sinewave01_256.bmp)

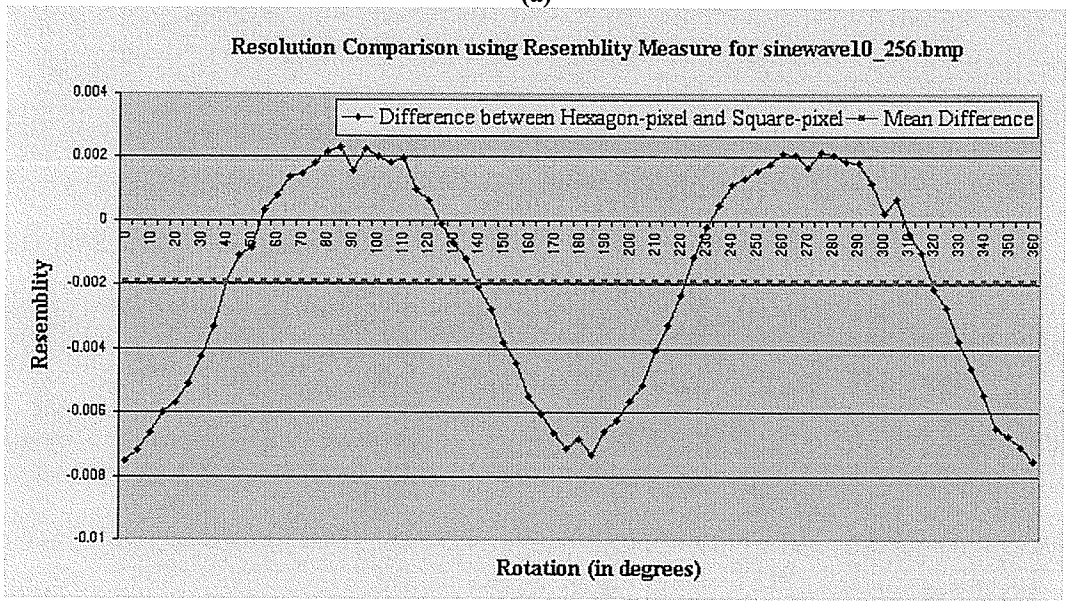
Quality	% Difference	2σ	Which is better?
Euclidean Distance	1.3260%	0.1069%	Hexagon
Resemblity	0.0295%	0.0006%	Hexagon
Entropy	1.6943%	0.1323%	Hexagon

(d)

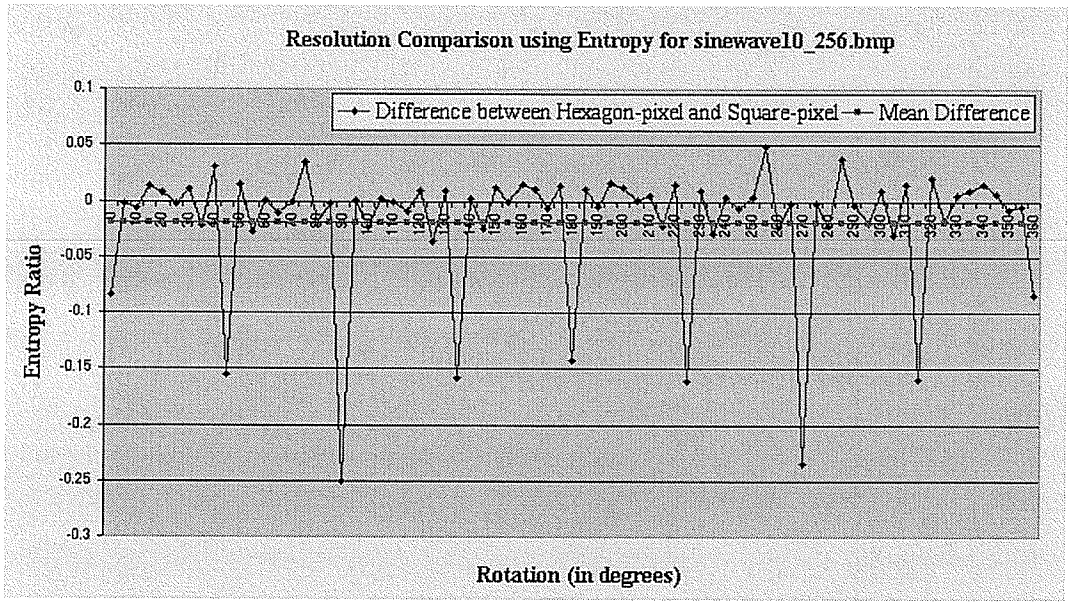
Figure 2-19. *Plots are shown for sinewave01_256.bmp image* ***(a)*** *Plot of Euclidean Distance Measure* ***(b)*** *Plot of Resemblity Measure* ***(c)*** *Plot of Entropy* ***(d)*** *Summary*


Resolution Comparison using Euclidean Distance for sinewave10_256.bmp
Difference between Hexagon-pixel and Square-pixel
Mean Difference
Normalized Euclidean Distance
Rotation (in degrees)

(a)


Resolution Comparison using Resemblity Measure for sinewave10_256.bmp
Difference between Hexagon-pixel and Square-pixel
Mean Difference
Resemblity
Rotation (in degrees)

(b)



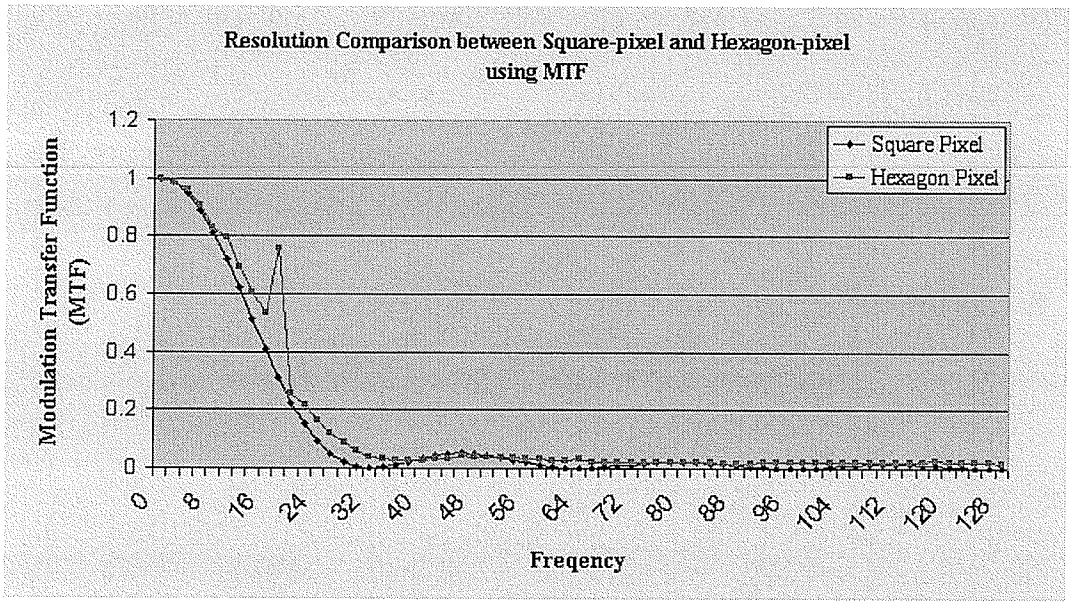
(c)

Summary (for sinewave10_256.bmp)

Quality	% Difference	2σ	Which is better?
Euclidean Distance	1.2471%	0.0958%	Hexagon
Resemblity	0.2065%	0.0098%	Hexagon
Entropy	1.9541%	0.1633%	Hexagon

(d)

Figure 2-20. *Plots are shown for sinewave10_256.bmp image* ***(a)*** *Plot of Euclidean Distance Measure* ***(b)*** *Plot of Resemblity Measure* ***(c)*** *Plot of Entropy* ***(d)*** *Summary*



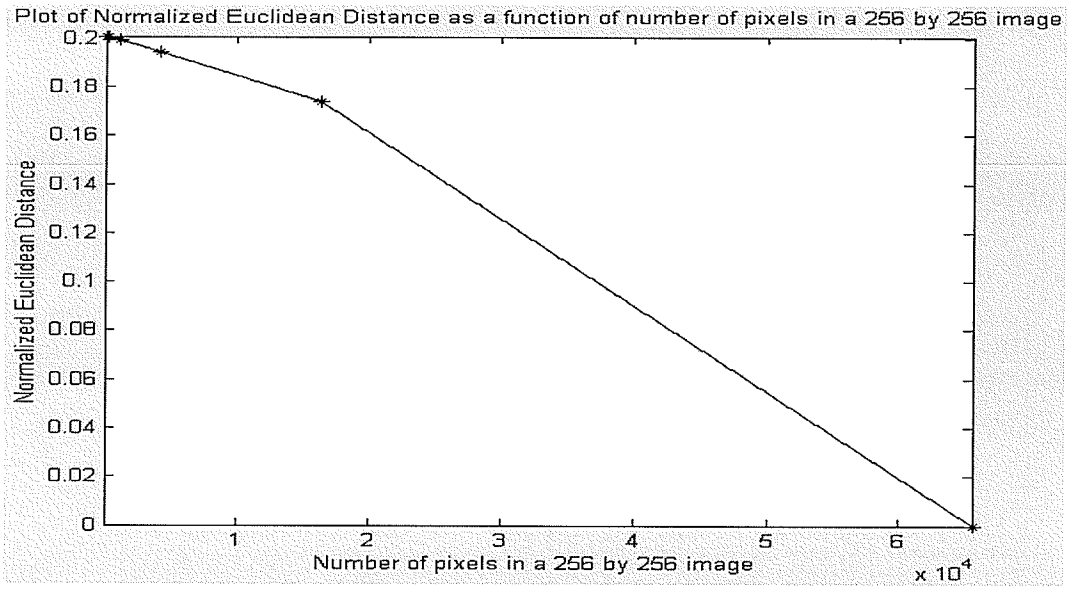
(a)

Summary

Quality	% Difference	2σ	Which is better?
MTF	3.1003%	0.1066%	Hexagon

(b)

Figure 2-21. *MTF plots* ***(a)*** *Plot of MTF* ***(b)*** *Summary*



(a)

Figure 2-22. *Plot of normalized Euclidean distance as a function of number of square pixels in a 256 by 256 image*

Chapter 3

Reconstruction (from Projections) Techniques

Note that this chapter is called Reconstruction from Projections because there are other approaches not based on reconstruction from projections (example - acoustic holography and techniques based on the transmission and reflection of ultrasonic waves [38]), to get three-dimensional information about structures within the body. Discussion of the same is beyond the scope of this thesis. This thesis will only talk about reconstruction from projection techniques.

3.1 Overview

This chapter reviews a few algorithms that have been proposed to solve the reconstruction problem. Each method has its own advantages and disadvantages. Workarounds have been discovered to eliminate some of the de-merits of the algorithms. This chapter does not talk about the de-merit removal techniques. The objective of this chapter is to give a good understanding of the different reconstruction techniques, which will help the reader in understanding the experiments discussed in chapter 4.

3.2 Statement of the Reconstruction Problem

Image reconstruction is one of the key components of Computed Tomography (CT). For limited dose, the projection data is limited and hence the reconstruction accuracy can only be improved by the development of more efficient detectors and the optimization of

reconstruction algorithms to make more efficient use of the available dose. The work in this thesis addresses the latter issue assuming that the detector quality has reached its limit. In CT this is referred to as the backward or inverse problem.

The problem of reconstructing two-dimensional (2-D) objects from a set of one-dimensional (1-D) projected images has arisen and been solved independently in fields ranging from medicine and electron microscopy to holographic interferometry. By using a source of radiation external to the object, one can obtain a transmission picture or projection of the 2-D object onto a 1-D detector array. The reconstruction problem is: Given a subset of all possible projections of an object, estimate its internal density distribution [37]. This is illustrated diagrammatically in Figure 3-1.

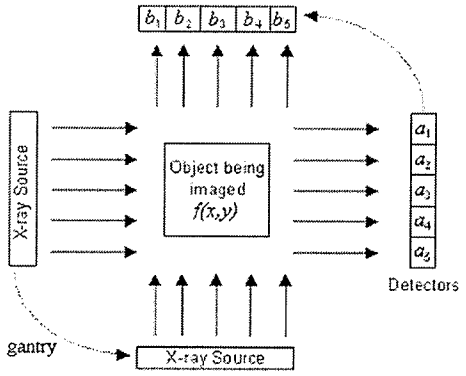


Figure 3-1a. *Graphical representation of the mathematical model of forward CT. The X-ray source and detector are rotated around the object and the detector readings are recorded at each angle.*

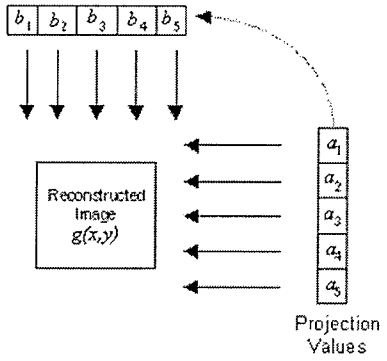


Figure 3-1b. *Graphical representation of the mathematical model of backward CT. The projection values are used to reconstruct back the image. Note that more the angles, the better the resolution of the reconstructed image, but higher would be the dose.*

All algorithms for reconstruction take as input the projection data, and all produce as output an estimate of the original structure based on the available data. The estimate varies from method to method. The relative performance of the various methods depends

on the object and how the data is collected. It is therefore important that qualitative judgments be made only after a careful and exhaustive study.

Three known reconstruction algorithms are widely known. They are –

a) Summation: The ray sums of the rays through each point are simply added to obtain an estimate of the density at the point.
b) Use of Fourier Transforms: It is possible to derive reconstruction algorithms using the Fourier Slice Theorem (also called the Projection Theorem) (explained in section 3.4).
c) Iterative Method

3.3 Summation Reconstruction Technique (SRT)

A rough but nonetheless elegant method [38] of obtaining an approximate reconstruction is the summation method. In the summation method the density of each point in the reconstructed picture is obtained by adding up the densities of all the rays going through that point. For example, if the test picture consists of a single point and two projections of it are made, the reconstruction is a four-pointed cross as shown in Figure 3-3. The cross demonstrates the roughness of the summation method. An exact method of reconstruction would reconstruct a point as a point and not as a cross. It should be mentioned, however, that a method that succeeds in reconstructing a single point as a point is not necessarily an exact method for reconstructing more complex pictures.

For the purpose of obtaining three-dimensional information that is quantitatively accurate for medical applications there are two major objections to employing the summation

method for the reconstructions. First, the reconstruction is inaccurate because every point in the original is blurred in the reconstruction. Second, if we take the reconstruction and calculate its projections, we find that they are not the same as the projections of the original picture.

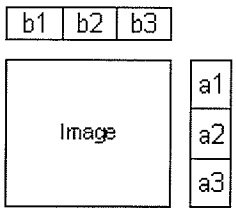


Figure 3-2a. *The projections of the image are taken.*

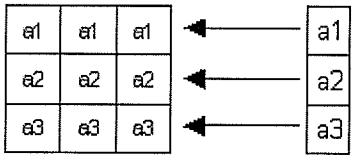


Figure 3-2b. *The projections at the first angle are back-projected such that each pixel in the grid that contributes to the projection is assigned a equal value as that of the projection.*

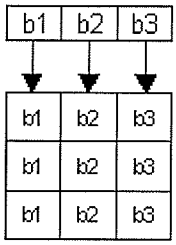


Figure 3-2c. *The procedure is repeated for all angles and the results are added up.*

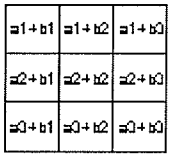


Figure 3-2d. *Reconstructed Image based on SRT.*

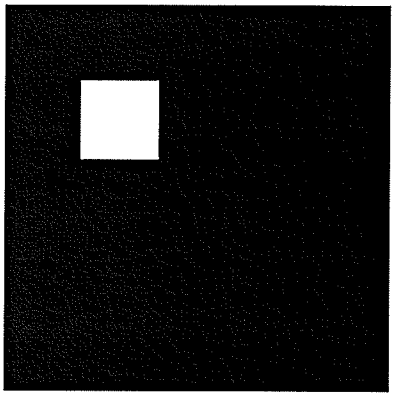


Figure 3-3a. *Original Image consists of a point near the top left corner. All the other values in the original image are 0.*

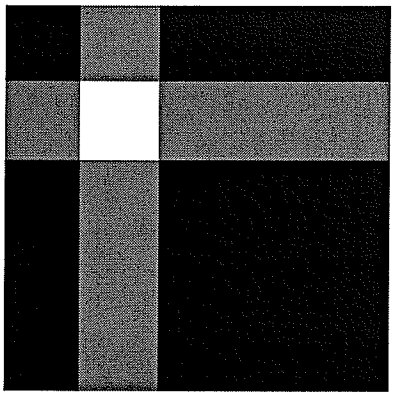


Figure 3-3b. *Reconstructed image reconstructed based on Summation Reconstruction Technique (SRT) using projections at 0° and 90°.*

3.4 Use of Fourier Transforms

There are various techniques that use Fourier Transforms to reconstruct images from projections. These techniques vary only slightly but they nevertheless have their own

names: Fourier Backprojection (FBP), which is the same as Convoluted Backprojection (CBP), Filtered Fourier Backprojection (FFBP) and Filtered Convoluted Backprojection (FCBP). The FFBP is done by adding a filter like Butterworth and Hanning to smoothen the projection data.

It is possible to derive reconstruction algorithms using the Fourier Slice Theorem (also called the Projection Theorem) [42]. The Projection Theorem relates the three spaces (image or spatial space, Fourier or frequency space and projection space) we encounter in image reconstruction from projections. Considering a 2-D image, the theorem states, "the Fourier Transform of 1-D projections of the 2-D image is equal to the radial section (slice) of the 2-D Fourier Transform of the 2-D image at the angle of the projection". This is illustrated graphically in Figure 3-4.

When the projection data is transformed to the frequency domain, there is lot of statistical noise. This noise introduces artifacts in the reconstructed image. Hence the frequency space is smoothened by the application of filters. The most commonly used filters are Butterworth, Hanning, Weiner, and low pass cosine filter. The filters have slightly different characteristics [43]. Regardless of the filter used, the end result is to display a final image that is relatively free from noise and is pleasing to the eye. Figure 3-5 shows the result of the application of the FBP approach in image reconstruction.

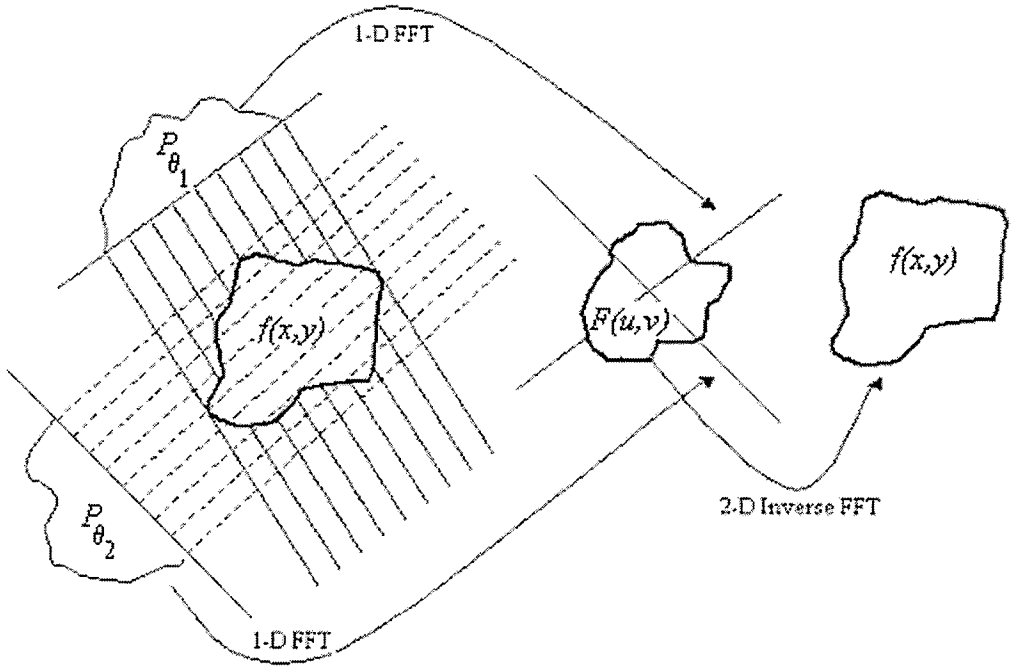
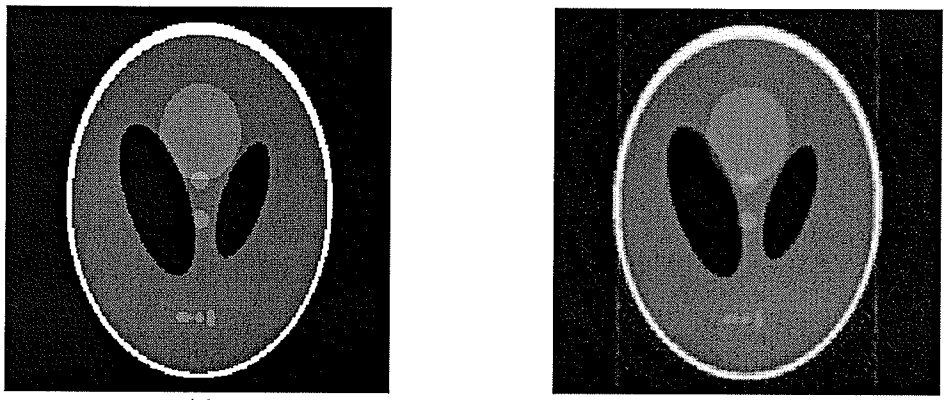


Figure 3-4. *Graphical illustration of the Fourier slice theorem [42]. f(x,y) is the original image. Projections are taken at angles θ_1 and θ_2. The projection values are called $P_{\theta 1}$ and $P_{\theta 2}$ respectively. 1-D Fourier transforms of these projections are taken and are backprojected in the Fourier space. This gives us F(u,v). Taking the inverse Fourier transform of F(u,v) gives us f(x,y).Note that f(x,y) is in the spatial space, $P_{\theta 1}$ and $P_{\theta 2}$ is in the projection or Radon space and F(u,v) is in the Fourier space.*



(a) **(b)**

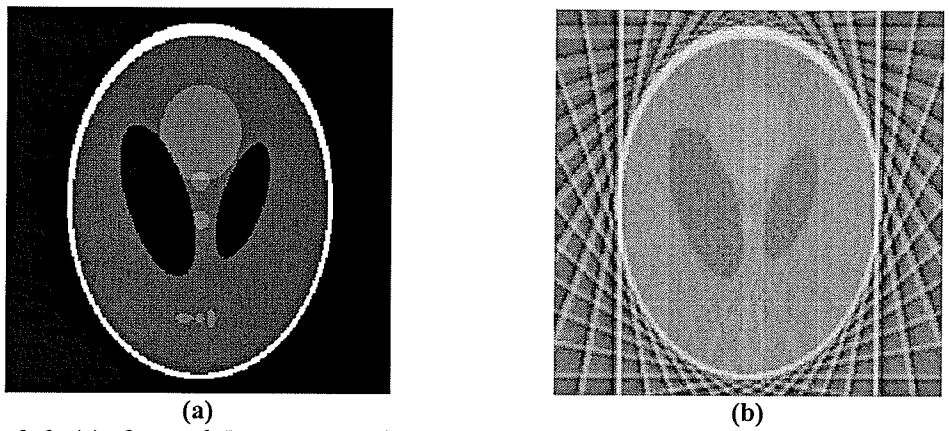
Figure 3-5. (a) *Original Image – 256 by 256 brain phantom image (generated in Matlab)* **(b)** *Reconstructed Image based on projections from 0 to 180 degrees intervals of 1 degree. Hann filter is used to smoothen the sinogram.The code is available in Matlab™ under the help section of the iradon function.*

Algorithm of Filtered Backprojection Technique

1. Measure Projection.

2. Compute the filtered projection
3. Backproject the filtered projection.
4. Repeat steps 1, 2 and 3 for all projection angles.

This technique is good if we have the projections in as many angles as possible (0 to 180), however this is not good for reducing dose. Figure 3-6 shows an example of a reconstruction with projection angles from 0 to 180 in intervals of 10 degrees.



(a) (b)

Figure 3-6. (a) *Original Image – 256 by 256 brain phantom image (generated in Matlab)* **(b)** *Reconstructed Image based on projections from 0 to 180 degrees intervals of 10 degrees. Hann filter is used to smoothen the sinogram. The code is available in Matlab™ under the help section of the iradon function.*

As one can see from Figures 3-5 and 3-6, the quality of the reconstruction deteriorates as soon as the available projection data is reduced.

3.5 Iterative Method

In the case of sparse projection data, the equation system for reconstructing the image may be underdetermined, i.e., the number of equations is less than the number of unknowns, which is usually the case in PET, SPECT, and sometimes in CT. In this case

the solution by inversion methods is hampered both by the size of the equation system and the inconsistencies caused by the inherent noise in the acquired projection data and the approximate description of the weight factors. Thus, a method devised by Gordon, Bender, and Herman [39] can be used: Iteratively, for each projection image ray, the grid is projected, the projection is compared with the corresponding ray value in the acquired projection image, and a correction term is computed and back-projected onto the grid. Ideally, each back-projection updates the grid to correspond more closely to the acquired projection data. The concept of iterative method can be understood by the method devised by Kaczmarz to solve linear simultaneous equations [26]. This is illustrated in Figure 3-7.

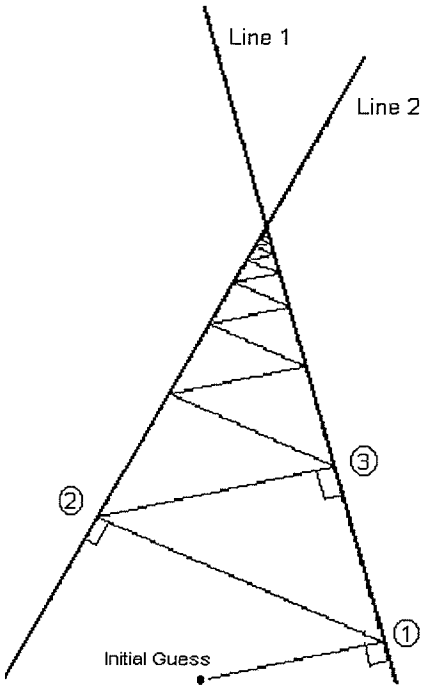


Figure 3-7. *Graphical illustration of Kaczmarz's approach for solving equations of two lines.*

The lines (line1 and line2) in Figure 3-7 represent two equations. The solution of these two equations is the intersection of the two lines. One starts with an initial guess ($S^{(0)}$). It can be shown that a correction to this initial guess is equivalent to dropping a line perpendicular to one of the lines (say line1) from the initial guess point. The point of intersection of this perpendicular and the line is our new solution (point 1 in Figure). This new point is closer to the actual solution than our initial guess. Now starting at point1, a perpendicular is drawn on line2. The point of intersection is point2, which is closer to the actual solution than point1. We then select the first line and perform this procedure again. The more the iterations the closer the solution will be to the actual solution. The iterative process is terminated when some convergence-rate threshold is reached.

It was shown by Andersen and Kak [1] that noise-like artifacts in the reconstruction can be reduced if the grid is corrected only once per projection image and not for every projection ray. Note that for the remainder of this thesis, a *cycle* constitutes a sequence of grid corrections in which all available projections are utilized exactly once and an *iteration* constitutes a sequence representing the number of times the grid is corrected. Although conceptually this approach is much simpler than the transform-based methods discussed in section 3-4, for medical applications it lacks accuracy and speed of computation [4]. However, there are situations where it is not possible to measure a large number of projections, or the projections are not uniformly distributed over 180° both these conditions being necessary requirements for the transform-based techniques to produce results with the accuracy desired in medical imaging [4]. Problems of this type are sometimes more amenable to solution by iterative techniques. Iterative techniques are also useful when the energy propagation paths between the source and receiver positions

are subject to ray bending on account of refraction, or when the energy propagation undergoes location dependent attenuation along ray paths as in emission CT.

In most literature, I found the formula confusing because the 1-D (projection-space) and 2-D (spatial space) variables were used ambiguously; I modified the representation of the formula to make it easier to understand and straightforward to code. To understand the formulae see Figure3-8 and understand the notations.

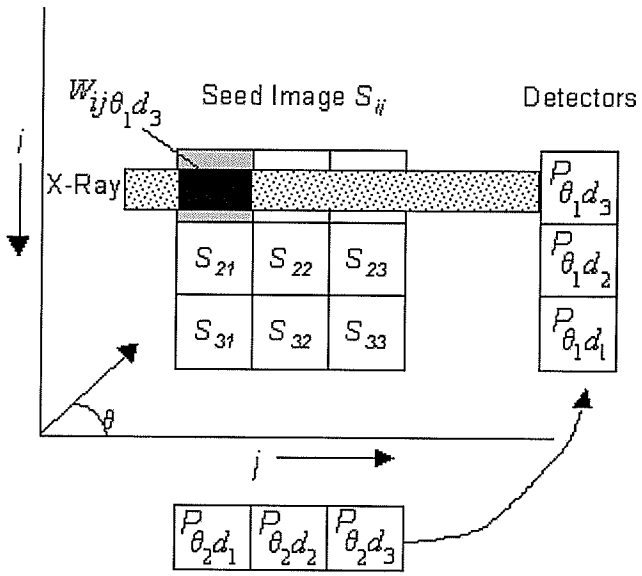


Figure 3-8. *Figure to help understand the formulae for ART, MART, SIRT and SART*

In this thesis, θ by itself represents the measure of the angle, whereas θ with subscript (example θ_1) or θ following a index (example θ1) indicates the index of the angle, example θ_1 is the first angle, θ_2 is the second angle and so on. Similarly d with subscript indicates the detector number, example d_1 indicates the first detector, d_2 indicates the second detector and so on.

- **Original Image (*O*)**

  Let (i,j) be the row and column number of the image respectively. O_{ij} represents the pixel value of the original image at (i,j) location. The size of the image is n *by* $n = N$.

- **Seed Image (*S*)**

  Let (i,j) be the row and column number of the image. Therefore *Sij* represents the pixel value of the seed image at (i,j) location. In the formulae given (i,j) are synonymous with (a,b), except they represent different algorithmic loops.

- **Projection Data (*P*)**

  If the projections are taken at an angle θ, the projection value measured on a detector is given by $P_{\theta d}$. M represents the total number of projection angles or if the gantry is moved, M is the number of detectors times the number of positions. T represents the total number of detectors.

- **Weight Matrix (*W*)**

  The weight matrix stores the contribution each pixel makes to each detector at all the projection angles. Hence the weight matrix is 4 dimensional. $W_{ij\theta_1 d_3}$ represents the contribution of pixel (i,j) to detector d_3 at projection angle θ_1.

- **Coordinate System**

  The programs are written in Matlab™. Since Matlab's geometry is different from Cartesian geometry, I made the Cartesian geometry equations fit Matlab's system. i represents the row number of the image and runs from top to bottom. j is the column number of the image and runs from left to right. The origin is (1,1) and

not (0,0) as in Cartesian geometry. θ is the angle made by the x-ray with the horizontal.

- **Other**

  K is the total number of cycles. *Cycle* signifies the completion of the consideration of all pixel values on all detectors at all projection angles. k is the current cycle. (k-1) is the previous cycle. *Iteration* signifies the completion of the consideration of all pixel values on a particular detector at one projection angle. In case of ART and MART, the seed image used to start the iteration is the one obtained after the end of the previous iteration. In case of SIRT and SART the seed image used to start the iteration is the one obtained after the end of the previous cycle. Hence it is necessary to declare two more variables. P is the total number of iterations (applies to ART and MART only). p is the current iteration and (p-1) is the previous iteration.

- **Basic Algorithm of Iterative Method**

  a) Start with a seed image.

  b) Copy the seed image into another variable called reconstructed image.

  c) Calculate the projections. Compare the projection value of the reconstructed image with that of the original image.

  d) Calculate how each pixel in the reconstructed image needs to be adjusted to get projection values same as that of the original. The adjustments are calculated differently based on the iterative method (ART, MART, SIRT and SART) used.

e) Apply the adjustments to the reconstructed image.

f) Applying adjustments for one projection data will typically throw off the values for another projection. Hence there is a need for iteration.

g) Go back to step b and repeat steps b-f. This procedure is repeated until a satisfactory convergence criterion is met.

3.5.1 Additive Algebraic Reconstruction Technique (AART or simply ART)

In many ART implementations the correction to the $(i,j)^{th}$ image cell is written as in Equation 1.

$$\left(\frac{P_{\theta d} - \sum_{a,b} W_{ab\,\theta d}\, S_{ab}^{(p-1)}}{\sum_{a,b} l\left(W_{ab\,\theta d}\right)} \right) \qquad (1)$$

where $l(x) = 1$ if $x > 0$

$= 0$ otherwise

where $\sum_{ab} l(W_{ab\,\theta d})$ represents the number of image cells which are contributing to the d^{th} detector at projection angle θ. This approximation is easier to implement. However it introduces artifacts in the reconstructed images. ART reconstructions suffer from salt and pepper noise, which is caused by the inconsistencies introduced in the set of equations by the approximations used for the weight factor [4]. The effect of such inconsistencies is worsened by the fact that as each equation corresponding to a ray in a projection is taken up, it changes some of the pixels just altered by the preceding equation in the same

projection. To reduce the effects of this noise in ART reconstructions, relaxation factors are commonly used. The relaxation factor λ is less than 1. In some cases, the relaxation factor is made as a function of the iteration number; that is, it becomes progressively smaller with increase in the number of iterations [4]. The resulting improvements in the quality of reconstruction are usually at the expense of convergence rate.

The formula for ART is given in equation 2-

$$\underbrace{\underbrace{\left[S_{ij}^{(p)} = S_{ij}^{(p-1)} + \lambda \left(\frac{P_{\theta d} - \sum_{a,b} W_{ab\theta d} S_{ab}^{(p-1)}}{\sum_{a,b} l\left(W_{ab\theta d}\right)} \right) \right]}_{d=d_1 \text{ to } d_T}}_{\theta=\theta_1 \text{ to } \theta_M} \tag{2}$$

3.5.2 Multiplicative Algebraic Reconstruction Technique (MART)

Most iterative methods encounter problems in determining the areas with lower density than the surrounding; MART comes in handy especially in images like the brain phantom (Figure 2-2c). Generally all iterations start with a homogeneous first estimate (seed image). The selection of the seed image is of main importance for the applicability of a method and for the number of iterations. Researchers [22] have reconstructed the first estimate from the Backprojection of projections where each element is assigned the minimum of the backprojected data. This modification provides the advantage that a cell content once set to 0 cannot increase. The problem with all methods is that there is no confirmed mathematical or physical reason to set particular pixel elements to zero, except for MART where this is done at least for all the pixels along a ray from one of the

detectors which showed zero content in the image. Because the total intensity within the images has to correspond to the total source distribution, some pixels at the outer surface of the object contain intensity, which is then missing in the center pixels. This is also most probably the reason why tomography tends to overestimate the electron density on the topside of the reconstruction [47]

In MART implementations the correction to the $(i,j)^{th}$ image cell is as shown in Equation 3.

$$\left(\frac{P_{\theta d}}{\sum_{a,b} W_{ab\theta d} S_{ab}} W_{ij\theta d} \right) \tag{3}$$

This approximation is easier to implement. MART suffers the same noise problems as that of ART. A priori information like ART improves the situation, but the median filter [22] best improves the signal to noise ratio of the reconstruction. The knowledge about the nature of noise sources helps in improving results by proper filtering.

The formula for MART is given in equation 4.

$$\underbrace{\underbrace{\left[S_{ij}^{(p)} = S_{ij}^{(p-1)} \left(\frac{P_{\theta d}}{\sum_{a,b} W_{ab\theta d} S_{ab}} \right)^{\lambda} \right]}_{d=d_1 \text{ to } d_T}}_{\theta=\theta_1 \text{ to } \theta_M} \tag{4}$$

3.5.3 Simultaneous Iterative Reconstruction Technique (SIRT)

This approach uses the same formula as that of ART, except the change in the $(i,j)^{th}$ pixel isn't done immediately. In this technique before making the adjustment to the pixel value, all equations are considered and only then at the end of each cycle are the pixel values changed, the change for each pixel being the average value of all the computed changes for that pixel. The SIRT algorithm also suffers from the same inconsistencies as that of ART in the forward process (i.e, computation of the weight factor), but by eliminating the continual and competing pixel update as each new equation is taken up, it results in smoother reconstructions. This technique leads to better looking images than those produced by ART, at the expense of slower convergence [4].

The formula for SIRT is shown in Equation 5.

$$S_{ij}^{(k)} = S_{ij}^{(k-1)} + \frac{\lambda}{MT} \sum_{\theta=\theta_1}^{\theta_M} \sum_{d=d_1}^{d_T} \left(\frac{P_{\theta d} - \sum_{a,b} W_{ab\,\theta d} S_{ab}^{(k-1)}}{\sum_{a,b} l\left(W_{ab\,\theta d}\right)} \right) \tag{5}$$

where $l(x) = 1$ if $x > 0$

$= 0$ otherwise

3.5.4 Simultaneous Algebraic Reconstruction Technique (SART)

SART is a variation of ART, and it combines the best of ART and SIRT. This technique yields reconstructions of good quality and numerical accuracy in only one iteration [4]. The main features of SART include –

a) Reduction of errors in the approximation of ray integrals of a smooth image by finite sums.

b) Traditional pixel basis is abandoned in favor of bilinear elements.

c) For a circular reconstruction region, only partial weights are assigned to the first and last picture elements on the individual rays.

d) To further reduce the noise resulting from the unavoidable but now presumably considerably small inconsistencies with real projection data, the correction terms are simultaneously applied for all the rays in one projection; this is in contrast with the ray-by-ray updates in ART.

e) In addition, a heuristic procedure is used to improve the quality of reconstructions. A longitudinal Hamming window is used to emphasize corrections applied near the middle of a ray relative to those applied near its ends.

The formula for SART is given in Equation 6.

$$S_{ij}^{(k)} = S_{ij}^{(k-1)} + \lambda \sum_{\theta=\theta_1}^{\theta_M} \sum_{d=d_1}^{d_T} W_{ij\theta d} \left(\frac{P_{\theta d} - \sum_{ab} W_{ab\theta d} S_{ab}^{(k-1)}}{\sum_{a,b} W_{ab\theta d}} \right) \quad (6)$$

The last step, heuristic in nature consists of modifying the back distribution of correction terms by a longitudinal Hamming window. The idea of the window is illustrated in Figure 3-9. The uniform back distribution according to the coefficients $W_{ij\theta d}$ is replaced by a weighted version. The weighting correction term is $H_{ij\theta d}$ where $H_{ij\theta d}$ is given by –

$$H_{ij\theta d} = W_{ij\theta d}\ h_{ij}$$

where h_{ij} is the Hamming window in pixel (i,j).

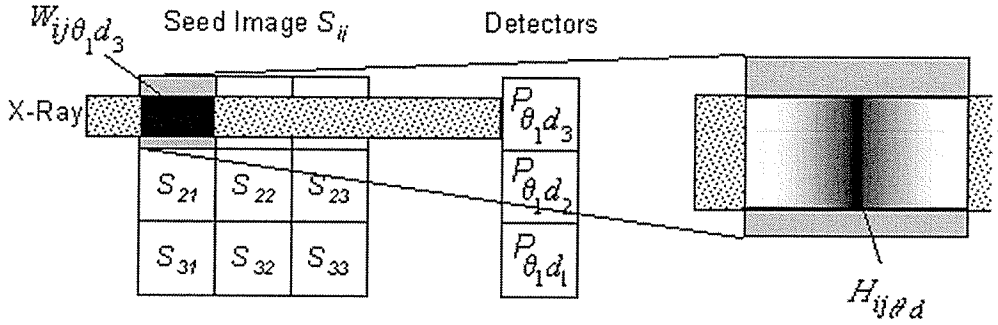


Figure 3-9. *Illustration of the Hamming window correction concept in SART.*

However SART is slightly slower than ART in software, due to the pixel based pooling of correctional updates [26].

3.6 Summary

In this chapter types of algebraic methods have been presented. In spite of the computational cost, algebraic methods have several advantages like –

a) Different ray geometries are easy to implement.

b) Possible to provide a priori knowledge about the reconstructed object in the algorithm.

c) Fewer projections than for the analytical methods are required which is proved mathematically [26].

d) Metal streaking artifacts are reduced

e) It is possible to handle detectors of variable size inside projections, provided that detector geometry remains unchanged from one projection to another [14]

f) Better reconstruction technique for low dose CT imaging.

I decided to do the experiments on *ART to see the effect of the different parameters on the reconstruction quality. In the next chapter, the experiments performed to analyze the different factors that affect *ART are discussed.

Chapter 4

Experiments on *ART

4.1 Overview

In case of reconstructions from highly underdetermined equations, Algebraic Reconstruction Techniques (*ART like AART, MART, SIRT and SART) prove very helpful. The reconstruction quality of each of the techniques is dependent on many factors. This chapter investigates some of the factors affecting the performance and quality of the *ART reconstruction and discusses the results.

4.2 Parameters affecting *ART

The different parameters that affect *ART are explained in detail in chapter 1. Below is a summary of the parameters that I experimented with and a brief description regarding how I used them for my experiments.

1) **Nature of the original image:** I did my experiments with 9 test images to make sure that the results are consistent across all the images and are not partial to any kind of image. The different test images used for the experiments are shown in Figure 4-1 and the reasoning for using each of them is provided in section 4.3.
2) **Ray Width:** In my experiments the ray width is always 1 pixel.
3) **Ray Gap:**. For all my experiments I consider the ray gap as equal to 0.
4) **Detector Width:** For all my experiments I consider the detector width as equal to 1 pixel.
5) **Detector Gap:** For all my experiments, I consider the detector gap as equal to 0.

6) **Pixel Width:** For all my experiments, I consider pixel width as equal to 1.

7) **Weighting Scheme:** The weight factors or the contribution that each pixel makes in a ray are calculated using four different approaches. "Binary", "Length of ray within a pixel", "Distance of center of pixel from center of ray" and "Contribution made by the pixel in adjacent rays" are the four weighting schemes presented in this thesis. The different schemes are explained in detail in section 4.5.

8) **Type of *ART:** I have written code for ART, MART, SIRT and SART.

9) **Seed i mage:** T he different test images used for the experiments are shown in Figure 4-3 and the details about it are provided in section 4.4.

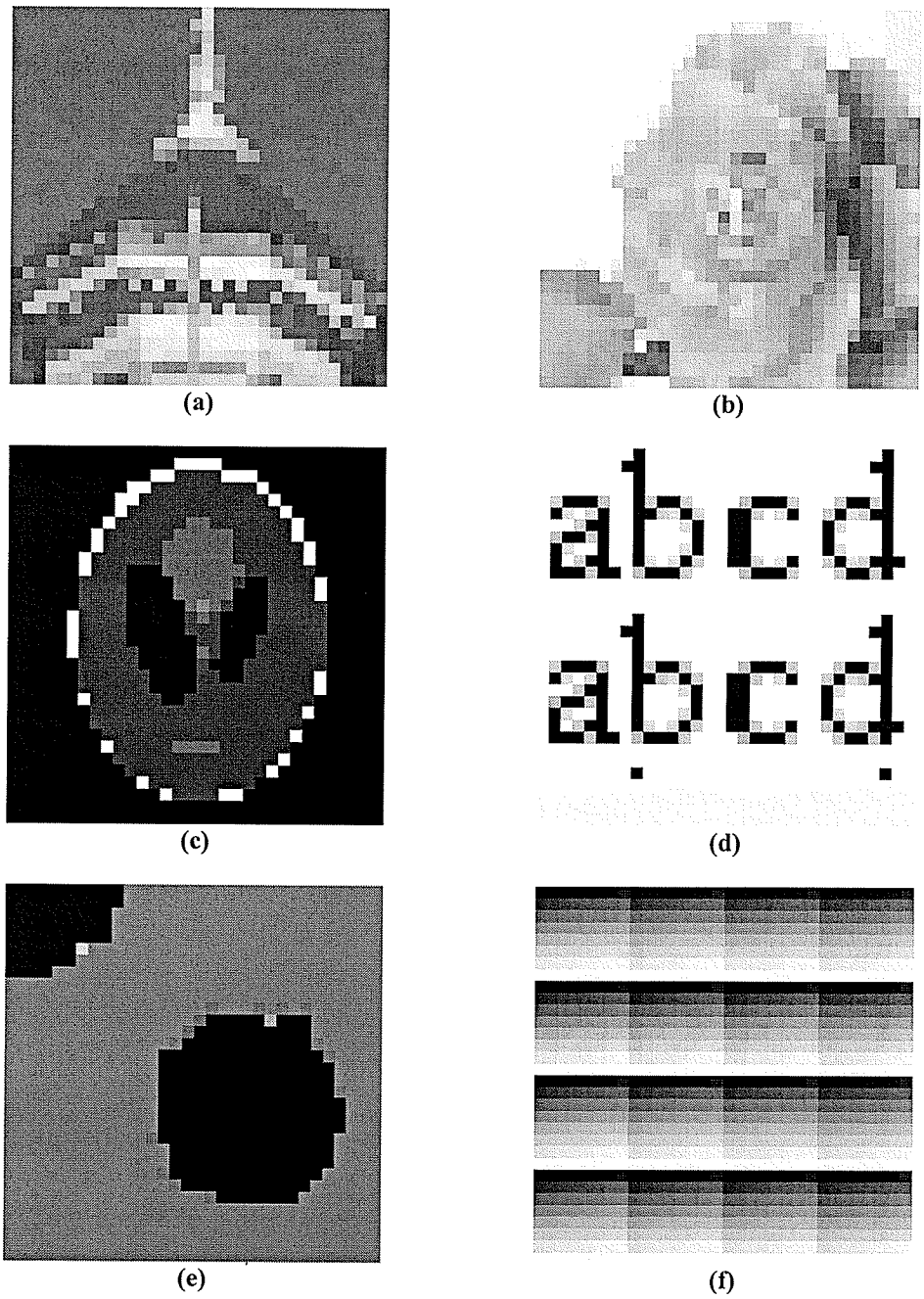
10) **Number of Cycles:** The algorithm is executed until the convergence criteria are met. The convergence criteria used for the experiment is explained in section 4.7.

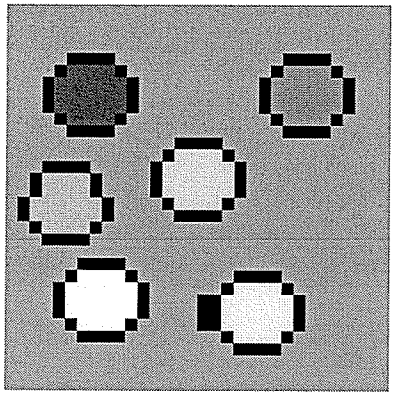
11) **Projection angles:** This parameter specifies the different angles at which the projections are taken.

12) **Projection Angle Ordering Scheme:** This refers to the order in which the projection angles are considered. "Sequential", "Fixed Angle", "Random", "Multilevel resolution access", "Weighted Distance" are the five projection angles ordering schemes discussed in this chapter. See section 4.6 for the discussion on the projection angle ordering schemes.

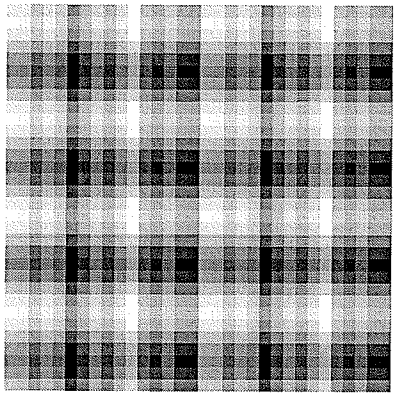
13) **Relaxation Factor:** For all my e xperiments, I consider the relaxation factor as equal to 0.5.

The different parameters are explained graphically in Figure 4-2. I wrote the code in Matlab^{TM} . The front-end application, where the user can enter parameters is shown in Figure 4-4.

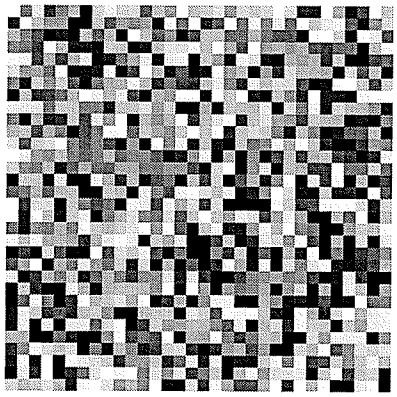




(g)



(h)



(g)

Figure 4-1. *Test images used for *ART experiments All images are 32 by 32 pixels.* **(a)** *admin32.bmp: University of Manitoba Administration Building (courtesy of Prof. W. Lehn, University of Manitoba, reproduced from his Digital Image Processing class).* **(b)** *rose32.bmp: White Rose.* **(c)** *phantom32.bmp: Phantom image* **(d)** *abcd32.bmp: Image containing the text "abcd".* **(e)** *paint32.bmp: This image is created mathematically. It contains a circular patter within a checkerboard pattern.* **(f)** *shingles32.bmp: This image is created mathematically. It contains repeated blocks where each block contains pixels whose value increases gradually from top left to bottom right.* **(g)** *circles32.bmp: This image is created mathematically. It contains 6 randomly placed circles of uniform radius having random gray scale background on a uniform gray scale background.* **(h)** *sine32.bmp: This image is created mathematically. It contains vertical and horizontal sine waves of frequency 10 superimposed on top of one another.* **(i)** *rand32.bmp: This image is created contains pixels with uniformly distributed pixel values.*

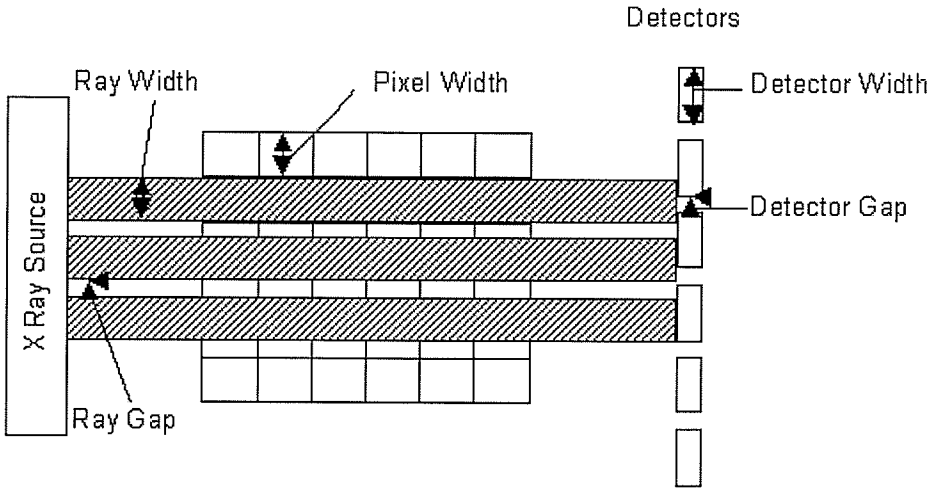


Figure 4-2. *Illustration of the different parameters that affect the convergence and quality of *ART reconstruction.*

4.3 Test Images used for the *ART experiments

In order to be fair in my analysis of results, I did the experiments on a variety of test images. Shown in Figure 4-1 are the different test images used for my experiments. All the test images are 32 by 32 pixels. The choice of the images is done to consider images with as diverse properties as possible –

a) admin32.bmp (henceforth referred to as test image *a*): This image is a low brightness low contrast image.

b) rose32.bmp (test image *b*): This image is a high brightness low contrast image.

c) phantom32.bmp (test image *c*): This image contains zeros except in the area inside the outer ellipse.

d) abcd32.bmp (test image *d*): This image was constructed to provide a better visual interpretation of the reconstruction quality.

e) paint32.bmp (test image *e*): This image is constructed mathematically. It contains circles within a checkerboard background. This test image is used to see the effect (if any) of the checkerboard background on the circles.

f) shingles32.bmp (test image *f*): This image is constructed mathematically. It contains repeated 8 by 8 blocks, where the pixel value in each block increases gradually from the top left to bottom right. It is a good representation of local concentration of high contrast.

g) circles32.bmp (test image *g*): This image is constructed mathematically. It contains 6 circles of uniform radius of varying gray levels (contrast) placed randomly on the 32 by 32 grid. This image is used as it is a good representation of localization of varying contrast areas.

h) sine32.bmp (test image *h*): This image is constructed mathematically. It contains a vertical and a horizontal sine wave of frequency 10 superimposed on top of one another.

i) rand32.bmp (test image *i*): This image is constructed such that each pixel value is a generated by a uniform random number generator. This image is not partial to any particular geometry, contrast or brightness.

4.4 Seed Images used for the *ART experiments

A variety of seed images was used for the experiments. Shown in Figure 4-3 are the different seed images used. All the seed images have a 32 by 32 pixel resolution.

a) Zeros seed (zeros): The value of all pixels in this seed image is 0.

b) Flat seed (flat): This seed contains pixels with uniform gray value (all ones).

c) Meshgrid seed (meshgrid): The values of all pixels in each column of this image is same. The value increases from left to right.

d) Random seed (rand): The pixel values in this seed image are generated by a uniformly distributed random number generator.

e) Random-Normal seed (randn): The pixel values in this seed image are generated by normally distributed random number generator with a mean of 128 and variance of 32.

f) Checkerboard seed (checkerboad): The pixels values form a checkerboard pattern. The dark square has value of 1 and the bright square has value of 255.

g) Shingles seed (shingles): This seed image contains pixels where the pixel value increases from top-left to bottom right. Because of the limitation of our eye in deciphering differing shades of gray and also because of the contrast resolution of the printer, the image shown in Figure 4-3(g) may not look that way.

h) Sinewave seed (sinewave): This seed image contains a vertical sine wave of frequency one cycle across the width of the image.

i) Noise seed (noise): Noise is added to the test image to create this seed. The Signal to Noise Ratio (SNR) for generating the noise is 50 (unless otherwise specified).

j) FBP seed (fbp): This seed image is created by taking the filtered back projection of the test image. i.e., the test image is reconstructed using filtered back projection technique first for the same projection data. The reconstructed image is then used as the seed for the *ART experiments.

k) Stretched seed (stretch): The seed image is generated by stretching the test image by a stretch factor of 150%.

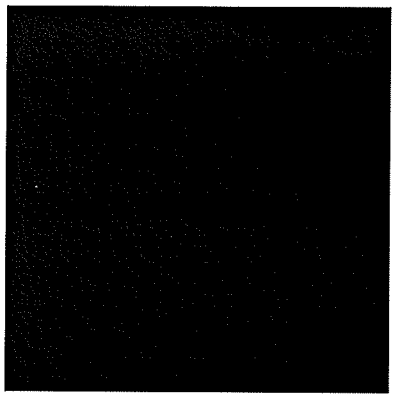
l) Blur seed (blur): This seed image is generated by blurring the test image by a blur factor. The blur factor for the experiments is 20% unless otherwise specified.

Algorithm for generating the blur image –

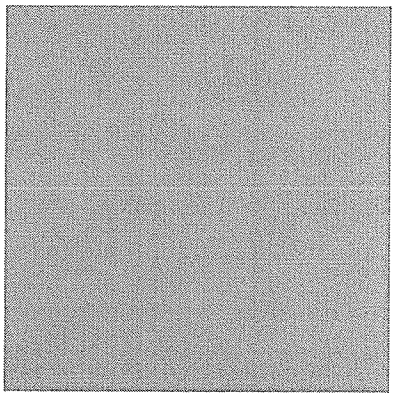
a. Take the test image. Call it *T*.

b. Reduce the image size of *T* to (100 - blur factor) % of its original size. Reducing the image size will blur the image. Call this image as *S*.

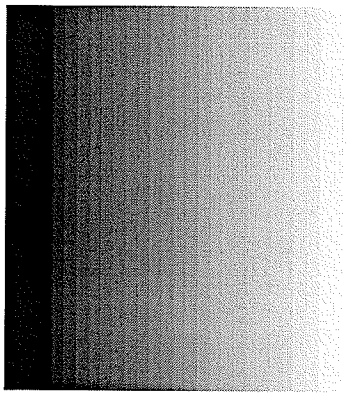
c. Increase size of *S* to the same size as *T*. This new image whose size is same as that of *T* is the blurred image.



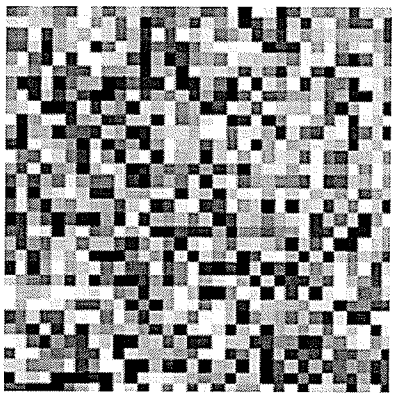
(a) zeros



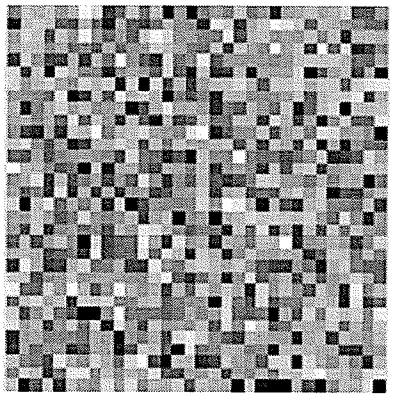
(b) flat



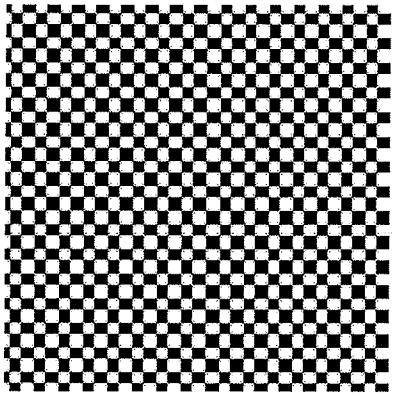
(c) meshgird



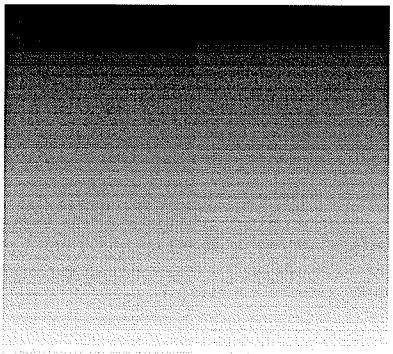
(d) rand



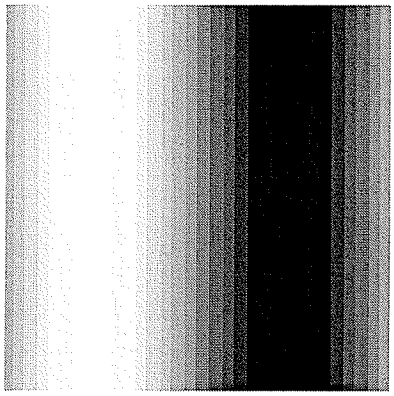
(e) randn



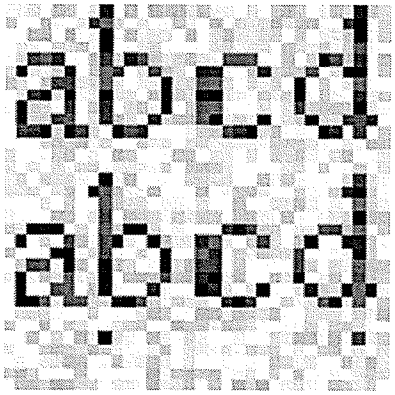
(f) checkerboard



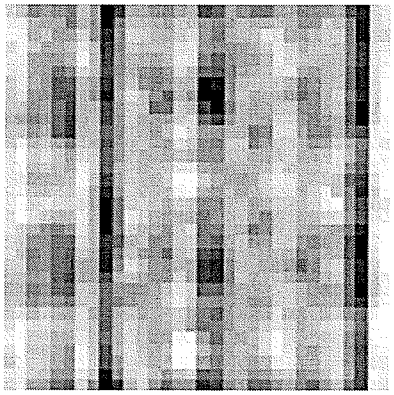
(g) shingles



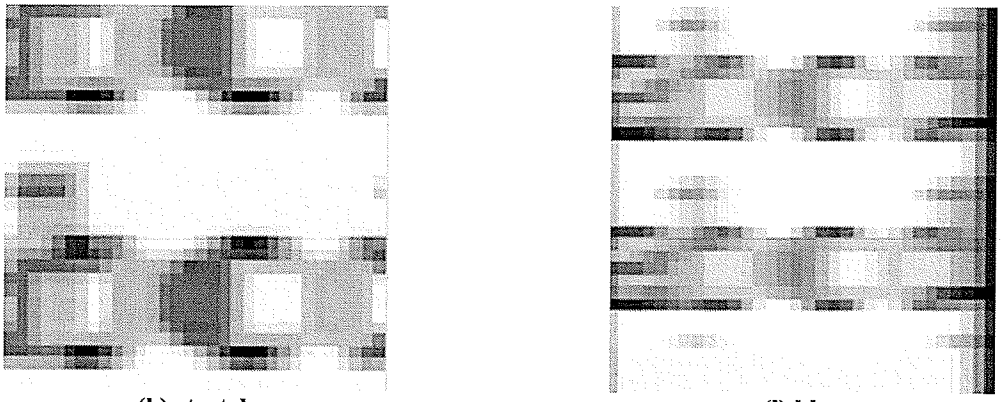
(h) sinewave



(i) noise



(j) FBP



(k) stretch **(l) blur**

Figure 4-3. *Seed images used for *ART experiments.* **(a)** *Zeros Seed: All pixel values are 0.* **(b)** *Flat seed: All pixel values are 1* **(c)** *Meshgrid seed: Generated mathematically where the pixel value increases from left to right.* **(d)** *Rand seed: Generated by uniformly distributed random number generator* **(e)** *Randn seed: Generated by normally distributed random number gernerator with a mean of 128 and a variance of 32.* **(f)** *Checkerboard seed: Alternate checkers are 1 and 255* **(g)** *Shingles seed: The pixel values in this seed progresses arithmetically from the top left to bottom right.* **(h)** *Sinewave seed: The pixel values represent a horizontal sine wave of frequency one* **(i)** *Noise seed: Seed image is generated by adding noise to the original image* **(j)** *FBP seed: Seed image is the reconstruction obtained by the Filtered backprojection technique* **(k)** *Stretched seed: Seed image is generated by stretching the original image* **(l)** *Blurred seed: Seed image is generated by blurring the original image.*

4.5 Pixel Weighting Schemes

For algebraic techniques a ray is defined as a "fat" line running through the image. This is illustrated in Figure 4-5. Researchers have often considered the ray width as equal to the image pixel width. I have done the same in my experiments. However my code gives the flexibility to experiment with different pixel and ray widths. When a ray passes through the image, each pixel that falls on the ray contributes a certain fraction on the ray sum. The fraction by which the pixel contributes is between 0 and 1, where 0 indicates no contribution and 1 indicates that the entire pixel lies within the ray. This fraction depends upon the material (atomic number and electron density) of the pixel and the area of the pixel that lies in the ray. Since this thesis is dealing with the inverse problem, all pixels are considered homogeneous and hence the effect of atomic number and electron density are not considered. In essence the physics behind forward CT is ignored. Since, precise

calculation of the area is generally difficult, people tend to simplify this by various weighting schemes.

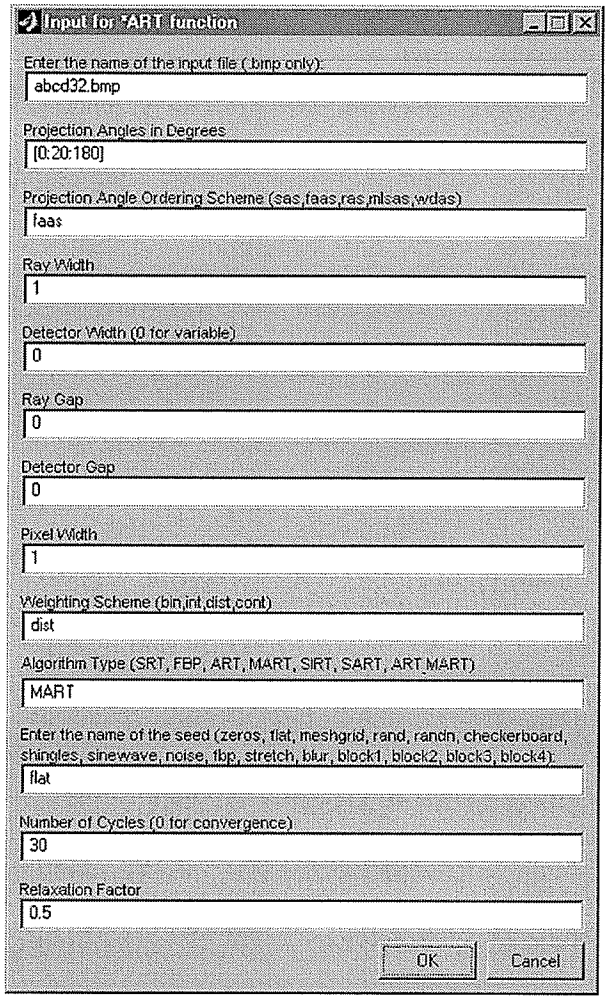


Figure 4-4. *Front-end application screen for entering the different parameters for the *ART experiments.*

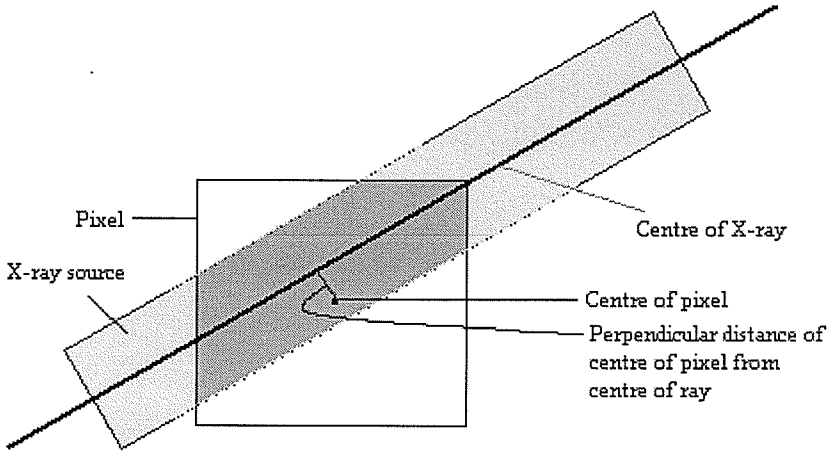


Figure 4-5. *Illustration of the intersection of a ray with a pixel.*

4.5.1 Binary Weighting Scheme (BIN)

The weighting scheme most commonly used in ART is the binary scheme. In this scheme the contribution made by the pixel in a ray sum is considered equal to one if the center of the pixel falls in the ray. If the center of the pixel does not lie in the ray then the contribution of the pixel in the ray sum is considered equal to zero.

4.5.2 Length of Center of Ray within Pixel Weighting Scheme (INT)

In this scheme the weighting factors are calculated as the length of the center of the ray within the pixel. Figure 4-6 is a graphical illustration of the length of ray within pixel scheme.

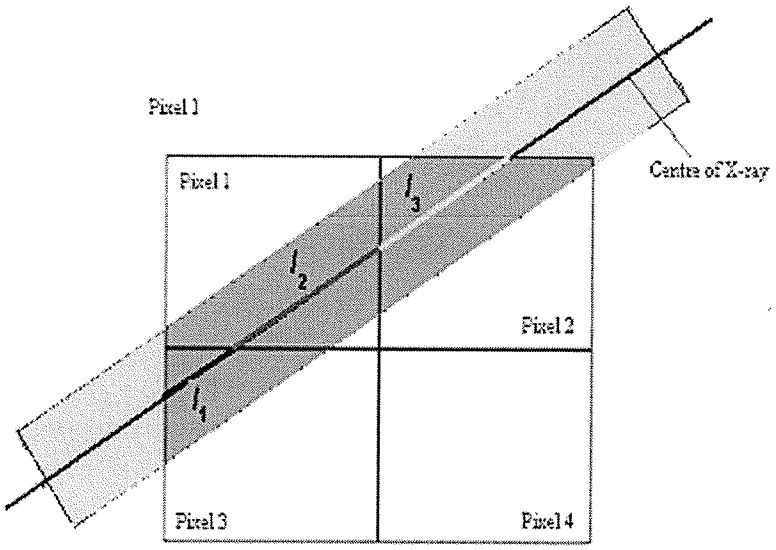


Figure 4-6. *Illustration of the "Length of center of ray within a pixel" weighting scheme.* l_1, l_2, l_3 are the length of the ray within pixel 3, 2 and 1 respectively.

At 45°, the ray covers the maximum distance within a pixel, if it passes through the center of the pixel. This distance is equal to $\sqrt{2}$ times the pixel width. Hence I normalize the length over this value to get the weighting factors. The weighting factors for pixels 1, 2, 3 and 4 in Figure 4-6 will be $l_2 / (p\sqrt{2})$, $l_3 / (p\sqrt{2})$, $l_1 / (p\sqrt{2})$ and 0 respectively where p is the pixel width.

4.5.3 Distance of Center of Pixel from Center of Ray Weighting Scheme (DIST)

In this scheme, the contribution made by a pixel in the ray is calculated as a function of the distance of the center of the pixel from the center of ray. If R is the ray width and d is the distance of the center of the pixel from the center of the ray, then the weighting factor is given by -

$$W = 1 - 2d/R \qquad d <= R/2$$

$$= 0 \qquad \text{otherwise}$$

By this scheme, if the center of the pixel lies on the center of the ray, it will have a weighting factor of one. The weighting factor will decrease linearly as the distance of the center of pixel from the center of ray increases and will become zero if the distance is greater than $R/2$. Figure 4-7 shows the plot of weighting factor W as a function of distance d.

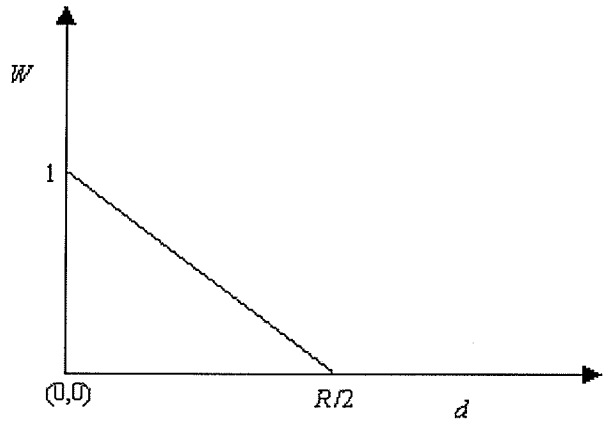
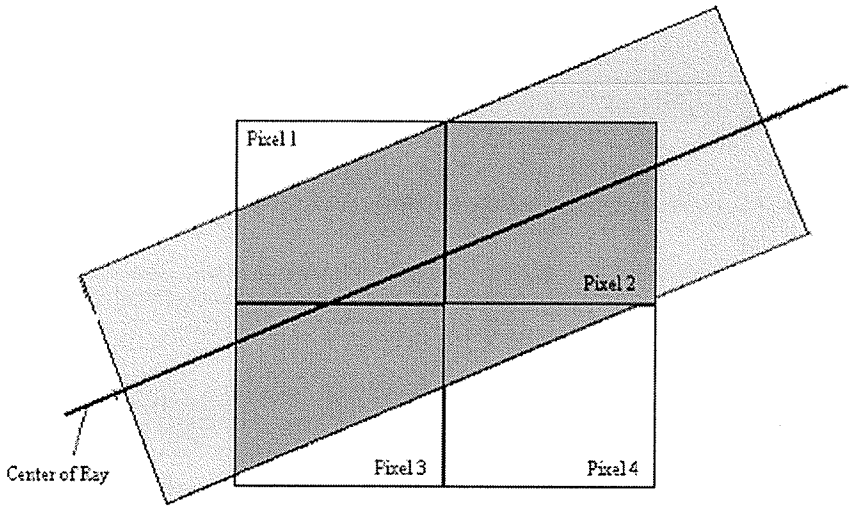


Figure 4-7. *Plot of weighting factor W as a function of d in the case of "Distance of center of pixel from center of ray" weighting scheme.*

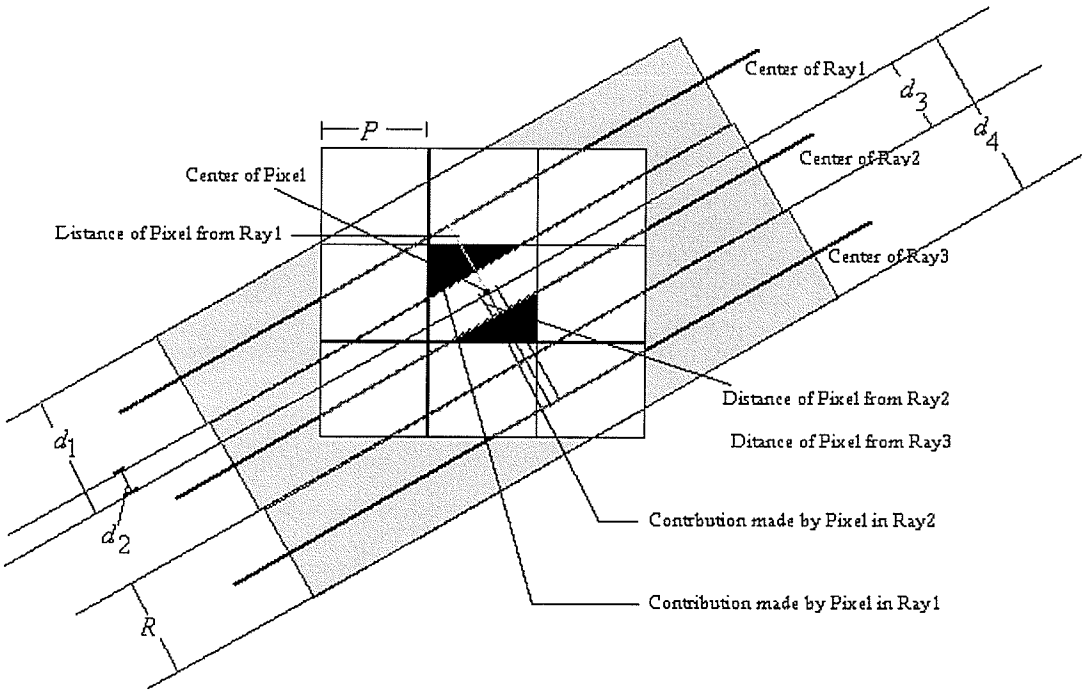
4.5.4 Distance of Center of Pixel from Farthest Edges of Adjacent Ray Scheme or Consideration of contribution made by Pixel on Adjacent Rays Scheme (CONT)

Intuitively any pixel in the original image would have contributed 100% in a particular ray or in a combination of adjacent rays. However the "Length of center of ray within pixel" as well as the "distance of center of pixel from center of ray" approach does not

guarantee this condition. Figure 4-8 (a and b) shows scenarios where the two previous schemes do not satisfy the 100% contribution condition.



(a)



(b)

Figure 4-8. (a) *Illustration of scenario where the "length of ray within pixel" weighting scheme does not satisfy the 100% pixel-contribution condition.* **(b)** *Illustration of scenario where the "distance of center of ray from center of pixel" does not satisfy the 100% pixel-contribution condition.*

Using the "length of center of ray within pixel" scheme as shown in Figure 4-8(a), we get the weight factor of pixel 2 as less than one since it is not passing through the opposite corners of the pixel. However, we can see that the pixel lies completely within the ray and hence ideally contributes 100%. This means that we are not getting the true contribution made by pixel 2 on the ray.

Similarly employing the "Distance of center of pixel from center of ray" scheme in the scenario shown in Figure 4-8(b), we get the weight factors of the centermost pixel on Ray 1 and Ray 3 as zero. Also the centermost pixel's contribution on the ray sum of Ray 2 is not equal to one, as the center of the pixel does not lie on the centerline of Ray 2. Hence essentially, we are losing the true contribution made by the pixel on the detectors.

To rectify this problem, I modified the "distance of center of pixel from center of ray" approach to consider the contribution made by the pixel on the adjacent rays to calculate the weight factor of the pixel on the current ray. The algorithm is explained more clearly below.

Algorithm of "Consideration of pixel contribution on adjacent rays" weighting scheme.

Consider Figure 4-8 (b).

a) Let R be the ray width, P the pixel width, d_1 the distance of the center of the pixel from the farthest-edge of Ray 1, d_2 the distance of the center of the pixel from the first-edge of Ray 2, d_3 the distance of the center of the pixel from the second-edge of Ray 2 and d_4 the distance of the center of the pixel from the farthest-edge of Ray 3.

b) Contribution made by the pixel on Ray 1

$$c_1 = 0 \quad \text{if } d_1 >= R+P/\sqrt{2}$$

$$= 1 - d_1/(R+P/\sqrt{2}) \quad \text{otherwise}$$

c) Contribution made by the pixel on Ray 3

$$c_3 = 0 \quad \text{if } d_4 >= R+P/\sqrt{2}$$

$$= 1 - d_4/(R+P/\sqrt{2}) \quad \text{otherwise}$$

d) If the pixel is contributing in Ray 2 (determined based on d_2 and d_3), then the contribution made by pixel on Ray 2 is given by $1 - (c_1+c_3)$

e) The above algorithm considers only the two adjacent rays and is true if $P <= R$. If R>P, more adjacent rays need to be considered.

4.6 Projection Angle Ordering Schemes

It has been known for quite some time [20], that both the quality of the approximation and the rate of convergence of the iterative reconstruction procedure depend, among other factors, on the order in which the projections are selected for grid correction. In this section I discuss a few projection angle order schemes.

4.6.1 Sequential Access Scheme (SAS)

In this scheme the projections are taken in the same order as that of the projection data. For example if the projections are taken at 0 degrees, then at 30 degrees then at 15 degrees and then at 85 degrees, the algorithm also considers the data in the same order 0, 30, 15 and 85. The schematic representation of the access order for 9 projections using SAS is shown in Figure 4-9.

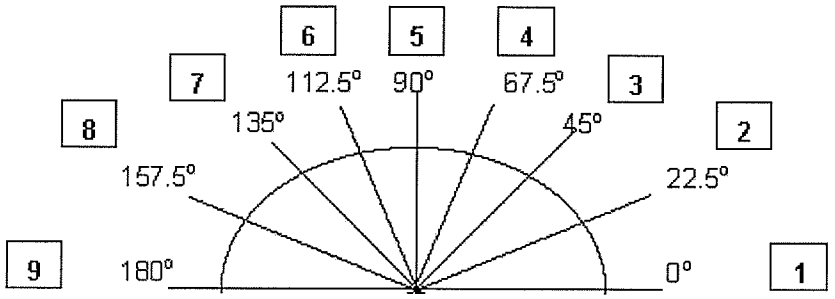


Figure 4-9. *Schematic representation of the access order for 9 projections using SAS*

4.6.2 Fixed Angle Access Scheme (FAAS)

A number of researchers have pointed out [19] [21] that it is desirable to order the projections in such a way that subsequently applied projections are largely uncorrelated. This means that consecutively applied projections must have significantly different angular orientations. Many implementations have used a fixed angle for projection spacing: In my code, I consider the constant angle to be 90 degrees.

For example consider 10 equally spaced projections taken from 0 to 180 degrees. i.e., the projection angles are 0, 20, 40, 60, 80, 100, 120, 140, 160 and 180. According to the FAAS algorithm the angle 0 will be considered first. The second angle is considered such that is as close to 90 degrees with the first angle (0 degrees) as possible. Hence the angle 80 is selected next. The third angle is considered such that it is as close to 90 degrees with the second angle as possible. Hence the angle 160 is selected and so on. However I modified the code such that the odd angles are selected in sequential order and the even angles are selected close to 90 degrees apart from their odd counterpart to make the access order cyclic in a 180° range. Hence for the given example the angles will be order in the sequence 0, 80, 20, 100, 40, 120, 60, 140, 160, 180.

The schematic representation of the access order for 9 projections using FAAS is shown in Figure 4-10.

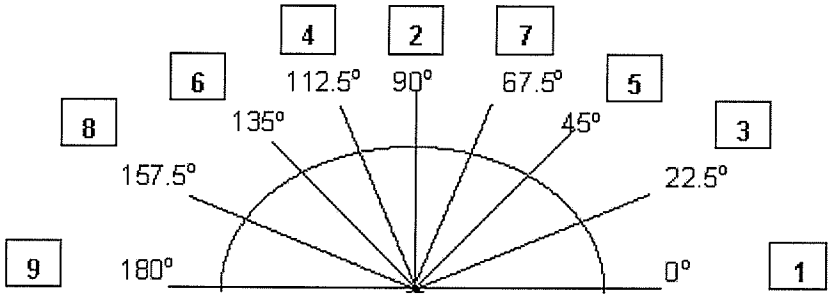


Figure 4-10. *Schematic representation of the access order for 9 projections using FAAS*

4.6.3 Random Access Scheme (RAS)

VanDijke [32] concluded that, among all schemes he tried, a random projection permutation gave the best results. In this scheme the projections are taken in a random order. The schematic representation of the access order for 9 projections using RAS is shown in Figure 4-11.

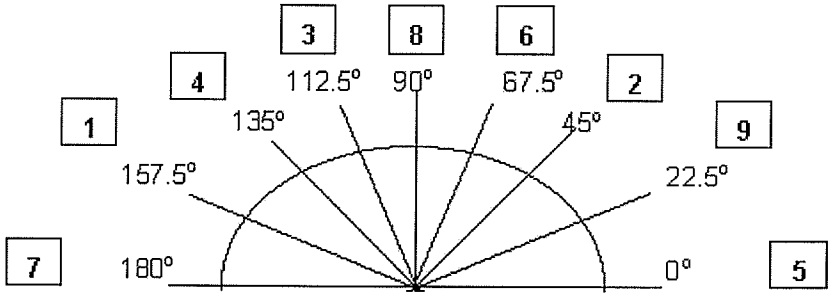


Figure 4-11. *Schematic representation of the access order for 9 projections using RAS*

4.6.4 Multilevel resolution select Access Scheme (MLSAS)

Using the Random Access Scheme as proposed by VanDijke will give inconsistent results each time the experiment is done due to the sheer nature of the angle ordering

scheme. Therefore one may prefer an ordering scheme that is more controllable and deterministic than a random number generator. Recently, Guan and Gordon [21] presented, what they termed, the Multilevel Access Scheme(MLS). This method works best when the number of projections is a power of 2, but can also be used, with minor modifications, in the general case. The following description is for the simple case of M being a power of 2: First, for level one and two, the method chooses the projections at 0*, 90*, 45*, and 135*. All subsequent levels L=3, .., $\log_2 S$ contain $2L$ views. The projection order at level L is computed by simply going through the list of all applied projections at levels l. MLS generates a permutation of the angle ordering such that there is an even spread of the applied projections around the reconstruction cycle.

For example consider 10 equally spaced projections taken from 0 to 180 degrees, i.e., the projection angles are 0, 20, 40, 60, 80, 100, 120, 140, 160 and 180. According to the MLSAS algorithm the angle 0 will be considered first. The second angle is considered such that is as close to the center angle of the reconstruction angle as possible. Hence 80 is considered next. The third angle is selected such that it is close the mean of the first two angles, hence 40 is selected next. The fourth angle is the mean of the third and the last angle and the angle that has the highest degree of uncorelation with the third angle. Hence 120 is selected next and so on. Hence for the given example the angles will be order in the sequence 0, 80, 40, 120, 20, 100, 60, 140, 160, 180.

The schematic representation of the access order for 9 projections using MLSAS is shown in Figure 4-12.

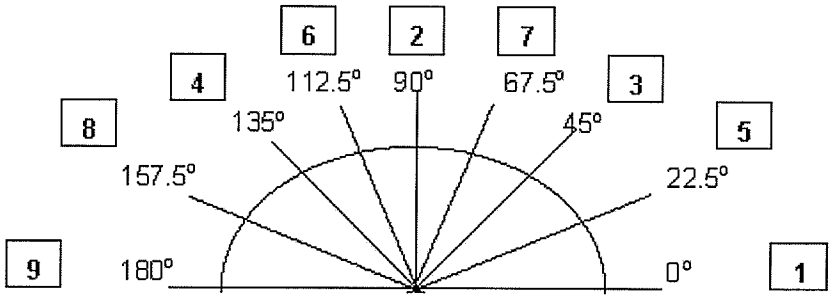


Figure 4-12. *Schematic representation of the access order for 9 projections using MLSAS*

4.6.5 Weighted Distance Access Scheme (WDAS)

[27] While all previously proposed ordering schemes take great care to space far apart consecutively chosen projections, they somewhat neglect the problem of optimizing the selection in a global sense. In the process of selecting a newly applied projection, all, or at least an extended history of, previously applied projection orientations must be taken into account and weighted by their time of application. The Weighted Distance Scheme (WDS) heuristically optimizes the angular distance of a newly selected projection with respect to the complete sequence of all previously applied projections (including those applied in the previous iteration) or any continuous, time-wise adjacent subset thereof. Thus the WDS projection angle scheme is suitable when -

a) a series of subsequently applied projections is evenly distributed across a wide angular range and

b) at no time is there an angular range that is covered more densely than others.

All of the existing methods tend to be strong in one of the two aspects, but weaker in the other. However, none of the previous methods comments on how one should proceed with the projection selection at iteration boundaries. It is clearly necessary to also include projections applied in previous iterations into the selection process. A smooth transition

between iterations is warranted if the selection scheme is continuous across cycle boundaries i.e., in the previous methods the same sequence of projections are repeated for all cycles, the weighted distance considers the projections taken in the previous cycle to determine the sequence of projections for the next cycle. The Weighted Distance Projection Ordering Method is designed to maintain a large angular distance among the whole set of used projections while preventing clustering of projections around a set of main view orientations. The method selects, from the pool of unused projections, that projection that optimizes both the angular spacing and the spread with respect to the complete set or a recent subset of all previously applied projectional views. Hereby it takes into account that more recent applied projections should have a stronger influence in the selection process than projections that have been applied earlier in the reconstruction procedure. The selection algorithm results in uniform sampling of the projection access space, minimizing correlation in the projection sequence. This produces more accurate images with less noise-like artifacts than previously suggested projection ordering schemes.

The schematic representation of the access order for 9 projections using WDAS is shown in Figure 4-13.

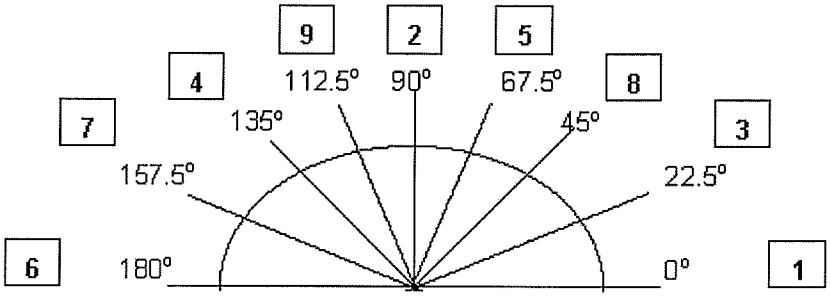


Figure 4-13. *Schematic representation of the access order for 9 projections using WDAS.*

Table of the projection angle ordering schemes –

Given the projection angles [0, 22.5, 45, 67.5, 90, 112.5, 135, 157.5, 180], the different ordering schemes will consider the angles as shown in Table 4-1.

Sequential Access Scheme (SAS)	Fixed Angle Access Scheme FAAS)	Random Access Scheme (RAS)	Multilevel Resolution Select Access Scheme (MLSAS)	Weighted Distance Access Scheme (WDSAS)
0	0	157.5	0	0
22.5	90	45	90	90
45	22.5	112.5	45	22.5
67.5	112.5	135	135	135
90	45	0	22.5	67.5
112.5	135	67.5	112.5	180
135	67.5	180	67.5	157.5
157.5	157.5	90	157.5	45
180	180	22.5	180	112.5

Table 4-1. *Table showing the different projection angle schemes for 9 equally spaced projections taken from 0 to 180 degrees.*

4.7 Results

The results presented in this chapter are based solely on the experiments that I performed. I have written all the code except the code for implementing the weighted distance scheme. I have reproduced this code from Chris Badea (Email: chris@orion.mc.duke.edu).

The various factors affecting the reconstruction based on *ART is discussed in section 4.2. The only factors that I experimented on are the seed image, projection angle ordering schemes, pixel weighting schemes and the different *ART algorithms. Since I used nine test images, nine seed images, five projection angle ordering scheme, four pixel weighting schemes and four *ART algorithms (ART, MART, SIRT and SART), there were 9*9*5*4*4 = 6480 experiments that needed to be performed. Due to the limitation of time I did only 500 experiments. The results are summarized in this section for illustrative purposes.

Number of Cycles:

ART, MART and SIRT were executed until convergence, but since SART gave noticeable differences at 250 cycles, the SART experiments were done for 250 cycles only (unless otherwise specified).

Convergence Criteria:

All the *ART algorithms have a strange behavior in the sense that if the number of iterations are increased beyond a certain number of iteration, the error starts increasing [35]. I used this criterion as the convergence criteria for my code. This is illustrated in Figure 4-14.

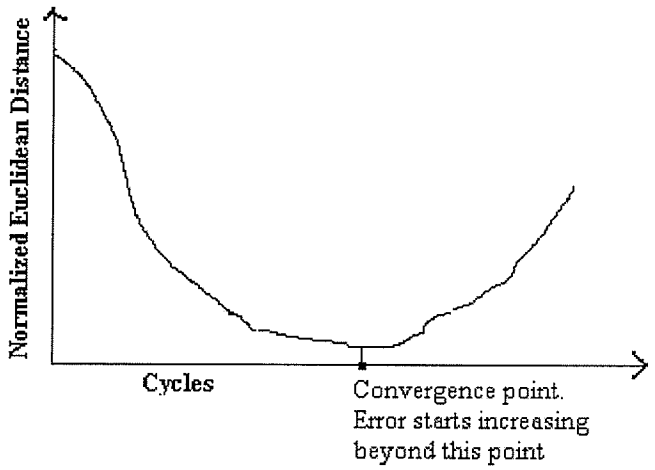


Figure 4-14. *Illustration of convergence criteria. Beyond a certain number of cycle, the error in *ART algorithms starts increasing, i.e., the algorithm is said to be converged at the cycle beyond which the error is equal to or more than the previous cycle.*

Size of Weight Matrix and Image size:

The code was written such that the weight factor matrix is stored in memory. The weight factor is a 4 dimensional array and increases rapidly with image size and number of projection angles. Storing the weight matrix in memory gives us the advantage of faster computing time. On the flip side, it needs more memory for large images and higher projection data. A typical 32 by 32 image for 10 projection angles will require 32*32*10*32 = 327680 elements where each array element will store (depending upon the weighting scheme used) at least 8-bit information (for binary weighting scheme). This requires a minimum of 327KB memory. If the image size increases to 256 by 256 and the number of projection angles is 30, then there will be 503316480 elements in the weight matrix array and a minimum of 503MB memory is required. If the weighting scheme used is non-binary, then more bits are required to represent the weight factor. Typically for a non-binary scheme, a 16-bit floating number is required to represent the weight

factor, which doubles the weight matrix size. I used the test images shown in Figure 4-1 for my experiments. All these images are of size 32 by 32. 10 projection angles were used for the experiments. Hence the task was to find 1024 (32 by 32) unknowns using 320 (10*32) equations. In the practical case the size of the image is at least 256 by 256 (65536 unknowns). The results shown in this chapter may or may not hold true for the practical case. Given a more powerful machine, the code written for this thesis can be used for larger images. This work is definitely warranted.

Image Quality Measure:

Euclidean distance was used as the image quality measure for quantitative differentiation of the reconstructions. Euclidean distance is calculated between the original test image and the reconstructed image and is normalized based on the maximum allowed pixel value and image size, to make it gray-scale independent and dimensionless (size independent). For more information about Euclidean distance refer section 2.3B in chapter 2.

4.7.1 Comparison of ART, MART, SIRT and SART

The experiments were done with *ART for all test images. All the parameters were kept constant except the algorithm type. The values for the different parameters are as below –

a) Projection Angles: 10 equally spaced projections from 0 to 180 degrees.

b) Projection angles ordering scheme: Fixed Angle Ordering Scheme.

c) Ray width: 1

d) Ray Gap: 0

e) Detector Width: 1

f) Detector Gap: 0

g) Pixel Width: 1

h) Weighting Scheme: "Distance of the center of pixel from center of ray" scheme

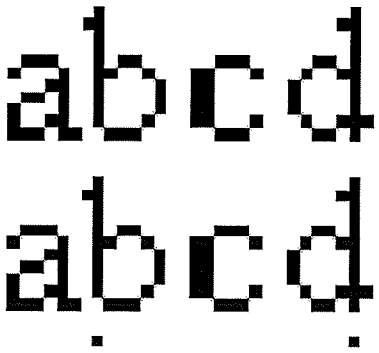
i) Test Image: Used all the test images shown in Figure 4-1.

j) Seed Image: All the seed images shown in Figure 4-3 were used. However, for illustrative purposes, only the results from Flat seed are shown in this thesis for this experiment.

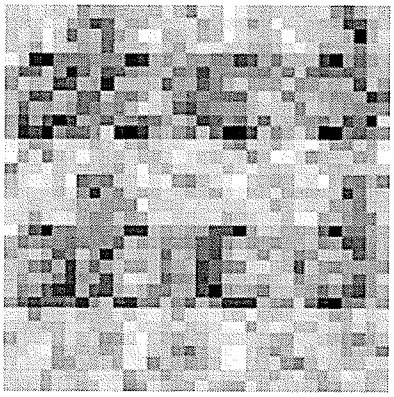
k) Number of Cycles: Until convergence. The convergence criterion used is that the normalized Euclidean distance starts increasing or is the same as the previous cycle.

l) Relaxation Factor: 0.5

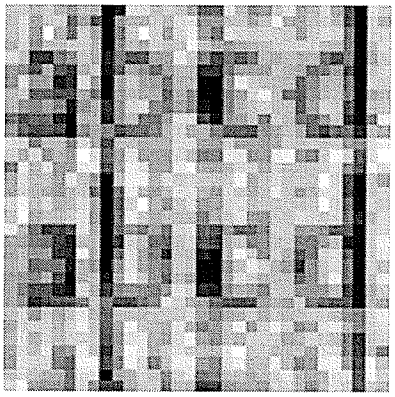
Figure 4-15 (b-e) shows the reconstructed images based on ART, MART, SIRT and SART. Note that even though this experiment was performed for all test images, only test image *d* is shown in the figure for illustrative purposes. Figure 4-15(f) shows the plot of the normalized Euclidean distance of each of the algorithms against the particular test image.



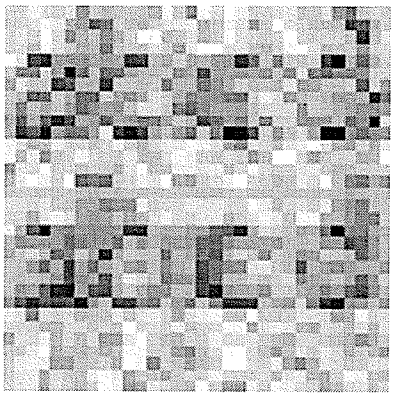
(a)



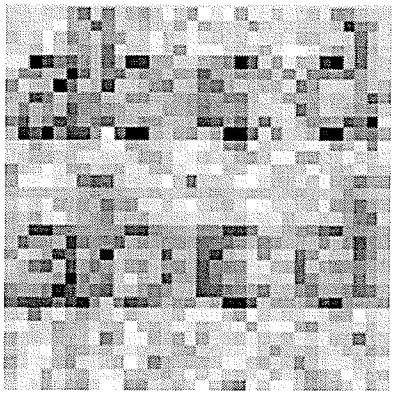
(b)



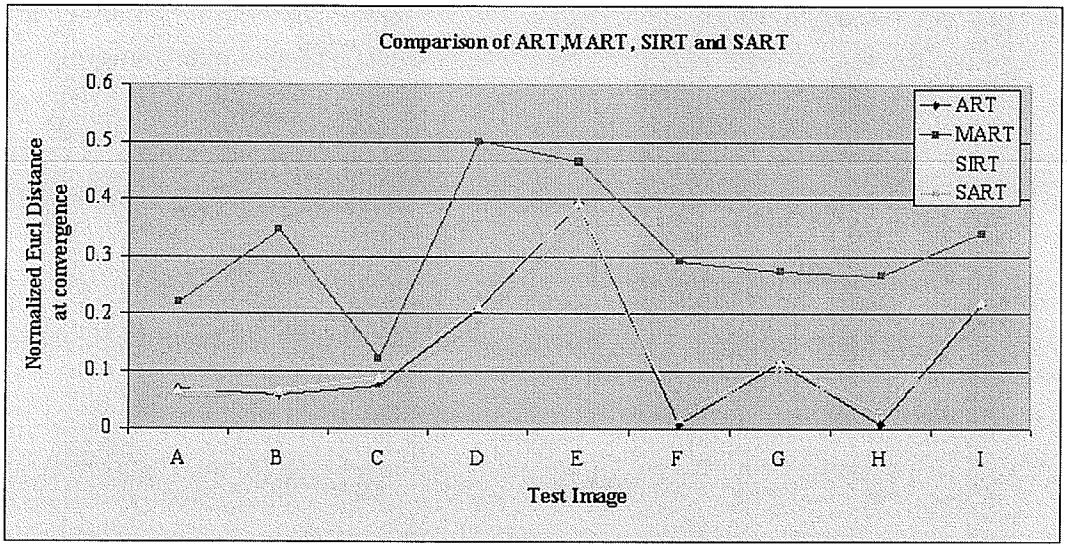
(c)



(d)



(e)



(f)

Algorithm Type	Mean Euclidean Distance	1σ
ART	0.1272	0.1250
MART	0.3138	0.1168
SIRT	0.1343	0.1209
SART	0.1174	0.1118
Conclusion: Based on the experiment parameters, the mean Euclidean distance was least (best) for SART followed by ART, SIRT and MART. Little significant difference was found between SART, ART and SIRT.		

(g)

Algorithm Type	Mean Number of Cycles to converge
ART	17
MART	4
SIRT	231
SART	1032
Conclusion: Based on the experiment parameters, MART converges faster than ART; ART converges faster than SIRT; SIRT converges faster than SART.	

(h)

Figure 4-15. *Comparison of ART, MART, SIRT and SART* **(a)** *Test Image* **(b)** *Reconstructed Image based on ART* **(c)** *Reconstructed Image based on MART* **(d)** *Reconstructed Image based on SIRT* **(e)** *Reconstructed Image based on SART* **(f)** *Plot comparing the different algorithm types for the different test images* **(g)** *Summary of the Euclidean distance comparison between ART, MART, SIRT and SART* **(h)** *Summary of the number of iterations required for convergence between ART, MART, SIRT and SART.*

As one can see from Figures 4-15 (b-e), the visual quality of the reconstructions is very close to one another. Although 4-15 (c) looks better than the others, one is not able to deduce quantitatively the degree of difference between the reconstructions. Hence Euclidean distance was used to provide a quantitative measure of the difference in the image quality between the different algorithms types. Figure 4-15 (f) shows that the Euclidean distance for MART is worse compared to the other algorithms for all the test images used. The Euclidean distance measure of SIRT is better than MART at all times and is the same or worse than ART and SART at all times. The Euclidean distance measure of ART is very close to SART and for some images (example test image *H*) becomes better than SART. At the outset, taking the mean of all the Euclidean distances for a particular algorithm type, one can arrive at the conclusion that the Euclidean distance of SART is the best among the algorithm types used for the experiment, ART ranks next after SART in terms of Euclidean distance measure, then is SIRT and finally MART.

I also compared the time required for the iterations to converge. The correction made in each cycle in SIRT and SART is less compared to the per-cycle-correction made in ART and MART. This is because the corrections at each projection in SIRT and SART are stored and averaged at the end of the cycle. This not only requires additional storage for the correction matrix but also the correction factor is less. Hence the time and the number of cycles required for SIRT and SART to converge is much larger than that of ART and MART. This is shown in Figure 4-15 (h).

4.7.2 Comparison of different seed images in *ART reconstruction

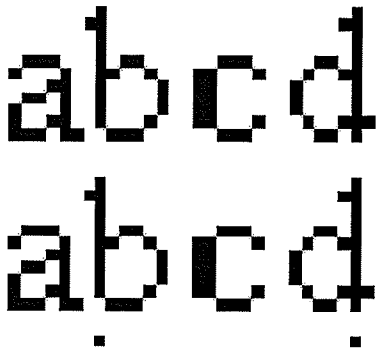
The experiments were done with *ART for all test images. All the parameters were kept constant except the seed image. The values for the different parameters are as below –

a) Projection Angles: 10 equally spaced projections from 0 to 180 degrees.
b) Projection angles ordering scheme: Fixed Angle Ordering Scheme.
c) Ray width: 1
d) Ray Gap: 0
e) Detector Width: 1
f) Detector Gap: 0
g) Pixel Width: 1
h) Weighting Scheme: "Distance of the center of pixel from center of ray" scheme
i) Test Image: Used all the test images shown in Figure 4-1.
j) Seed Image: All the seed images shown in Figure 4-3 were used. However, for illustrative purposes, only the results from Flat seed, Meshgrid seed and FBP seed are shown in this thesis for this experiment.

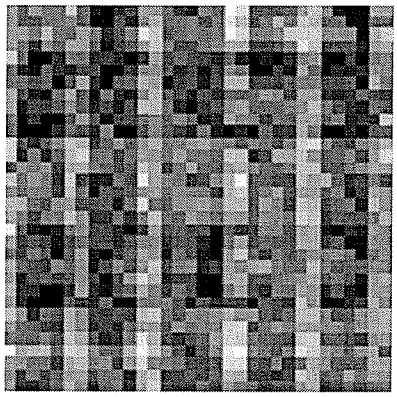
k) Number of Iterations: Until convergence

l) Relaxation Factor: 0.5 for ART, SIRT and SART (0.1 for MART)

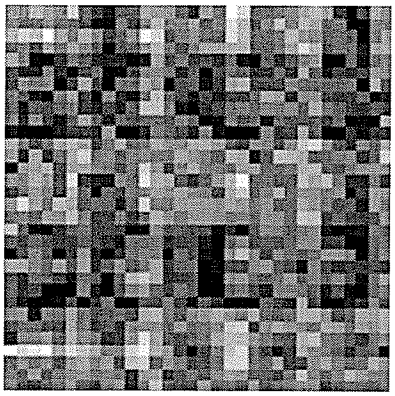
Figure 4-16 (b-d) shows the reconstructed images based on the flat seed, meshgrid seed and fbp seed respectively. Note that even though this experiment was performed for all seed images, only flat, meshgrid and fbp seeds are shown in the figure for illustrative purposes. Figure 4-16 (e) shows the plot of the normalized Euclidean distance of each of the seed image for MART.



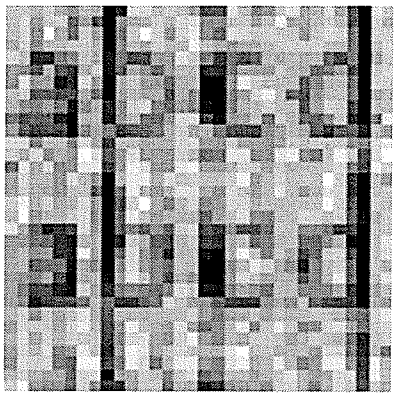
(a)



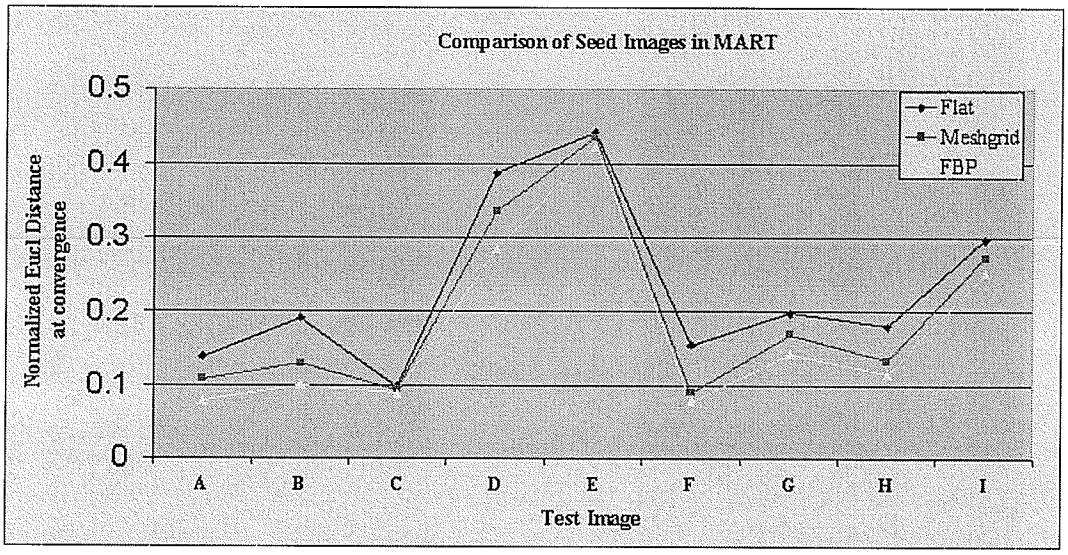
(b)



(c)



(d)



(e)

Seed Image Used	Mean Euclidean Distance	1σ
Flat	0.2318	0.1173
Meshgrid	0.1963	0.1227
FBP	0.1735	0.1201
Conclusion: Based on the experiment parameters, the mean Euclidean distance measure was least (best) for FBP seed image, followed by meshgrid and flat seeds.		

(f)

Seed Image	Mean Number of Cycles to converge
Flat	20
Meshgrid	22
FBP	22
Conclusion: Based on the experiment parameters, there is not much difference in the convergence speed between the seed images.	

(g)

Figure 4-16. *Comparison of Seed images in MART* **(a)** *Test Image* **(b)** *Reconstructed Image based on a Flat seed* **(c)** *Reconstructed Image based on a Meshgrid seed* **(d)** *Reconstructed Image based on FBP seed* **(e)** *Plot comparing the Euclidean distance of the reconstructed images based on flat, meshgrid and fbp seed* **(f)** *Summary of the Euclidean distance measure of the reconstructed images of the different seed images* **(g)** *Summary of the convergence (number of cycles required) for the different seed images.*

As one can see from Figures 4-16 (b-d), the visual quality of the reconstructions is very close to one another. Not much can be deduced just by looking at the reconstructed images. The Euclidean distance was used to provide a quantitative measure of the difference in the image quality between the different seeds. The plot in figure 4-16 (e) shows that the Euclidean distance for the FBP seed is the best. This is intuitive because the image reconstructed using the Filtered backprojection technique itself has brought the solution much closer to the actual solution. This also indicates that the FBP solution does not satisfy the linear equation criteria of CT reconstruction. The difference in the Euclidean distance of FBP when used as a seed in MART reconstruction over FBP reconstruction is shown in Figure 4-17.

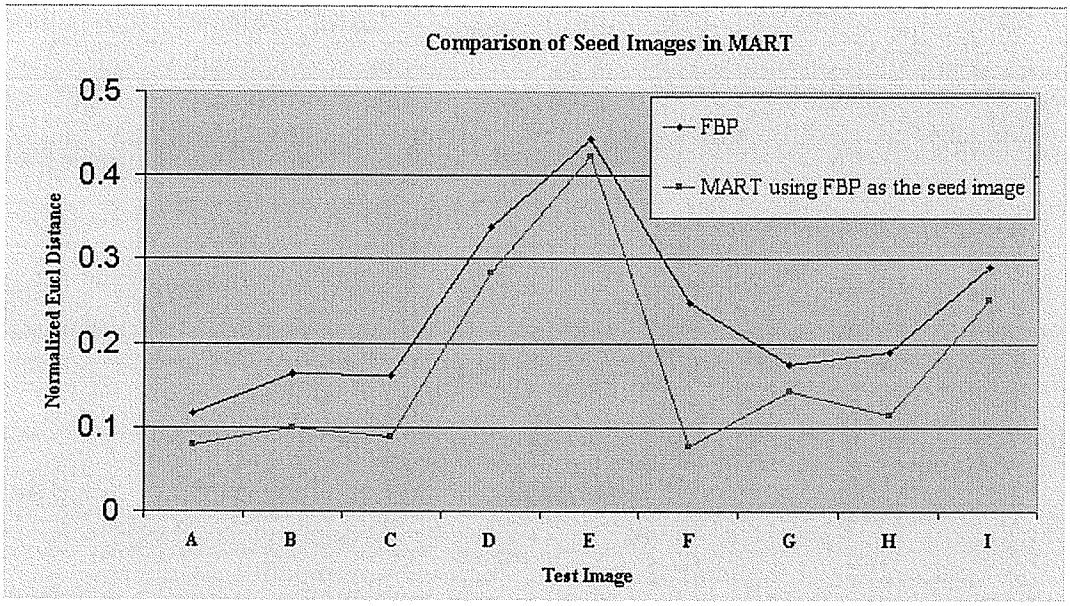


Figure 4-17. *Graph showing difference in the normalized Euclidean distance between FBP reconstruction and *ART reconstruction (when FBP is used as a seed).*

I compared the time required for the cycles to converge in case of flat, meshgrid and FBP seed images. Figure 4-16(g) shows that there is little difference in the convergence speed of the different seed images.

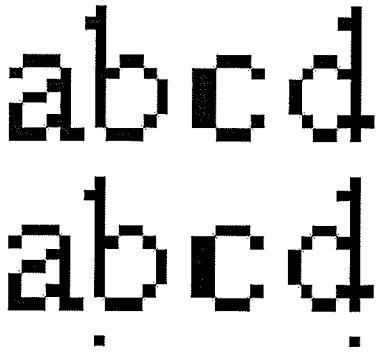
4.7.3 Comparison of the different projection angle ordering schemes in*ART

The experiments were done with *ART for all test images. All the parameters were kept constant except projection angle ordering scheme. The values for the different parameters are as below –

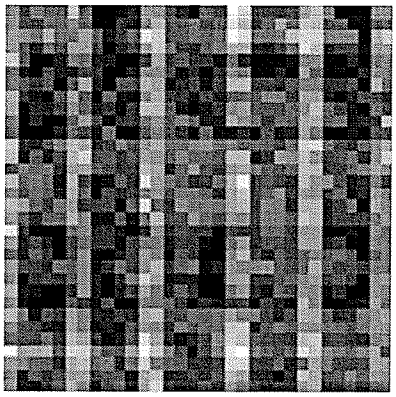
a) Projection Angles: 10 equally spaced projections from 0 to 180 degrees.
b) Projection angles ordering scheme: {SAS, FAAS, RAS, MLSAS, WDS}
c) Ray width: 1
d) Ray Gap: 0
e) Detector Width: 1
f) Detector Gap: 0
g) Pixel Width: 1
h) Weighting Scheme: "Distance of the center of pixel from center of ray" scheme
i) Test Image: Used all the test images shown in Figure 4-1.
j) Seed Image: Flat seed.
k) Number of Iterations: Until convergence
l) Relaxation Factor: 0.5 for ART, SIRT and SART, 0.1 for MART.

Figure 4-18 (b-f) shows the reconstructed images based on Sequential Access Scheme (SAS), Fixed Angle Access Scheme (FAAS), Random Access Scheme (RAS), Multilevel

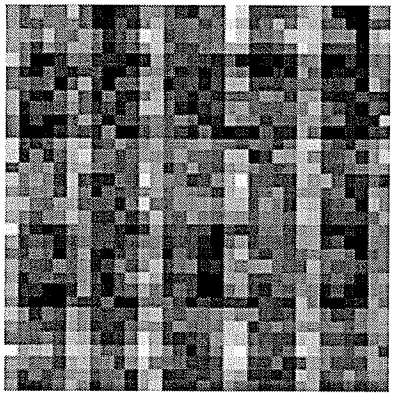
Resolution Select Access Scheme (MLSAS), and Weighted Distance Access Scheme (WDAS) respectively. Figure 4-18(g) shows the plot of the normalized Euclidean distance of each of the projection angle-ordering scheme for MART.



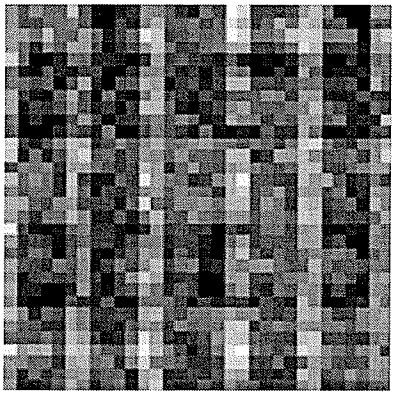
(a)



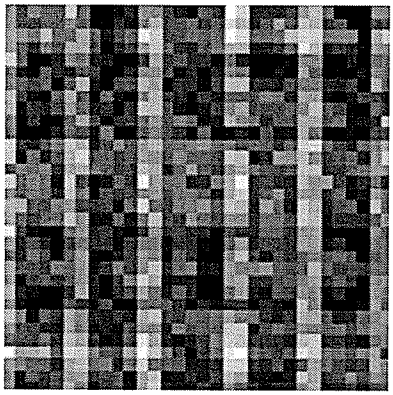
(b)



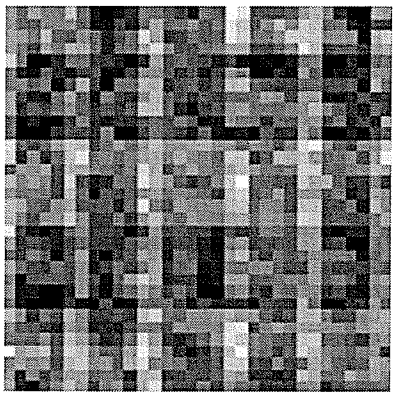
(c)



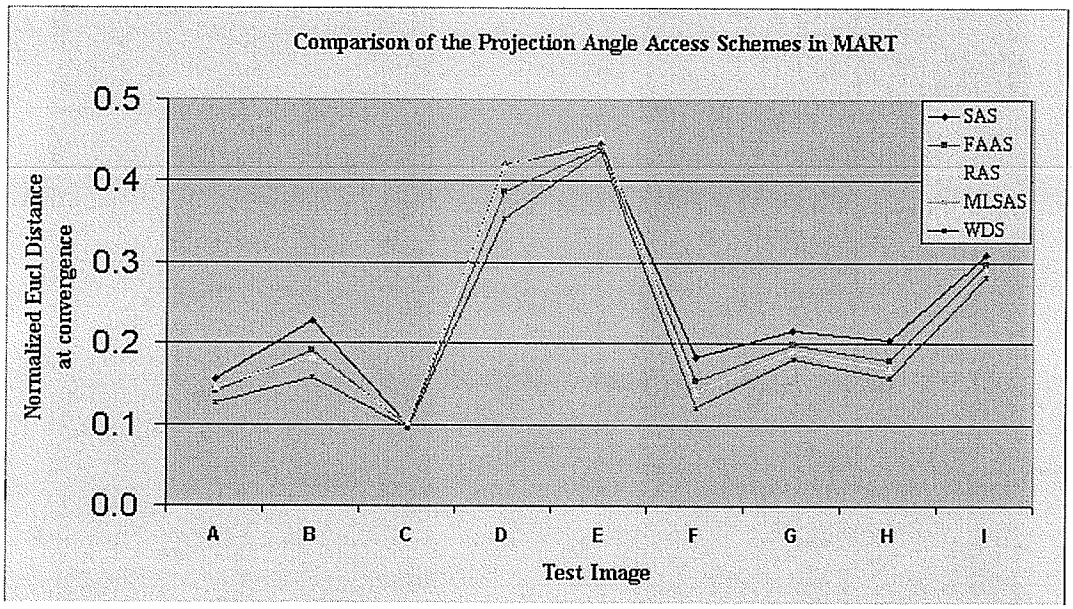
(d)



(e)



(f)



(g)

Projection Angle Ordering Scheme	Mean Euclidean Distance	1σ
Sequential Access Scheme (SAS)	0.2510	0.1176
Fixed Angle Access Scheme (FAAS)	0.2318	0.1173
Random Access Scheme (RAS)	0.2327	0.1272
Multilevel Resolution Select Access Scheme (MLSAS)	0.2532	0.1212
Weighted Distance Scheme (WDS)	0.2125	0.1184
Conclusion: Based on the experiment parameters, the mean Euclidean distance measure of the reconstructed image is best for WDAS, followed by MLSAS, FAAS, RAS and SAS.		

(h)

Figure 4-18. *Comparison of projection angle ordering schemes in MART* **(a)** *Test Image* **(b)** *Reconstructed Image based on Sequential Access Scheme (SAS)* **(c)** *Reconstructed Image based on a Fixed Angle Access Scheme (FAAS)* **(d)** *Reconstructed Image based on Random Access Scheme (RAS)* **(e)** *Reconstructed Image based on Multilevel Select Access Scheme (MLSAS)* **(f)** *Reconstructed Image based on Weighted Distance Access Scheme (WDAS)* **(g)** *Plot comparing the Euclidean distance of the reconstructed images based on the different projection angles ordering schemes* **(h)** *Summary of the Euclidean distance measure of the reconstructed images based on the different projection angles ordering scheme.*

Euclidean distance was used for comparison. The plot of the Euclidean distance measure for the different projection angles ordering scheme is shown in Figure 4-18 (g). The Weighted Distance Access Scheme gives the least (best) Euclidean distance measure as it heuristically optimizes the angular distance of a newly selected projection with respect to the complete sequence of all previously applied projections. The Multi Level resolution Select Access Scheme came in second. For test images A, C and H, the MLSAS gives better Euclidean distance than WDAS. The fixed angle-ordering scheme came next best to MLSAS. In the fixed angle-ordering scheme we are forcing the projection angles to be as orthogonal (requirement for subsequent projection data to be uncorrelated) as possible, unlike the Random Access Order scheme, where the orthogonality between subsequent projections is random. The Euclidean distance measure of random projection order came out better than sequential. This is because in the sequential access scheme the subsequent projections are the least uncorrelated among the different projection access ordering schemes.

4.7.4 Comparison of different pixel weighting scheme in*ART

The experiments were done with *ART for all test images. All the parameters were kept constant except the pixel weighting scheme. The values for the different parameters are as below –

a) Projection Angles: 10 equally spaced projections from 0 to 180 degrees.

b) Projection angles ordering scheme: FAAS

c) Ray width: 1

d) Ray Gap: 0

e) Detector Width: 1

f) Detector Gap: 0

g) Pixel Width: 1

h) Weighting Scheme: {

1. "Binary Scheme (BIN)",
2. "Length of pixel within pixel scheme (INT)",
3. "Distance of the center of pixel from center of ray (DIST)" ,
4. "Distance of center of pixel from adjacent ray scheme (CONT)" }

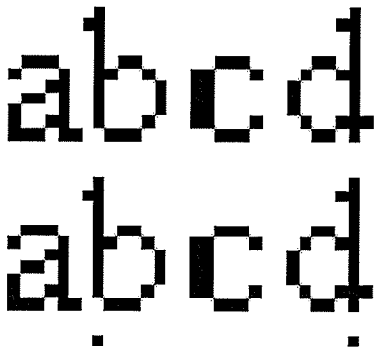
i) Test Image: Used all the test images shown in Figure 4-1.

j) Seed Image: Flat seed.

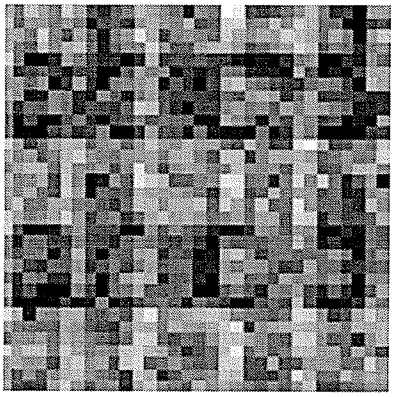
k) Number of Iterations: Until convergence

l) Relaxation Factor: 0.5 for ART, SIRT and SART, 0.1 for MART.

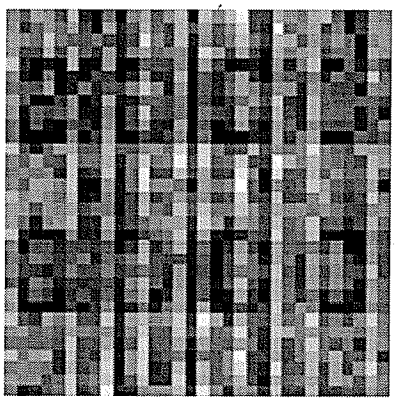
Figure 4-19 (b-e) shows the reconstructed images based on BIN, INT, DIST and CONT pixel weighting scheme respectively. Figure 4-19(f) shows the plot of the normalized Euclidean distance of each of the pixel weighting schemes for MART.



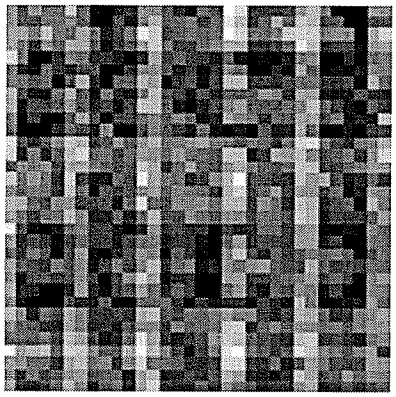
(a)



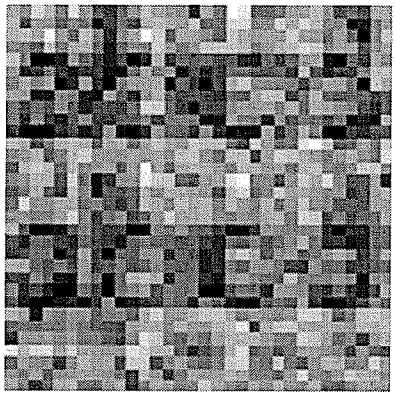
(b)



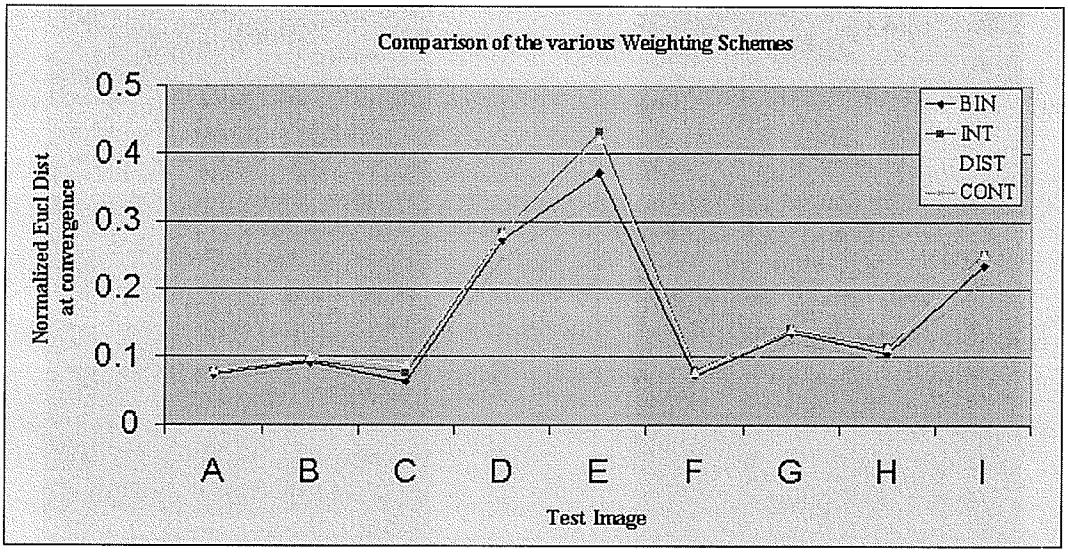
(c)



(d)



(e)


Comparison of the various Weighting Schemes
Normalized Eucl Dist at convergence
0.5
0.4
0.3
0.2
0.1
0
A
B
C
D
E
F
G
H
I
Test Image
BIN
INT
DIST
CONT

(f)

Pixel Weighting Scheme	Mean Euclidean Distance	1σ
Binary Scheme (BIN)	0.1579	0.1098
Length of Center of Ray in pixel (INT)	0.1728	0.1245
Distance of Center of Pixel from Center of Ray (DIST)	0.1736	0.1201
Distance of Center of Pixel from Adjacent Ray scheme (CONT)	0.1441	0.0988
Conclusion: Based on the experiment parameters, the mean Euclidean distance measure of the reconstructed image is best for CONT followed by BIN, DIST and INT.		

(g)

Figure 4-19. *Comparison of Weighting schemes in MART* **(a)** *Test Image* **(b)** *Reconstructed Image based on Binary Scheme (BIN)* **(c)** *Reconstructed Image based on "length of center of ray in pixel" scheme (INT)* **(d)** *Reconstructed Image based on "Distance of center of pixel from center of ray" scheme (DIST)* **(e)** *Reconstructed Image based on "Distance of Center of Pixel from Adjacent Ray scheme" (CONT)* **(f)** *Plot comparing the Euclidean distance of the reconstructed images based on the different weighting schemes* **(g)** *Summary of the Euclidean distance measure of the reconstructed images based on the different weighting schemes.*

Euclidean distance is the image quality measure used for this experiment. The plot of the Euclidean distance measure for the different pixel weighting schemes is shown in Figure 4-19 (f). The INT weighting scheme showed the worst Euclidean distance. This is because the complete contribution of the pixel is not considered. This scenario is illustrated in Figure 4-8(a). The DIST weighting scheme showed a better Euclidean distance than INT. Even DIST suffers from the problem of not considering the true contribution of pixel (illustrated in Figure 4-8(b)). However, the likelihood of deviating from the 100% contribution of the pixel on the detectors is more in case of INT than in DIST because the necessary condition for 100% contribution in INT is met only by the ray passing at a 45° degree angle through the diagonal points of the pixel. In case of DIST the 100% contribution is met by any ray passing through the center of the pixel regardless

of its angle. Hence the improvement in the Euclidean distance measure of DIST over INT was expected. The binary scheme does not give a true contribution but forces all pixels to contribute 100% on exactly one ray at a given angle. Hence the BIN scheme gave a better Euclidean distance. The CONT scheme forces all pixel to contribute 100% on the detectors and at the same time measures the contribution relative to the position of the pixel and the ray. Hence the CONT scheme gives the best Euclidean distance.

Once the weight matrix is calculated, the time taken by any of the reconstruction algorithm only depends on the algorithm type (ART, MART, SIRT and SART). However, the time taken to calculate the weight matrix in case of INT and CONT is much larger than the time taken to calculate the weight matrix in BIN and DIST.

4.8 Conclusion

The object of the experiments was to come up with as many permutations and combinations of CT parameters as possible to arrive at an optimal parameter solution. The object was accomplished successfully. Based on the experiments performed and using Euclidean distance as the quantitative measure for comparing the reconstruction quality, I conclude that the below parameters gives an optimal solution for *ART.

Typical Parameters for best *ART solution

Projection Angle Ordering Scheme: Weighted Distance Access Scheme

Weighting Scheme: "Distance of center of pixel from adjacent ray scheme"

Algorithm Type: SART

Seed Image: FBP

Similarly the typical parameters for the worst solution of *ART are –

Typical Parameters for worst *ART solution

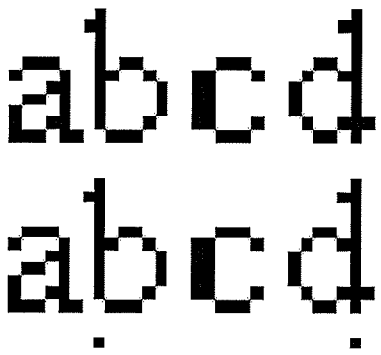
Projection Angle Ordering Scheme: Sequential Access Scheme

Weighting Scheme: "Length of center of ray in pixel scheme"

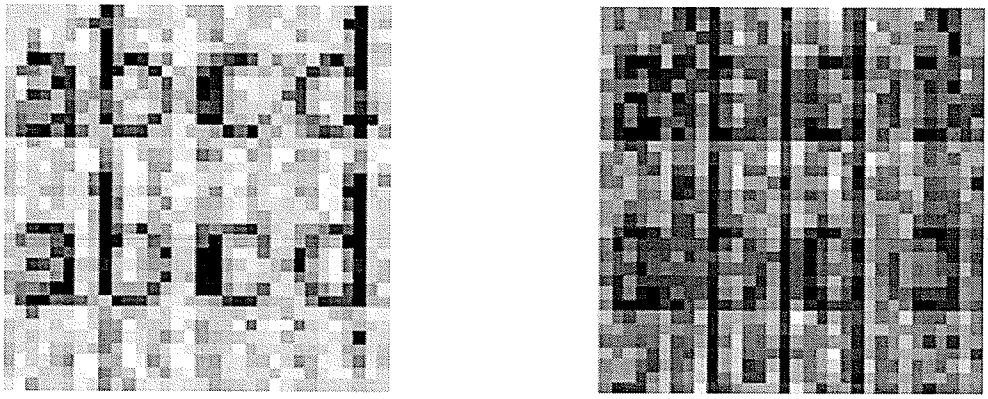
Algorithm Type: MART

Seed Image: Flat (there are others like blur and stretch but since they are not presented above, I did this experiment with the flat seed)

Figure 4-20 (b and c) shows the difference between the reconstruction images obtained by the best case *ART parameters and the worst case *ART parameters. Just by looking at the images we can see the improvement in the reconstruction quality. This suggests that the space of the parameters may be single peaked.



(a)



(b) (c)

Figure 4-20. *Comparison of the best-case parameters and the worst-case parameters in *ART* **(a)** Test Image* **(b)** *Reconstructed Image based on the best case *ART parameters* **(c)** Reconstructed Image based on the worst case *ART parameters.*

In general the weight matrix coding style for *ART is a good way of writing *ART code. It provides the ease of implementing different weighting schemes; projection angle ordering schemes fairly easily and also gives the advantage of faster execution.

Pros of using the Weight Matrix approach –

a) Simple to code

b) Flexibility in implementation

c) Faster execution

Cons of using the Weight Matrix approach –

a) Restricted to size of image and projection data due to the large size of weight matrix created. But with the decreasing cost of memory this may not be a problem in the future.

Chapter 5

Future Work

5.1 Future Work

The comparison between square pixels and hexagonal pixels presented in chapter 2 is biased towards hexagonal pixels because they have fewer pixels compared to the square pixels. i.e., each hexagonal pixel has 62 pixels compared to 64 pixels in square pixels. Experiments need to be done such that both the pixel-methods contain the same number of pixels. Also the test images used in the square pixel and hexagonal pixel resolution comparison experiment are either created mathematically or acquired with a digital camera. The experiments need to be performed on more samples of real life CT data to see if one pixel-method can be preferred over another in CT imaging. The center of rotation is not the center of the centermost hexagon. Had this been not the case we would have seen uniformity in the hexagonal plots at angles that are multiples of 60° (example 0°, 60°, 120° etc). The algorithm should create the hexagons such that the center of the centermost hexagon is the center of the image. It was interesting to note that the hexagonal pixel consistently gave a slightly better resemblity measure over square pixel. It will be interesting to see the effect of other quality measures on the two pixel methods. The experiments need to be performed by employing different tiling techniques other than a hexagonal grid for example work using a triangular tessellation is warranted, since it is the only other regular tessellation of the plane.

In chapter 4, the different factors that affect the reconstruction quality of *ART algorithms are presented. The experiments presented in chapter 4 show the results of the effect of algorithm type (ART, MART, SIRT and SART), seed images, the pixel-weighting scheme and the projection angle-ordering scheme. The experiments on *ART presented dealt only with parallel CT. The effect of seed images needs to be tested on cone beam and helical CT.

Only five different projection angle-ordering schemes were compared. The weighted distance approach gives the best result (in terms of Euclidean distance measure) among the ones that were compared. However, the weighted distance approach considers the history of past-applied projections and is hence not "global" in its real sense. Considering 10 projection angles, there is a possibility of 3,628,800 (10 factorial) different permutations of angles possible out of which one or more would be the optimal solution. The object of the projection angle-ordering scheme is to find the angle-order that gives the least value for the dot product of subsequent angles, but other optimization functions are conceivable. Work in this area is required. Work has already been done [26] in comparing various angle-ordering scheme in cone beam CT. However, work needs to be done in obtaining optimal angle-ordering schemes for cone beam and helical CT.

Work is being done in the area of using the scattering photon information. Since *ART algorithms give good reconstruction when a priori information is applied, the algorithms will come in very handy when the research data becomes available with scattering photons. Also CT of moving body organs like the heart and lungs face the challenges of distortions due to the organ movement. *ART is extremely useful in places where ray tracing techniques need to be employed, The ray tracing techniques can be modified to

organ movement techniques to make it extremely useful in CT imaging of moving organs. Between successive scans of the breast of a woman, the physical characteristics of the breast typically change. Image subtraction techniques are employed in this case. Image subtraction techniques can be combined with *ART to give more appropriate results.

Euclidean distance was used to quantitatively compare the quality of the images in the *ART experiments. The fundamental difficulty of Euclidean distance is that it calculates the quality of the image precisely, which could be different from human perception. Since the images are interpreted visually other image quality measures should be used.

The weight factors calculated are stored in a matrix. The matrix is 4 dimensional and is extremely large. However, most of the factors in the weight matrix are 0. My code did not consider compressing the weight matrix and hence I had lot of difficulty in running the program for larger images. Compressing the weight matrix will enable the implementation of the program in real CT data system. This needs to be done.

While Computed Tomography allows determination of relatively static zones, it shows serious limitations in regions with physiologic motion. The main challenge is to give the physician the best functionality needed to take into account the dynamic nature of a living human body having moving organs. This is a key issue for diagnostic imaging where motion induces blurred images and for all interventional procedures where the organ is a target and motion compensation is of first importance to reach the region of interest and

preserve nearby organs. especially chest and abdomen. The modality CT fluoroscopy (CTF or continuous imaging CT) has become the usual imaging technique for real-time guidance during biopsy of pulmonary nodules [33]. But the key issue in X-ray CT fluoroscopy is the dose delivery. Dose reduction can be achieved using motion compensated reconstruction. The use of weight matrix could be extended to construct a good motion compensated reconstruction algorithm. Depending upon the a priori knowledge of the physical characteristics of the organ, different weight matrices could be pre-developed and stored in memory. Reconstructions based on projection data from limited dose CT can be made fast and of better quality using these weight matrices. This will expand the possibilities for minimal invasive therapeutic procedures, the surgery of the next century.

Appendix A

Matlab™ Code

The source codes listed in this appendix are written in Matlab™ 6.0 (for windows) and used throughout the thesis. They consist of two sections. Section A.1 contains the programs written for the square and hexagon pixel comparison experiment. Section A.2 contains the programs written for the*ART experiment. Some of the test images were generated mathematically. Code for generating the test images is not included in this appendix.

A.1 Code for comparing square pixel and hexagonal pixel resolution

Extract all the files in this section into a particular folder on your computer. Change your Matlab™ session's present working directory to this folder. Simply type project1_main in the Matlab™ command prompt for comparing the resolution based on Euclidean distance, fidelity measure, resemblance measure, hausdorff distance and entropy. For comparing resolution based on modulation transfer function (MTF) type project1_mtf in the Matlab™ command prompt. Note that you will need test images for using the code. The different test images (all are 256 by 256 images) that I used are tabulated below.

Sr#	Test Image File Name	Description	How the image was generated?
1.	admin256.bmp	University of Manitoba Administration Building	Reproduced with Prof.W. Lehn's permission from his Digial Image Processing class.
2.	balcony256.bmp	Three friends standing in the balcony of a high rise building	Produced using a digital camera
3.	phantom256.bmp	Phantom Image	Produced using the phantom function in Matlab™
4.	rand256.bmp	Uniformly distributed	Produced using the rand function in

		random Image	Matlab™.
5.	square256.bmp	Image containing 8 by 8 squares of the same pixel value.	Created using the square_pixel_image function.
6.	hexagon256.bmp	Image containing hexagons of length 4.5 of the same pixel value	Created using the hexagon_pixel_image function
7.	sinewave01_256.bmp	Image containing sine wave of frequency 1	Created using the sine_wave function
8.	sinewave10_256.bmp	Image containing sine wave of frequency 10	Created using the sine_wave function

project1_main.m

```
%=========================================================================================================================
%  Created By : Girish Tirunelveli
%  Created On : June 8, 2001
%  File Name  : project1_main.m
%  Description: This program when executed in Matlab, will ask the user to enter the length of the square
%               and hexagon pixel. It will also ask the  user to enter the image  he wants to compare the
%               resolution for. After accepting  the user parameters, the  program will rotate  the image
%               from 0 to 360 degrees in increments of 5 and  compute the square and hexagon image quality
%               based on different algorithms (Euclidean Distance, Fidelity, Resemblance, Hausdorff dstnce
%               and Entropy). The program takes 4.5 hours to process a 256 by 256 image.
%=========================================================================================================================
% Clear the picture window and clear all variables
clf
clear

%--------------------------------
% Accept the input parameters.
%--------------------------------
file = input('Enter name of image file (.bmp only) to perform resolution comparison:  ','s');
original_image=imread(file, 'bmp');

length_of_square_pixel  = input ('Enter length of square pixel: ');
length_of_hexagon_pixel = input ('Enter length of hexagon pixel: ');
theta_in_degrees=[0:5:360];

time_start=cputime;

%-------------------------------------------------------------------------------
% For each rotation angle, get the square pixel image, the hexagon pixel image,
% the image quality using Euclidean distance, fidelity measure, resemblity
% measure and Hausdorff distance.
%-------------------------------------------------------------------------------
for angle_number=1:length(theta_in_degrees)

    % Rotate the image
    rotated_image = imrotate(original_image, theta_in_degrees(angle_number), 'bicubic', 'crop');
    %rotated_image = rotate(original_image, theta_in_degrees(angle_number));
    rotated_image = double(rotated_image);

    % Convert the rotated_image to square_pixel_image and get the image_quality
    % using Euclidean distance between the square and rotated image.
    SPI=square_pixel_image (rotated_image, length_of_square_pixel);

    % Convert the rotated_image to hexagon_pixel_image
    HPI=hexagon_pixel_image_2template (rotated_image, length_of_hexagon_pixel);

    % Calcuate the image quality using Euclidean distance.
    SIQ_ed(angle_number) = image_quality_using_ed (rotated_image, SPI);
    HIQ_ed(angle_number) = image_quality_using_ed (rotated_image, HPI);

    % Calcuate the image quality using fidelity measure.
    SIQ_fd(angle_number) = image_quality_using_fd (rotated_image, SPI);
    HIQ_fd(angle_number) = image_quality_using_fd (rotated_image, HPI);

    % Calcuate the image quality using resemblity measure.
    SIQ_rb(angle_number) = image_quality_using_rb (rotated_image, SPI);
    HIQ_rb(angle_number) = image_quality_using_rb (rotated_image, HPI);

    % Calcuate the image quality using hausdorff distance
    SIQ_hd(angle_number) = image_quality_using_hd (rotated_image, SPI);
    HIQ_hd(angle_number) = image_quality_using_hd (rotated_image, HPI);

    % Get entropy (information content) of each of the images
```

```
    % and compare the preservation of information in square_pixel image
    % compared to hexagon_pixel_image.
    [histogram_rotated_image, entropy_rotated_image] = entropy (rotated_image);
    [histogram_SPI, entropy_SPI]                     = entropy (SPI);
    [histogram_HPI, entropy_HPI]                     = entropy (HPI);

    SIQ_en(angle_number) = entropy_SPI/entropy_rotated_image;
    HIQ_en(angle_number) = entropy_HPI/entropy_rotated_image;

end;  % for angle_number

time_end=cputime;

disp(strcat('Process Completed. Time taken:  ', num2str(time_end-time_start), ' seconds'));

%-----------------------------------------------------------
% Plot the Euclidean distance image quality difference.
%-----------------------------------------------------------
clf
plot(theta_in_degrees, SIQ_ed, 'r+-');
hold
plot(theta_in_degrees, HIQ_ed, 'bo-');
title  ('Resolution Comparison between square and hexagon pixel using Euclidean distance')
xlabel ('Rotation (in degrees)')
ylabel ('Normalized Euclidean Distance');
legend ('Square Pixel', 'Hexagon Pixel');
pause

%-----------------------------------------------------------
% Plot the fidelity image quality measure.
%-----------------------------------------------------------
clf
plot(theta_in_degrees, SIQ_fd, 'r+-');
hold
plot(theta_in_degrees, HIQ_fd, 'bo-');
title  ('Resolution Comparison between square and hexagon pixel using Fidelity measure');
xlabel ('Rotation (in degrees)')
ylabel ('Fidelity Measure');
legend ('Square Pixel', 'Hexagon Pixel');
pause

%-----------------------------------------------------------
% Plot the resemblity image quality measure.
%-----------------------------------------------------------
clf
plot(theta_in_degrees, SIQ_rb, 'r+-');
hold
plot(theta_in_degrees, HIQ_rb, 'bo-');
title  ('Resolution Comparison between square and hexagon pixel using Resemblity measure')
xlabel ('Rotation (in degrees)')
ylabel ('Resemblity Measure');
legend ('Square Pixel', 'Hexagon Pixel');
pause

%-----------------------------------------------------------
% Plot the Hausdorff Distance measure.
%-----------------------------------------------------------
clf
plot(theta_in_degrees, SIQ_hd, 'r+-');
hold
plot(theta_in_degrees, HIQ_hd, 'bo-');
title  ('Resolution Comparison between square and hexagon pixel using Hausdorff Distance');
xlabel ('Rotation (in degrees)')
ylabel ('Hausdorff Distance');
legend ('Square Pixel', 'Hexagon Pixel');
pause

%-----------------------------------------------------------
% Plot the entropy image quality measure.
%-----------------------------------------------------------
clf
plot(theta_in_degrees, SIQ_en, 'r+-');
hold
plot(theta_in_degrees, HIQ_en, 'bo-');
title  ('Resolution Comparison between square and hexagon pixel using Entropy');
xlabel ('Rotation (in degrees)')
ylabel ('Entropy');
legend ('Square Pixel', 'Hexagon Pixel');
```

square_pixel_image.m

```
%=============================================================================================
% Created By : Girish Tirunelveli
% Created On : June 19, 2001
% File Name  : square_pixel_image.m
% Description: This program accepts the image which needs to be converted into
%              square pixel image. The second parameter is the length of the
%              square pixel desired.
```

```
%================================================================================
function [square_pixel_image]=square_pixel_image(original_image,length_of_square_pixel)

    %-------------------------------------------------------------------------
    % Step1. Assign the output variables as 0, just in case the job abhends
    %        in the middle. Also store all the values required for this
    %        program in variables.
    %-------------------------------------------------------------------------
    original_image = double (original_image);
    square_pixel_image (size(original_image)) = 0;
    [no_of_pixels_in_x_direction, no_of_pixels_in_y_direction] = size(original_image);
    no_of_tsi_pixels_in_x_direction = ceil (no_of_pixels_in_x_direction/length_of_square_pixel);
    no_of_tsi_pixels_in_y_direction = ceil (no_of_pixels_in_y_direction/length_of_square_pixel);

    %-----------------------------------------------------------------------------------------
    % Step3. Create the tiny_square image based on the original image such
    %        such that the value of hte pixel in the tiny_square image is
    %        the average of all the pixels that falls in the tiny square
    %        image. Initialize the tiny_square_image pixel values to 0.
    %        Note that tsi in the variable names indicate tiny square image.
    %-----------------------------------------------------------------------------------------
    tiny_square_img       (1:no_of_tsi_pixels_in_x_direction, 1:no_of_tsi_pixels_in_y_direction) = 0;
    no_of_pixels_in_square(1:no_of_tsi_pixels_in_x_direction, 1:no_of_tsi_pixels_in_y_direction) = 0;

    for row_no = 1:no_of_pixels_in_x_direction
        for column_no = 1:no_of_pixels_in_y_direction
            %-------------------------------------------------------------------------------
            % Find the integer value of the division of row_no by length_of_square_pixel
            % and the column_no by length_of_square, because that becomes the index
            % value of the tiny_square_image.
            %-------------------------------------------------------------------------------
            tsi_row_no = floor((row_no+length_of_square_pixel-1)/length_of_square_pixel);
            tsi_column_no = floor((column_no+length_of_square_pixel-1)/length_of_square_pixel);
            tiny_square_img(tsi_row_no, tsi_column_no) = tiny_square_img(tsi_row_no, tsi_column_no) +
original_image(row_no, column_no);
            no_of_pixels_in_square(tsi_row_no, tsi_column_no) = no_of_pixels_in_square(tsi_row_no, tsi_column_no) + 1;
        end;
    end;

    %---------------------------------------------------------------------
    % Take the average of the pixel values. In one square there are
    % length_of_square_pixel^2 pixels
    %---------------------------------------------------------------------
    tiny_square_img(1:no_of_tsi_pixels_in_x_direction, 1:no_of_tsi_pixels_in_y_direction) = ...
           tiny_square_img(1:no_of_tsi_pixels_in_x_direction, 1:no_of_tsi_pixels_in_y_direction)./  ...
                (no_of_pixels_in_square(1:no_of_tsi_pixels_in_x_direction, 1:no_of_tsi_pixels_in_y_direction));

    %-------------------------------------------------------------------------
    % Step4: For quality measurement, it will be simpler if we enlarge the
    %        tiny_square image to the size of the original image. Create
    %        the enlarged_square_img matrix which contains the pixel value
    %        of the enlarged square image.
    %-------------------------------------------------------------------------
    for esi_row_no = 1:no_of_pixels_in_x_direction
        for esi_column_no = 1:no_of_pixels_in_y_direction
            tsi_row_no = floor((esi_row_no+length_of_square_pixel-1)/length_of_square_pixel);
            tsi_column_no = floor((esi_column_no+length_of_square_pixel-1)/length_of_square_pixel);
            enlarged_square_img(esi_row_no, esi_column_no) = tiny_square_img(tsi_row_no, tsi_column_no);
        end;
    end;

    square_pixel_image = enlarged_square_img;
```

hexagon_pixel_image_2template.m

```
%================================================================================
% Created By : Girish Tirunelveli
% Created On : June 19, 2001
% File Name  : hexagon_pixel_image.m
% Description: This program accepts the image which needs to be converted into
%              hexagon pixel image. The second parameter is the length of the
%              hexagon pixel desired.
% Algorithm:
% For comparison with hexagonal resolution, we have to break the original image
% into tiny hexagonal image. For better comparison purposes the program should
% give the choice while reconstruction. Either it should have the same number
% of hexagons as the square reconstruction or there should be equal number of
% pixels within one hexagon as there would be in a square. Note that all the
% hexagons are considered in the following orientation: -
%                            F ----------------- E
%                             /        5        \
%                            /6                 4\
%                           /                     \
%                          /                       \
%                       A \                        / D
%                          \1                    3/
%                           \                    /
```

```
%                            \         2       /
%                           B ----------------- C
%
% Note that the number of horizontal and vertical hexagons differ as the
% horizontal distance between the farthest points that lie within the hexagon is
% more than the vertical distance. Note that the length_of_hexagon_pixel
% parameter will later depend only on the optimizer_mode. This program will
% calculate the length_of_hexagon_pixel parameter.
%
% This program is copied from hexagon2.m to test for all fractional
% length_of_hexagon_pixel.
%         H11     H12     H13
%             H21     H22     H23
%         H31     H32     H33
%             H41     H42     H43....
% This program will move the rectangular template in the below order -
%         H11     H12     H13
%
%         H31     H32     H33
% and once all the odd layers are done, it will start doing the even layers in the
% below sequence -
%         H21     H22     H23
%
%         H41     H42     H43
% The hexagon conventions are still the same. Note the orientation below -
%                           F ----------------- E
%                            /|       5        \
%                           /6|                 4\
%                          /  |L2                 \
%                         /L1 |                    \
%                      ---------------------------
%                         A \                       / D
%                            \1                    3/
%                             \                    /
%                              \         2       /
%                           B ----------------- C
% Imagine segments joining vertices AD and FB.
%     The perpendicular distance between FB and vertex A =
%                                L1 = length_of_hexagon_pixel * Cos60
%     The perpendicular distance between AD and vertex F =
%                                L2 = length_of_hexagon_pixel * Sin60
%=========================================================================================
function [hexagon_pixel_image]=hexagon_pixel_image(original_image,length_of_hexagon_pixel)
    %-------------------------------------------------------------------------
    % Step1. Assign the output variables as 0, just in case the job abhends
    %        in the middle. Also store all the values required for this
    %        program in variables.
    %-------------------------------------------------------------------------
    original_image = double (original_image);
    hexagon_pixel_image (size(original_image)) = 0;
    [no_of_pixels_in_x_direction, no_of_pixels_in_y_direction] = size(original_image);

    %-------------------------------------------------------------------------------------------------------
    % Step2. Initialize the number of pixels in x and y direction of the hexagonal image. Note that
    %        x is the vertical direction and y is the horizontal direction. Note that we are going
    %        to have some restrictions on the value that the length_of_hexagon_pixel is going to have.
    %        For understanding the variable names see the hexagon picture above.
    %-------------------------------------------------------------------------------------------------------
    L1 = length_of_hexagon_pixel * cos(pi/3);
    L2 = length_of_hexagon_pixel * sin(pi/3);

    % Calculate the increment for the top left corner along the same layer.
    increment_top_left_x = 0;
    increment_top_left_y = round(length_of_hexagon_pixel * 3);

    %-------------------------------------------------------------------------------------------
    % Calculate the number of hexagons in x and y direction that will fit in the original image.
    % Bogus this calculation has to be revised for accuracy.
    %-------------------------------------------------------------------------------------------
    no_of_hexagons_in_y_direction = ceil(no_of_pixels_in_y_direction / (3 * length_of_hexagon_pixel));
    no_of_hexagons_in_x_direction = ceil((no_of_pixels_in_x_direction) / (length_of_hexagon_pixel*0.8660254));


    %-------------------------------------------------------------------------------------------------------
    % Initialize all the hexagons that fit in the original image to 0. The program follows the
    % accumulative logic. Also note that the origin in matlab is (1,1). While converting this
    % program to C, this must be taken care of. Also this program considers two template approach.
    % Template1 is first created using the regular equations of a hexagon. Template 2 is created
    % containing all pixels that are not already taken up by template1. To indicate whether or not
    % the pixel is not taken already by template1 create a matrix variable called is_pixel_taken
    % and initialize to 0.
    %-------------------------------------------------------------------------------------------------------
    hexagon_img(1:no_of_hexagons_in_x_direction, 1:no_of_hexagons_in_y_direction) = 0;
    no_of_pixels_in_hexagon(1:no_of_hexagons_in_x_direction, 1:no_of_hexagons_in_y_direction) = 0;
    is_pixel_taken(1:no_of_pixels_in_x_direction, 1:no_of_pixels_in_y_direction) = 0;


    %-------------------------------------------------------------------------------------------
    % The algorithm of the program is written as below :-
    % 1. Create a rectangular template that acts a circum-rectangle to one hexagon.
    % 2. Move the template over the entire image.
    % 3. Within the template the equations of all 6 sides of the hexagon are determined.
    % 4. The template movement is started from the first point on the top-left corner of the
```

```
%    image. Each pixel in the original image that lies inside this rectangular template
%    is analysed to determine whether or not they lie inside the hexagon.
% 5. If they do then the pixel value at that point is accumulated against the hexagonal
%    image array. Also the number of pixels that went into the accumulation is calculated.
% 6. Once this first pass is over then the same routine is done, then the average value at
%    each hexagonal array is calculated.
% 7. The routine is traversed again to unallocate the averaged hexagonal array value to the
%    pixels in the original image that lie within the hexagon.
%-------------------------------------------------------------------------------------------
for hex_x = 1:2:no_of_hexagons_in_x_direction       % This will get all the odd layers.
    for hex_y = 1:no_of_hexagons_in_y_direction

        %-------------------------------------------------------------------------------------------
        % Calculate the co-ordinates of the top_left and bottom-right corners of the rectangle that
        % exactly covers the entire hexagon. The calculation of these co-ordinates should depend
        % only on length_of_hexagon_pixel and the hexagon number(hex_x, hex_y).
        %-------------------------------------------------------------------------------------------
        if (hex_x == 1) & (hex_y == 1)      % Indicates the first hexagon.
            top_left_x = 1;
            top_left_y = 1;
            bottom_right_x = top_left_x + round(2*L2);
            bottom_right_y = round(top_left_y + (2*L1) + length_of_hexagon_pixel-1);  %GT
            vertical_no_of_pixels = bottom_right_x - top_left_x + 1;

            %-------------------------------------------------------------------------------
            % Bogus create variables to store the row number of the "A" pixel of the first
            % hexagon in the odd layer. This is used for determining the top_left_x position
            % of the first hexagon in the even layer. Similary create a variable to store the
            % column number of the "E" pixel of the first hexagon in the odd layer. This will
            % be used in determining the top_left_y position of the first hexagon in the even
            % layer.
            %-------------------------------------------------------------------------------
            A_x1 = top_left_x + L2;
            E_y1 = top_left_y + L1 + length_of_hexagon_pixel-1;

        elseif (hex_y == 1)                 % Indicates the start of a new layer of hexagons
                top_left_x = top_left_x + vertical_no_of_pixels;
                top_left_y = 1;

        else % Indicates that the hexagon is in the same layer as the previous one that was considered.
            top_left_x = top_left_x + increment_top_left_x;
            top_left_y = top_left_y + increment_top_left_y;

        end;    % end if (hex_x == 1) & (hex_y == 1)

        bottom_right_x = top_left_x + round(2*L2);
        bottom_right_y = round(top_left_y + (2*L1) + length_of_hexagon_pixel-1);   %GT


        %-------------------------------------------------------------------------------------------
        % To draw and identify all hexagons, each hexagon is given a different pixel value.
        % All odd layer (see documentation) hexagons are given a pixel value of 75 and 225 while all
        % even layer hexagons are given a pixel value of 150 and 300 respectively.
        %-------------------------------------------------------------------------------------------
        if (rem(hex_x, 4) == 0)     % Indicates every second even layer (eg L4, L8, L12 ...)
            pixel_value = 300;
        elseif (rem(hex_x, 4) == 1) % Indicates every second odd layer (eg L3, L7, L11 ...)
            pixel_value = 225;
        elseif (rem(hex_x, 4) == 2) % Indicates every first even layer (eg L2, L6, L10 ...)
            pixel_value = 150;
        else                                % Indicates every first odd layer (eg L1, L5, L9 ...)
            pixel_value = 75;
        end;

        %-------------------------------------------------------------------------------------
        % Determine the coordinates of the vertices of the hexagon based on the top left corner.
        % Note that in the calculations below there should be only the top_left_x, top_left_y,
        % sin60, cos60 and length_of_hexagon_pixel variable. If we have more then the calculation is
        % incorrect. Bogus Try the logic first without rounding the vertices and then by rounding
        % them.
        %-------------------------------------------------------------------------------------
        A_x = top_left_x + L2;
        A_y = top_left_y;
        B_x = top_left_x + 2*L2;
        B_y = top_left_y + L1;
        C_x = top_left_x + 2*L2;
        C_y = top_left_y + L1 + length_of_hexagon_pixel-1;
        D_x = top_left_x + L2;
        D_y = top_left_y + (2* length_of_hexagon_pixel -1);
        E_x = top_left_x;
        E_y = top_left_y + L1 + length_of_hexagon_pixel-1;
        F_x = top_left_x;
        F_y = top_left_y + L1;


        %-------------------------------------------------------------------------------------------
        % Calculate the slope of segments AB, BC, CD, DE, EF and FA. Note that the slopes of
        % segments BC and EF will be 0. Note that the geometry is opposite of conventional geometry
        % and also the origin is (1,1) instead of (0,0) as in conventional geometry.
        %-------------------------------------------------------------------------------------------
        slope_ab = (B_x - A_x)/(B_y - A_y);
        slope_bc = (C_x - B_x)/(C_y - B_y);
        slope_cd = (D_x - C_x)/(D_y - C_y);
```

```
            slope_de = (E_x - D_x)/(E_y - D_y);
            slope_ef = (F_x - E_x)/(F_y - E_y);
            slope_fa = (A_x - F_x)/(A_y - F_y);

            %-----------------------------------------------------------------------------------------------
            % After calculating the top_left and bottom_right corner within which each hexagon will fall,
            % loop only across this rectangle of pixels to determine which pixels fall inside and which
            % pixels fall outside the hexagon.
            %-----------------------------------------------------------------------------------------------
            for x_loop = top_left_x: bottom_right_x
                for y_loop = top_left_y: bottom_right_y

                    boo_outside_hexagon = 0;            % Initialize the boo_outside_hexagon variable to indicate that
the pixel is inside the hexagon.
                    if round((x_loop - B_x)) > round(slope_ab * (y_loop - B_y))
                        boo_outside_hexagon = 1;
                    end;
                    if round((x_loop - B_x)) > round(slope_bc * (y_loop - B_y))
                        boo_outside_hexagon = 2;
                    end;
                    if round((x_loop - C_x)) > round(slope_cd * (y_loop - C_y))
                        boo_outside_hexagon = 3;
                    end;

                    if round((x_loop - E_x)) < round(slope_de * (y_loop - E_y))
                        boo_outside_hexagon = 4;
                    end;
                    if round((x_loop - F_x)) < round(slope_ef * (y_loop - F_y))
                        boo_outside_hexagon = 5;
                    end;
                    if round((x_loop - F_x)) < round(slope_fa * (y_loop - F_y))
                        boo_outside_hexagon = 6;
                    end;

                    if (boo_outside_hexagon == 0)   % Indicates that the pixel is inside a hexagon.
                        if (x_loop <= no_of_pixels_in_x_direction & y_loop <= no_of_pixels_in_y_direction)

                            hexagon_img(hex_x, hex_y) = hexagon_img(hex_x, hex_y) + original_image(x_loop, y_loop);
                            no_of_pixels_in_hexagon(hex_x, hex_y) = no_of_pixels_in_hexagon(hex_x, hex_y) + 1;

                            %-------------------------------------------------------------------------
                            % Store the hex_x value in another matrix variable to indicate that the
                            % pixel(x_loop, y_loop lies in the hexagon whose hex_x value is so-and-so.
                            % This way we wouldn't have to write the re-allocating algorithm.
                            %-------------------------------------------------------------------------
                            original_img_lies_in_x(x_loop, y_loop) = hex_x;
                            original_img_lies_in_y(x_loop, y_loop) = hex_y;

                            % Bogus the below line is only for debugging purposes. It shows how the tiled layer
                            % of regular hexagons will look like.
                            regular_hexagon(x_loop, y_loop) = pixel_value;

                            % Create a matrix variable to indicate whether the pixel value is taken or not by the
                            % first template.
                            is_pixel_taken(x_loop, y_loop) = 1;

                        end;
                    end;

                end;    % End for y_loop
            end;        % End for x_loop

        end;     % End for hex_x

    end;    % End for hex_y


    %=============================================================================================
    % The below logic is an exact replica of the above code, but considers only even layers. The
    % above code did the processing for odd layers. Note that this program works on the principle
    % of two templates. The odd layer will have a different template compared to the even layers.
    %=============================================================================================
    for hex_x = 2:2:no_of_hexagons_in_x_direction              % This will get all the even layers.
        for hex_y = 1:no_of_hexagons_in_y_direction

            %-----------------------------------------------------------------------------------------
            % Calculate the co-ordinates of the top_left and bottom-right corners of the rectangle that
            % exactly covers the entire hexagon. The calculation of these co-ordinates should depend
            % only on length_of_hexagon_pixel and the hexagon number(hex_x, hex_y).
            %-----------------------------------------------------------------------------------------
            if (hex_x == 2) & (hex_y == 1)      % Indicates the first hexagon in the even layer
                top_left_x = round(A_x1);
                top_left_y = round(E_y1) + 0;
                bottom_right_x = top_left_x + round(2*L2);
                bottom_right_y = round(top_left_y + (2*L1) + length_of_hexagon_pixel-1);  %GT

            elseif (hex_y == 1)                 % Indicates the start of a new layer of hexagons
                    top_left_x = top_left_x + vertical_no_of_pixels;
                    top_left_y = round(E_y1) + 0;

            else % Indicates that the hexagon is in the same layer as the previous one that was considered.
                top_left_x = top_left_x + increment_top_left_x;
                top_left_y = top_left_y + increment_top_left_y;
```

```
            end;

            bottom_right_x = top_left_x + round(2*L2);
            bottom_right_y = round(top_left_y + (2*L1) + length_of_hexagon_pixel-0);  %GT


            %-------------------------------------------------------------------------------------------
            % To draw and identify all hexagons, each hexagon is given a different pixel value.
            % All odd layer (see documentation) hexagons are given a pixel value of 75 and 225 while all
            % even layer hexagons are given a pixel value of 150 and 300 respectively.
            %-------------------------------------------------------------------------------------------
            if (rem(hex_x, 4) == 0)     % Indicates every second even layer (eg L4, L8, L12 ...)
                pixel_value = 300;
            elseif (rem(hex_x, 4) == 1) % Indicates every second odd layer (eg L3, L7, L11 ...)
                pixel_value = 225;
            elseif (rem(hex_x, 4) == 2) % Indicates every first even layer (eg L2, L6, L10 ...)
                pixel_value = 150;
            else                        % Indicates every first odd layer (eg L1, L5, L9 ...)
                pixel_value = 75;
            end;


            %-------------------------------------------------------------------------------------------
            % After calculating the top_left and bottom_right corner within which each hexagon will fall,
            % loop only across this rectangle of pixels to determine which pixels fall inside and which
            % pixels fall outside the hexagon.
            %-------------------------------------------------------------------------------------------
            for x_loop = top_left_x: bottom_right_x
                for y_loop = top_left_y: bottom_right_y
                    if (x_loop <= no_of_pixels_in_x_direction & y_loop <= no_of_pixels_in_y_direction)

                        %-------------------------------------------------------------------------------------------
                        % Check whether the pixel is already taken by the hexagon template above. If it has then
                        % ignore the pixel, else put it in this template.
                        %-------------------------------------------------------------------------------------------
                        if (is_pixel_taken(x_loop, y_loop) == 0)
                            hexagon_img(hex_x, hex_y) = hexagon_img(hex_x, hex_y) + original_image(x_loop, y_loop);
                            no_of_pixels_in_hexagon(hex_x, hex_y) = no_of_pixels_in_hexagon(hex_x, hex_y) + 1;
                            regular_hexagon(x_loop, y_loop) = pixel_value;     % Bogus this line is not actually needed
but helps if we have to determine breaks or overlaps in the regular hexagon structure.

                            %-------------------------------------------------------------------------
                            % Store the hex_x value in another matrix variable to indicate that the
                            % pixel(x_loop, y_loop lies in the hexagon whose hex_x value is so-and-so.
                            % This way we wouldn't have to write the re-allocating algorithm.
                            %-------------------------------------------------------------------------
                            original_img_lies_in_x(x_loop, y_loop) = hex_x;
                            original_img_lies_in_y(x_loop, y_loop) = hex_y;

                            is_pixel_taken(x_loop, y_loop) = 1;
                        end;

                    end;

                end;    % End for y_loop
            end;        % End for x_loop

        end;     % End for hex_x

    end;    % End for hex_y



    % Calculate the averaged pixel values.
    for hex_x = 1:no_of_hexagons_in_x_direction
        for hex_y = 1:no_of_hexagons_in_y_direction
            if no_of_pixels_in_hexagon(hex_x, hex_y) == 0
                averaged_hexagon(hex_x, hex_y) = 0;
            else
                averaged_hexagon(hex_x, hex_y) = hexagon_img(hex_x, hex_y)/no_of_pixels_in_hexagon(hex_x, hex_y);
            end;
        end;
    end;

    for x_loop = 1:no_of_pixels_in_x_direction
        for y_loop = 1:no_of_pixels_in_y_direction
            if original_img_lies_in_x(x_loop, y_loop) == 0 | original_img_lies_in_y(x_loop, y_loop) == 0
                enlarged_hexagon_img(x_loop, y_loop) = original_image(x_loop, y_loop);
            else
                enlarged_hexagon_img(x_loop, y_loop) = averaged_hexagon(original_img_lies_in_x(x_loop, y_loop),
original_img_lies_in_y(x_loop, y_loop));
            end;
        end;    % End for y_loop
    end;        % End for x_loop

    hexagon_pixel_image = enlarged_hexagon_img;
```

hexagon_pixel_image_6neighbor.m

```
%================================================================================================
% Created By : Girish Tirunelveli
% Created On : June 07, 2002
% File Name  : hexagon_pixel_image_6neighbor.m
% Description: This program accepts the image which needs to be converted into hexagon
%              pixel image. The second paramter is the length of the hexagon pixel
%              desired. It creates the hexagon pixel image based on the six-neighbor
%              approach.
%================================================================================================
%function [hexagon_img] = hexagon_pixel_image_6neighbor (image_variable, length_of_hexagon_pixel)
    length_of_hexagon_pixel = 4.5;
    length_of_rectangular_template = ceil(length_of_hexagon_pixel*3);
    height_of_rectangular_template = ceil(length_of_hexagon_pixel * 4 * sin(pi/3));

    column_section_length = floor(length_of_rectangular_template * 1/6);
    row_section_length = floor (height_of_rectangular_template * 1/4);
    clear hexagon_img;
    for column_no = 1:length_of_rectangular_template
        for row_no = 1:height_of_rectangular_template
            if column_no <= floor(length_of_rectangular_template *1/6)
                    if row_no <= height_of_rectangular_template/2
                        hexagon_img (row_no, column_no) = 0;  % Black1
                      else
                        hexagon_img (row_no, column_no) = 200;  % Yellow1
                    end;
                elseif column_no <= floor(length_of_rectangular_template * 2/6)
                    if row_no <= 0*row_section_length + (column_no - column_section_length) * sin (pi/3)
                        hexagon_img (row_no, column_no) = 75;  % Red1
                      else
                        hexagon_img (row_no, column_no) = 0;  % Black1
                    end;
                    if row_no <= 1*row_section_length + (column_no - column_section_length) * cos (pi/3)
                        hexagon_img (row_no, column_no) = 0;  % Black1
                      else
                        hexagon_img (row_no, column_no) = 150;  % Green1
                    end;
                    if row_no <= 2*row_section_length + (column_no - column_section_length) * sin(pi/3)
                        hexagon_img (row_no, column_no) = 150;  % Green1
                      else
                        hexagon_img (row_no, column_no) = 200;  % Yellow1
                    end;
                    if row_no <= 3*row_section_length + (column_no - column_section_length) * cos (pi/3)
                        hexagon_img (row_no, column_no) = 200;  % Yellow1
                      else
                        hexagon_img (row_no, column_no) = 75;   % Red2
                    end;
                elseif column_no <= length_of_rectangular_template *3/6
                    if row_no <= height_of_rectangular_template/4
                        hexagon_img (row_no, column_no) = 75;  % Red1
                      elseif row_no <= height_of_rectangular_template/2
                        hexagon_img (row_no, column_no) = 150;  % Green1
                      elseif row_no <= height_of_rectangular_template*.75
                        hexagon_img (row_no, column_no) = 150;  % Green1
                      else
                        hexagon_img (row_no, column_no) = 75;  % Red2
                    end;
                elseif column_no <= length_of_rectangular_template * 4/6
                    if row_no <= height_of_rectangular_template/4
                        hexagon_img (row_no, column_no) = 75;  % Red1
                      elseif row_no <= height_of_rectangular_template/2
                        hexagon_img (row_no, column_no) = 150;  % Green1
                      elseif row_no <= height_of_rectangular_template*.75
                        hexagon_img (row_no, column_no) = 150;  % Green1
                      else
                        hexagon_img (row_no, column_no) = 75;  % Red2
                    end;


            end;
        end;
    end;
```

image_quality_using_ed.m

```
%================================================================================================
% Created By : Girish Tirunelveli
% Created On : June 19, 2001
% File Name  : image_quality_using_ed.m
% Description: This program calculates the image quality between two images using
%              the Euclidean Distance method.
%================================================================================================
function [image_quality]= image_quality_using_ed (image1, image2)

    %------------------------------------------------------------------------
    % Step1. Assign the output variables as 0, just in case the job abhends in
```

```
%        the middle. Also do all initializing required for this function.
%------------------------------------------------------------------------
image_quality = 0;
image1 = double(image1);
image2 = double (image2);
[no_of_pixels_in_x_direction, no_of_pixels_in_y_direction] = size(image1);

%------------------------------------------------------------------------
% Step2. Calculate the image quality-
%        Euclidean Distance Image Quality =
%                sqrt(sum((image1 - enlarged_hexagon_img)^2)).
%        For calculating the image_quality consider a circle that is only
%        1/3rd the size of the image to get rid of the distortions caused
%        in the edge while rotating.
%------------------------------------------------------------------------
radius   = 100; %floor (no_of_pixels_in_x_direction/3);
x_center = floor (no_of_pixels_in_x_direction/2);
y_center = floor (no_of_pixels_in_y_direction/2);
for x_loop = 1:no_of_pixels_in_x_direction
    for y_loop = 1:no_of_pixels_in_y_direction
        if (x_loop-x_center)^2 + (y_loop-y_center)^2 <= radius^2
            difference_img(x_loop, y_loop) = image1(x_loop, y_loop) - image2(x_loop, y_loop);
          else
            difference_img(x_loop, y_loop) = 0;
        end;
    end;
end;

difference_img = difference_img.^2;
image_quality = sqrt(sum(sum(difference_img)));
image_quality = image_quality/(255 * sqrt(no_of_pixels_in_x_direction*no_of_pixels_in_y_direction));
    % 255 is the maximum grey value in the image.
```

image_quality_using_fd.m

```
%=========================================================================================
% Created By : Girish Tirunelveli
% Created On : June 19, 2001
% File Name  : image_quality_using_fd.m
% Description: This program calculates the image quality between two images using
%              the fidelity measure
%=========================================================================================
function [image_quality]= image_quality_using_fd (image1, image2)

    %------------------------------------------------------------------------
    % Step1. Assign the output variables as 0, just in case the job abhends in
    %        the middle. Also do all initializing required for this function.
    %------------------------------------------------------------------------
    image_quality = 0;
    image1 = double(image1);
    image2 = double(image2);

    %------------------------------------------------------------------------
    % Step2. Calculate the image quality-
    %        Fidelity BZD = sum(sum(image1*image2)) / sum(sum(image1*image1)).
    %        Note that the multiplication above is matrix mulplication.
    %------------------------------------------------------------------------
    BZD_numerator   = sum(sum(image1.*image2));
    BZD_denominator = sum(sum(image1.*image1));

    image_quality = BZD_numerator/BZD_denominator;
```

image_quality_using_rb.m

```
%=========================================================================================
% Created By : Girish Tirunelveli
% Created On : June 19, 2001
% File Name  : image_quality_using_rb.m
% Description: This program calculates the image quality between two images using
%              the resemblity measure
%=========================================================================================
function [image_quality]= image_quality_using_rb (image1, image2)

    %------------------------------------------------------------------------
    % Step1. Assign the output variables as 0, just in case the job abhends in
    %        the middle. Also do all initializing required for this function.
    %------------------------------------------------------------------------
    image_quality = 0;
    image1 = double(image1);
    image2 = double(image2);
```

```
    %-------------------------------------------------------------------------
    % Step2. Calculate the image quality-
    %        XSD_numerator   = sum(sum(image1*image2))
    %        XSD_denominator = sqrt(sum(sum(image1*image1))) *
    %                          sqrt(sum(sum(image2*image2)))
    %        XSD = XSD_numerator/XSD_denominator;
    %        Note that the multiplication above is matrix mulplication.
    %-------------------------------------------------------------------------
    XSD_numerator   = sum(sum(image1.*image2));
    XSD_denominator = sqrt(sum(sum(image1.*image1))) * sqrt(sum(sum(image2.*image2)));

    image_quality = XSD_numerator/XSD_denominator;
```

image_quality_using_hd.m

```
%=========================================================================================================================
% Created By : Girish Tirunelveli
% Created On : January 12, 2003
% File Name  : image_quality_using_hd.m
% Description: This program calculates the image quality between two images using
%              the hausdorff distance
%=========================================================================================
function [image_quality]= image_quality_using_hd (image1, image2)

    %-------------------------------------------------------------------
    % Step1. Initialize all required variables. Also initialize the output
    %        parameter just in case the program abhends in the middle.
    %-------------------------------------------------------------------
    image1        = double(image1);
    image2        = double(image2);
    image_quality = 0;

    %-------------------------------------------------------------------
    % Step2. Resize the image to 16 by 16 for faster computation.
    %-------------------------------------------------------------------
    RI_image1 = imresize (image1, [16 16], 'bicubic');
    RI_image2 = imresize (image2, [16 16], 'bicubic');
    PC = zeros (16);

    %-------------------------------------------------------------------------
    % Step3. For each pixel value of image1, find the closest match in image2.
    %        Add all the differences.
    %-------------------------------------------------------------------------
    total_distance = 0;
    for rn1=1:16
        for cn1=1:16
            distance = 1000000;
            PC_x = [];
            PC_y = [];

            for rn2= 1:16
                for cn2 = 1:16
                    if PC(rn2,cn2) == 0   % Indicates not chosen
                        pixel_value_diff =  abs(RI_image1(rn1, cn1) - RI_image2(rn2, cn2));

                        if pixel_value_diff < distance
                            if (isempty(PC_x) == 0)    % Indicates chosen before
                                PC(PC_x, PC_y) = 0;    % Unmarked to not chosen
                            end;
                            PC(rn2, cn2) = 1;      % Marked as chosen
                            PC_x = rn2;
                            PC_y = cn2;
                            distance = pixel_value_diff;
                        end;
                    end;
                end;
            end;
            total_distance = total_distance + distance;
        end;
    end;
    image_quality = total_distance;
```

entropy.m

```
%=========================================================================================
% Created By : Girish Tirunelveli
% Created On : July 31, 2001
% File Name  : entropy.m
% Description: This program calculates the information content (entropy) present
%              in an image. It returns the histogram and entropy as output
%              parameters.
%=========================================================================================
```

```
function [histogram, entropy] = entropy(original_image)
    original_image = double (original_image);
    min_pixel_value=round(min(min(original_image)));
    max_pixel_value=round(max(max(original_image)));

    pixel_value = min_pixel_value:max_pixel_value;

    histogram(size (pixel_value)) = 0;

    [number_of_rows, number_of_columns] = size(original_image);
    for row_number = 1:number_of_rows
        for column_number = 1:number_of_columns
            current_pixel_value = round (original_image (row_number, column_number));
            histogram (current_pixel_value-min_pixel_value+1) = ...
                  histogram(current_pixel_value - min_pixel_value+1) + 1;
        end;
    end;

    %------------------------------------------------------------------------
    % Entropy  H = - Sum {PV(i) * log2[PV(i)]} where i runs for all grey
    % levels.
    %------------------------------------------------------------------------
    entropy = 0;
    for pixel_value_number = 1:max_pixel_value-min_pixel_value+1
        if histogram(pixel_value_number) ~= 0
            norm_hist = histogram (pixel_value_number) / ...
                              (number_of_rows * number_of_columns);
            entropy = entropy - norm_hist * log2 (norm_hist);
        end;
    end;
```

project1_mtf.m

```
%=============================================================================================================================
%  Created By : Girish Tirunelveli
%  Created On : June 8, 2001
%  File Name  : project1_mtf.m
%  Description: Draws the MTF Curve (Modulation Transfer Function Curve) of the square pixel image
%               and the hexagon pixel image.
%  Algorithm 1. Create sine_wave_image of frequency 1. This is SWI.
%            2. Create the square pixel image of the sine wave. Call it SPI.
%            3. Create the hexagon pixel image of the sine wave. Call it HPI.
%            4. Take the fourier transform of all images. fft_SWI, fft_SWI, fft_HPI.
%            5. Point of interest in the transform is (129, 129-frequency).
%            6. Ratio of this point between two images is the MTF.
%            7. Plotting MTF for a range of frequencies gives the MTF curve.
%=============================================================================================================================
clear
%--------------------------------
% Accept the input parameters.
%--------------------------------
length_of_square_pixel  = input ('Enter length of square pixel: ');
length_of_hexagon_pixel = input ('Enter length of hexagon pixel: ');
frequency=[0:2:128];

time_start=cputime;

%-----------------------------------------------------------------------------
% For each frequency, construct the sine_wave_image. Get the square pixel image
% and hexagon pixel image. Get the fourier transform of each of these images.
% Get the value at (129, 129-frequency) coordinate. Lets say this value is
% A0 for the original sine image
% A1 for the square pixel image and
% A2 for the hexagon pixel image.
% MTF at the frequncy = A1/A0 for square.
%                     = A2/A0 for hexagon.
% Do this for all frequencies and plot a curve.
%-----------------------------------------------------------------------------
for frequency_number=1:length(frequency)

    % Get the sine_wave_image at the specified frequency.
    SWI=sine_wave (frequency (frequency_number), 256);

    % Convert it to square_pixel_image and get SPI.
    SPI=square_pixel_image (SWI, length_of_square_pixel);

    % Convert it to hexagon_pixel_image and get HPI.
    HPI=hexagon_pixel_image_2template (SWI, length_of_hexagon_pixel);

    % Get 2D FFT of all the images.
    fft_SWI = real(fftshift(fft2 (SWI)));
    fft_SPI = real(fftshift(fft2 (SPI)));
    fft_HPI = real(fftshift(fft2 (HPI)));

    % Get the amplitude of the transform at the point of interest.
    amp_SWI = fft_SWI (129, 129-frequency(frequency_number));
    amp_SPI = fft_SPI (129, 129-frequency(frequency_number));
```

```
    amp_HPI = fft_HPI (129, 129-frequency(frequency_number));


    SIQ_mtf(frequency_number) = amp_SPI/amp_SWI;
    HIQ_mtf(frequency_number) = amp_HPI/amp_SWI;

end;  % for frequency_number

time_end=cputime;

disp(strcat('Process Completed. Time taken:  ', num2str(time_end-time_start), ' seconds'));

%-------------------------
% Plot the MTF curve.
%-------------------------
clf
plot(frequency, SIQ_mtf, 'r+-');
hold
plot(frequency, HIQ_mtf, 'bo-');
title  ('Resolution Comparison between square and hexagon pixel using MTF')
xlabel ('Frequency ')
ylabel ('Modulation Transfer Function (MTF)');
legend ('Square Pixel', 'Hexagon Pixel')
```

sine_wave.m

```
%================================================================================
% Created By : Girish Tirunelveli
% Created On : June 07, 2002
% File Name  : sine_wave.m
% Description: This program creates a sine wave at a particular frequency.
%================================================================================
function [sine_wave_img] = sine_wave (frequency, size_of_img)
    sine_wave_img = zeros(size_of_img);
    for column_no=1:size_of_img
        theta_in_degrees = column_no * 360 * frequency / size_of_img;
        theta_in_radians = theta_in_degrees * pi / 180;
        sine_wave_img(:, column_no) = 128 + sin (theta_in_radians) * 128;
    end;
```

A.2 Code for *ART experiment

Extract all the files in this section into a particular folder on your computer. Change your Matlab™ session's present working directory to this folder. Simply type project2_main in the Matlab™ command prompt. This will show a parameter screen. The parameter screen is shown in Figure A-1. Enter the parameters for your experiment and click Ok. Based on the parameters entered, the code will do its magic and display as output the original test image, the seed image used (applies to *ART), the reconstructed image and the plot of the normalized Euclidean distance with respect to cycles. Note that test images are needed for using the code. The different test images that I used (all are 32 by 32 images) are tabulated below.

Sr#	Test Image File Name	Description	How the image was generated?
1.	admin32.bmp	University of Manitoba Administration Building	Reproduced with permission from Prof.W. Lehn's Digial Image Processing class.
2.	rose32.bmp	A low contrast image of a rose	Produced using my digital camera.
3.	phantom32.bmp	Phantom Image	Produced using the phantom function in Matlab™
4.	abcd32.bmp	Image containing the alphabets "abcd"	Created manually in Microsoft® Paintbrush.
5.	paint32.bmp	Image containing pattern from Microsoft Paintbrush	Created manually in Microsoft® Paintbrush.
6.	checker32.bmp	Image containing 8 by 8 blocks of sequential pixel values	Created manually in Microsoft® Paintbrush.
7.	circles32.bmp	Circles of uniform radius and different contrast thrown randomly on a uniform background.	Created manually in Microsoft® Paintbrush.
8.	sine32.bmp	Horizontal sine wave of frequency 1 and 10 is added with vertical sine wave of frequency 1 and 10.	Created mathematically in Matlab™.
9.	rand32.bmp	Uniformly distributed random image.	Created using Matlab's rand function

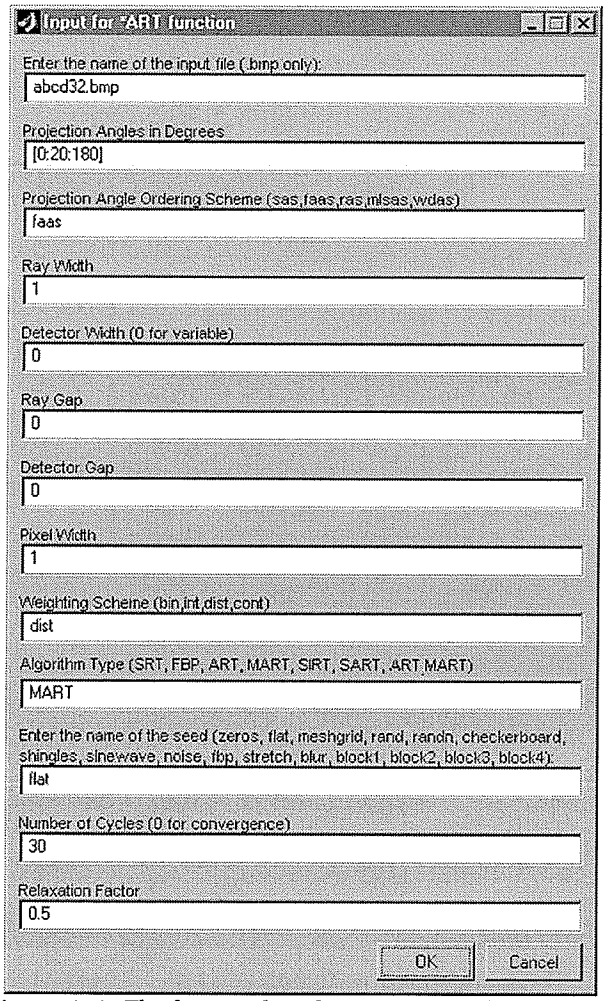


Figure A-1. *The front-end application screen for entering *ART parameters*

project2_main.m

```
%=============================================================================================
% Created By : Girish Tirunelveli
% Created On : December 31, 2002
% File Name  : project2_main.m
% Description: This program asks for all the input parameters required for studying the
%              confidence of *ART algorithms. Note that this program is written only
%              for square images.
%=============================================================================================

pack;    % Helps in storing the weight matrix variable especially if the image size is too large.

%---------------------------------------------------------------------
% Step1. Display the GUI input screen. Give appropriate defaults.
%---------------------------------------------------------------------
clc
input_options.Resize      ='off';
input_options.WindowStyle ='normal';
input_options.Interpreter ='tex';
input_window_title        ='Input for *ART function';
input_prompts             = {'Enter the name of the input file (.bmp only):', ...
                             'Projection Angles in Degrees', ...
                             'Projection Angle Ordering Scheme (sas,faas,ras,mlsas,wdas)', ...
                             'Ray Width', ...
                             'Detector Width (0 for variable)', ...
```

```
                            'Ray Gap', ...
                            'Detector Gap', ...
                            'Number of Detectors (Enter 0 if you need to cover the entire image)', ...
                            'Pixel Width', ...
                            'Weighting Scheme (bin,int,dist,cont)', ...
                            'Algorithm Type (SRT, FBP, ART, MART, SIRT, SART, ART_-MART)', ...
                            'Enter the name of the seed (zeros, flat, meshgrid, rand, randn, checkerboard, shingles,
sinewave, noise, fbp, stretch, blur, block1, block2, block3, block4):', ...
                            'Number of Cycles (0 for convergence)', ...
                            'Relaxation Factor', ...
                            'Enter Weight Matrix Variable (Enter blank if recalculation required)', ...
                            'Enter the Projection Values Variable (Enter blank if recalculation required)' ...
                           };
input_defaults            = {'abcd32.bmp', ...
                            '[0:20:180]', ...
                            'faas', ...
                            '1', ...
                            '0', ...
                            '0', ...
                            '0', ...
                            '0', ...
                            '1', ...
                            'dist', ...
                            'MART', ...
                            'flat', ...
                            '30', ...
                            '1', ...
                            'WM', ...
                            '' ...
                           };

% For proper display in desktop
input_lines               = [0.9, 0.8, 0.9, 0.1, 1.0, 0.1, 0.1, 0.1, 0.1, 0.8, 0.8, 0.9, 0.9, 0.8, 0.6, 0.6];
% For proper display in laptop
%input_lines              = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1];
answer=inputdlg(input_prompts,input_window_title,input_lines,input_defaults,input_options);
answer

%---------------------------------------------------------------------
% Step2. Convert the user response into proper datatypes.
%---------------------------------------------------------------------
input_number=0;

%=============================================
% INPUT1:     Original Image
%=============================================
input_number=input_number+1;
clear original_image
original_image_txt  = cell2struct (answer(input_number), 'value', 2);

%-------------------------------------------------------------------------
% If the file name has a equal to sign in it, then it indicates that
% the user wants to execute a function and store the function in the
% original_image variable. If the original file name has a period in
% it then the user wishes to open a image file, the characters flwing
% the image name are treated as the file format. If the file does not
% have either a equal to sign or a period in it then it is a .mat
% variable.
%-------------------------------------------------------------------------
equal_sign_found = find(original_image_txt.value == '=');
period_found     = find(original_image_txt.value == '.');
if (isempty(equal_sign_found) == 0)
        eval (sprintf('original_image %s;', original_image_txt.value));
    elseif (isempty(period_found) == 0)
        original_image      = imread (original_image_txt.value, 'bmp');
        original_image      = double(original_image);
        original_image      = original_image+1;
    else
        eval (sprintf('load %s;', original_image_txt.value));
        original_image(3,3)=8;
end;
[number_of_rows_in_image, number_of_columns_in_image] = size (original_image);

%=================================================
% INPUT2: Projection Angle (Theta) in Degrees
%         If the seed image is fbp then calculate the seed image
%         based on Filtered Back-Projection.
%=================================================
input_number=input_number+1;
clear theta_in_degrees
theta_in_degrees_txt = cell2struct (answer(input_number), 'value', 2);
theta_in_degrees     = str2num(theta_in_degrees_txt.value);


%=============================================================================
% INPUT3: Projection Angles Ordering Scheme (1,2,3,4,5,6)
%          1 indicates Sequential (angles are taken in order the data is
%            gathered) (SAS)
%          2 indicates Fixed Angle Access Scheme (FAAS)
%                           (angles are 90 degrees apart in this case)
%          3 indicates PNDAS (Prime Number Decomposition)
%          4 indicates Random Access Scheme (RAS)
%          5 indicates Multilevel Access Sceme (MLSAS)
```

```
%          6 indicates Weighted Distance Scheme (WDAS)
%================================================================================
input_number=input_number+1;
projection_angles_ordering_txt = cell2struct (answer(input_number), 'value', 2);
switch lower(projection_angles_ordering_txt.value)
    case {'sas'}
        theta_in_degrees = generate_paos_sas  (theta_in_degrees);
    case {'faas'}
        theta_in_degrees = generate_paos_faas (theta_in_degrees);
    case {'pndas'}
        theta_in_degrees = generate_paos_pndas(theta_in_degrees);
    case {'ras'}
        theta_in_degrees = generate_paos_ras  (theta_in_degrees);
    case {'mlsas'}
        theta_in_degrees = generate_paos_mlsas(theta_in_degrees);
    case {'wdas'}
        theta_in_degrees = generate_paos_wdas (theta_in_degrees);
    otherwise
        disp('Invalid Value for Projection Angles Ordering Scheme.')
        return;
end;


%================================================
% INPUT4:       Ray Width
%================================================
input_number=input_number+1;
clear ray_width
ray_width_txt = cell2struct (answer(input_number), 'value', 2);
ray_width     = str2num (ray_width_txt.value);

%================================================
% INPUT5:     Detector Width
%================================================
input_number=input_number+1;
clear detector_width
detector_width_txt = cell2struct (answer(input_number), 'value', 2);
detector_width     = str2num (detector_width_txt.value);

%================================================
% INPUT6:       Ray Gap
%================================================
input_number=input_number+1;
clear ray_gap
ray_gap_txt = cell2struct (answer(input_number), 'value', 2);
ray_gap     = str2num (ray_gap_txt.value);

%================================================
% INPUT7:     Detector Gap
%================================================
input_number=input_number+1;
clear detector_gap
detector_gap_txt = cell2struct (answer(input_number), 'value', 2);
detector_gap     = str2num (detector_gap_txt.value);

%================================================
% INPUT8:     Number of Detectors
%================================================
input_number=input_number+1;
clear number_of_detectors
number_of_detectors_txt = cell2struct (answer(input_number), 'value', 2);
number_of_detectors     = str2num (number_of_detectors_txt.value);
if (number_of_detectors == 0)
    number_of_detectors = number_of_rows_in_image + number_of_columns_in_image-1;
end;

%================================================
% INPUT9:     Pixel Width
%================================================
input_number=input_number+1;
clear pixel_width
pixel_width_txt = cell2struct (answer(input_number), 'value', 2);
pixel_width     = str2num (pixel_width_txt.value);

%================================================================================
% INPUT10: Weighting Scheme
%          1 indicates Binary scheme (bin)
%          2 indicates Length of ray withing pixel scheme (int)
%          3 indicates Distance of Ray-Center from center of pixel scheme (dist)
%          4 indicates Distance of center of pixel from farthest edge of
%                      adjacent ray scheme (cont).
%================================================================================
input_number=input_number+1;
clear weighting_scheme_txt
weighting_scheme_txt = cell2struct(answer(input_number), 'value', 2);


%================================================
% INPUT11:   Algorithm Type
%================================================
input_number=input_number+1;
clear algorithm_type
algorithm_type_txt = cell2struct (answer(input_number), 'value', 2);
```

```
algorithm_type     = algorithm_type_txt.value;

%============================================
% INPUT12:       Seed Image
%============================================
input_number=input_number+1;
clear seed_image
seed_image_txt      = cell2struct (answer(input_number), 'value', 2);
switch lower(seed_image_txt.value)
    case {'zeros'}
            seed_image       = zeros(size(original_image));
            seed_image_title = 'Seed Image(zeros)';
    case 'flat'
            seed_image       = ones(size(original_image));
            seed_image_title = 'Seed Image(flat)';
    case 'meshgrid'
            seed_image       = meshgrid(1: length(original_image));
            seed_image_title = 'Seed Image(meshgrid)';
    case 'rand'
            seed_image       = rand(size(original_image));
            seed_image_title = 'Seed Image(rand)';
    case 'randn'
            seed_image       = abs(128+randn(size(original_image))*32);
            seed_image_title = 'Seed Image(randn)';
    case 'checkerboard'
            seed_image       = checkerboard(size(original_image, 1), 1, 255);
            seed_image_title = 'Seed Image(checkerboard)';
    case 'shingles'
            seed_image       = shingles(size(original_image));
            seed_image_title = 'Seed Image(shingles)';
    case 'sinewave'
            seed_image       = sinewave(length(original_image));
            seed_image_title = 'Seed Image(sinewave)';
    case 'noise'
            Poriginal_image  = sqrt(sum(sum(original_image.^2)));
            seed_image       = rand(size(original_image)) * Poriginal_image/50 + original_image;
            seed_image_title = 'Seed Image(noise)';
    case 'fbp'
            pv_radon         = radon(original_image,theta_in_degrees);
            seed_image       = iradon(pv_radon,theta_in_degrees,'nearest','Hann', 1, number_of_rows_in_image);
            seed_image_title = 'Seed Image(fbp)';
    case 'stretch'
            seed_image       = abs(stretch(original_image, 1.5));
            seed_image_title = 'Seed Image(stretch)';
    case 'blur'
            seed_image       = blur(original_image, 1);
            seed_image_title = 'Seed Image(blur)';
    case 'block1'
            seed_image       = block (ones(size(original_image)), 1);
            seed_image_title = 'Seed Image(block1)';
    case 'block2'
            seed_image       = block (ones(size(original_image)), 2);
            seed_image_title = 'Seed Image(block2)';
    case 'block3'
            seed_image       = block (ones(size(original_image)), 3);
            seed_image_title = 'Seed Image(block3)';
    case 'block4'
            seed_image       = block (ones(size(original_image)), 4);
            seed_image_title = 'Seed Image(block4)';
    otherwise
            disp('Invalid Seed Image.')
            seed_image_title = 'Seed Image';
            return;
end;

%============================================
% INPUT13:   Number of Cycles
%============================================
input_number=input_number+1;
clear number_of_cycles
number_of_cycles_txt = cell2struct (answer(input_number), 'value', 2);
number_of_cycles     = str2num(number_of_cycles_txt.value);

%=============================================================================
% INPUT14: Relaxation Factor
%=============================================================================
input_number=input_number+1;
clear relaxation_factor;
relaxation_factor_txt = cell2struct (answer(input_number), 'value', 2);
relaxation_factor     = str2num(relaxation_factor_txt.value);


%=============================================================================
% INPUT15: Weight Matrix
% If the weight matrix is entered then assign the weight matrix to the entered
% variable. else recalculate the weight matrix. This is done for speeding up
% testing. Note that the weight matrix is a 4-D matrix as below -
%       WM (x of pixel, y of pixel, angle_number, detector number)
% The weight matrix is computed by four different methods -
%     1. Binary Scheme (bin)
%     2. Length of the ray within the pixel Scheme (int).
%     3. Distance of the center of the pixel from the center of the ray Scheme (dist).
%     4. Contribution made by pixel to the adjacent rays Scheme (cont).
```

```
%=========================================================================================
input_number=input_number+1;
weight_matrix_txt = cell2struct (answer(input_number), 'value', 2);
if (isempty(weight_matrix_txt.value))
        time_start=cputime;
        switch lower(weighting_scheme_txt.value)
            case {'bin'}
                [WM]=generate_wm_bin (original_image, theta_in_degrees, number_of_detectors, detector_width,
pixel_width);
            case {'int'}
                [WM]=generate_wm_int (original_image, theta_in_degrees, number_of_detectors, detector_width,
pixel_width);
            case {'dist'}
                [WM]=generate_wm_dist(original_image, theta_in_degrees, number_of_detectors, detector_width,
pixel_width);
            case {'cont'}
                [WM]=generate_wm_cont(original_image, theta_in_degrees, number_of_detectors, detector_width,
pixel_width);
            otherwise
                disp('Invalid Value for Weighting Scheme');
                return;
        end;
        time_end=cputime;
        disp(strcat('Weight Matrix Computed. Time taken:  ', num2str(time_end-time_start), ' seconds'));
    else
        eval (strcat('WM =', weight_matrix_txt.value, ';'));
end;

%=========================================================================================
% INPUT16: Projection Values (Forward Simulation)
% If the variable is entered then assign the PV variable to the entered variable,
% else recalculate PV by calling generate_projections function. This is done for
% speeding up testing. Note that the PV is a 2-D matrix as below -
%            PV (theta_in_degrees, detector_number)
%=========================================================================================
input_number=input_number+1;
projection_values_txt = cell2struct (answer(input_number), 'value', 2);
if (isempty(projection_values_txt.value))
        time_start=cputime;
        [PV]=generate_projections(original_image, theta_in_degrees, WM, number_of_detectors);
        time_end=cputime;
        disp(strcat('Forward Simulation Complete. Time taken:  ', num2str(time_end-time_start), ' seconds'));
    else
        eval (strcat('PV =', projection_values_txt.value, ';'));
end;


%-------------------------------------------------------------------------
% Step3. Validate the input. This input validates include both the
%        program limitations and also the algorithm limitations.
%-------------------------------------------------------------------------
if size(original_image) ~= size(original_image')
    disp('Input Error');
    disp('Original Image must be square.');
    return;
end;
if size(seed_image) ~= size(original_image)
    disp('Input Error');
    disp('Seed Image must be the same size as that of the original image.');
    return;
end;
%
%if ~isempty(find(original_image==0))
%    disp('Input Error');
%    disp('Original Image must not have any elements with 0 value.');
%    return;
%end;
%if ~isempty(find(seed_image==0))
%    disp('Input Error');
%    disp('Seed Image must not have any elements with 0 value.');
%    return;
%end;

%-------------------------------------------------------------------------
% Step4. Do the appropriate processing
%-------------------------------------------------------------------------
switch upper(algorithm_type)
    case {'FBP'}
            time_start=cputime;
            precision_multiplier = 10000000;
            [RI_fbp,ID_fbp]=ifbp(seed_image, original_image, ...
                                 PV, theta_in_degrees, ...
                                 number_of_cycles, number_of_detectors, ...
                                 WM, precision_multiplier, relaxation_factor);
            time_end=cputime;
            disp(strcat('Backward Simulation (FBP) Complete. Time taken:  ', num2str(time_end-time_start), '
seconds'));
    case {'SRT'}
            time_start=cputime;
            precision_multiplier = 10000000;
            [RI_srt,ID_srt]=isrt(seed_image, original_image, ...
                                 PV, theta_in_degrees, ...
                                 number_of_cycles, number_of_detectors, ...
```

```
                                WM, precision_multiplier, relaxation_factor);
            time_end=cputime;
            disp(strcat('Backward Simulation (SRT) Complete. Time taken:  ', num2str(time_end-time_start), '
seconds'));
    case {'ART'}
            time_start=cputime;
            precision_multiplier = 10000000;
            [RI_art,ID_art]=iart(seed_image, original_image, ...
                                PV, theta_in_degrees, ...
                                number_of_cycles, number_of_detectors, ...
                                WM, precision_multiplier, relaxation_factor);
            time_end=cputime;
            disp(strcat('Backward Simulation (ART) Complete. Time taken:  ', num2str(time_end-time_start), '
seconds'));
    case {'MART'}
            time_start=cputime;
            precision_multiplier = 10000000;
            [RI_mart,ID_mart]=imart(seed_image, original_image, ...
                                    PV, theta_in_degrees, ...
                                    number_of_cycles, number_of_detectors, ...
                                    WM, precision_multiplier, relaxation_factor);
            time_end=cputime;
            disp(strcat('Backward Simulation (MART) Complete. Time taken:  ', num2str(time_end-time_start), '
seconds'));
    case 'SIRT'
            time_start=cputime;
            precision_multiplier = 10000000;
            [RI_sirt,ID_sirt]=isirt(seed_image, original_image, ...
                                    PV, theta_in_degrees, ...
                                    number_of_cycles, number_of_detectors, ...
                                    WM, precision_multiplier, relaxation_factor);
            time_end=cputime;
            disp(strcat('Backward Simulation (SIRT) Complete. Time taken:  ', num2str(time_end-time_start), '
seconds'));
    case {'SART'}
            time_start=cputime;
            precision_multiplier = 10000000;
            [RI_sart,ID_sart]=isart(seed_image, original_image, ...
                                    PV, theta_in_degrees, ...
                                    number_of_cycles, number_of_detectors, ...
                                    WM, precision_multiplier, relaxation_factor);
            time_end=cputime;
            disp(strcat('Backward Simulation (SART) Complete. Time taken:  ', num2str(time_end-time_start), '
seconds'));
    case {'ART_MART'}
            time_start=cputime;
            precision_multiplier = 10000000;
            [RI_art_mart,ID_art_mart]=iart_mart(seed_image, original_image, ...
                                            PV, theta_in_degrees, ...
                                            number_of_cycles, number_of_detectors, ...
                                            WM, precision_multiplier, relaxation_factor);
            time_end=cputime;
            disp(strcat('Backward Simulation (ART_MART) Complete. Time taken:  ', num2str(time_end-time_start), '
seconds'));
    otherwise
            disp('Invalid Algorithm Type.')
            return;
end;

%-----------------------------------------------------------------
% Step5. Clear the figure window and show results.
%-----------------------------------------------------------------
current_figure=figure(15);
clf
set (current_figure, 'NumberTitle', 'off');
set (current_figure, 'Name',        'Project2 Results');

subplot (2, 2, 1), imagesc (original_image);
colormap (gray);
title 'Original Image'
axis square
axis off

subplot (2, 2, 2), imagesc (seed_image);
colormap (gray);
title (seed_image_title)
axis square
axis off

cycle=[0:number_of_cycles];
switch upper(algorithm_type)
    case {'FBP'}
        subplot (2, 2, 3), imagesc (RI_fbp);
        colormap (gray);
        title 'Reconstructed Image (FBP)'
        axis square
        axis off

        ID_fbp(1:number_of_cycles+1)=ID_fbp;
        subplot (2,2,4), plot (cycle, ID_fbp, 'r-', cycle, ID_fbp, 'bo')
        grid on;
        xlabel ('Cycle');
        ylabel ('Normalized Eucl. Distance');
```

```
    title ('Plot of Normalized Eucl.Distance Vs Cycle');
    axis ([0 number_of_cycles 0 1.1*max(ID_fbp)])
    ID_fbp'

case {'SRT'}
    subplot (2, 2, 3), imagesc (RI_srt);
    colormap (gray);
    title 'Reconstructed Image (SRT)'
    axis square
    axis off

    ID_srt(1:number_of_cycles+1)=ID_srt;
    subplot (2,2,4), plot (cycle, ID_srt, 'r-', cycle, ID_srt, 'bo')
    grid on;
    xlabel ('Cycle');
    ylabel ('Normalized Eucl. Distance');
    title ('Plot of Normalized Eucl.Distance Vs Cycle');
    axis ([0 number_of_cycles 0 1.1*max(ID_srt)])
    ID_srt'

case {'ART'}
    subplot (2, 2, 3), imagesc (RI_art);
    colormap (gray);
    title 'Reconstructed Image (ART)'
    axis square
    axis off

    ID_art(1:number_of_cycles+1)=ID_art;
    subplot (2,2,4), plot (cycle, ID_art, 'r-', cycle, ID_art, 'bo')
    grid on;
    xlabel ('Cycle');
    ylabel ('Normalized Eucl. Distance');
    title ('Plot of Normalized Eucl.Distance Vs Cycle');
    axis ([0 number_of_cycles 0 1.1*max(ID_art)])
    ID_art'


case {'MART'}
    subplot (2, 2, 3), imagesc (RI_mart);
    colormap (gray);
    title 'Reconstructed Image (MART)'
    axis square
    axis off

    ID_mart(1:number_of_cycles+1)=ID_mart;
    subplot (2,2,4), plot (cycle, ID_mart, 'r-', cycle, ID_mart, 'bo')
    grid on;
    xlabel ('Cycle');
    ylabel ('Normalized Eucl. Distance');
    title ('Plot of Normalized Eucl.Distance Vs Cycle');
    axis ([0 number_of_cycles 0 1.1*max(ID_mart)])
    ID_mart'

case {'SIRT'}
    subplot (2, 2, 3), imagesc (RI_sirt);
    colormap (gray);
    title 'Reconstructed Image (SIRT)'
    axis square
    axis off

    ID_sirt(1:number_of_cycles+1)=ID_sirt;
    subplot (2,2,4), plot (cycle, ID_sirt, 'r-', cycle, ID_sirt, 'bo')
    grid on;
    xlabel ('Cycle');
    ylabel ('Normalized Eucl. Distance');
    title ('Plot of Normalized Eucl.Distance Vs Cycle');
    axis ([0 number_of_cycles 0 1.1*max(ID_sirt)])
    ID_sirt'

case {'SART'}
    subplot (2, 2, 3), imagesc (RI_sart);
    colormap (gray);
    title 'Reconstructed Image (SART)'
    axis square
    axis off

    ID_sart(1:number_of_cycles+1)=ID_sart;
    subplot (2,2,4), plot (cycle, ID_sart, 'r-', cycle, ID_sart, 'bo')
    grid on;
    xlabel ('Cycle');
    ylabel ('Normalized Eucl. Distance');
    title ('Plot of Normalized Eucl.Distance Vs Cycle');
    axis ([0 number_of_cycles 0 1.1*max(ID_sart)])
    ID_sart'

case {'ART_MART'}
    subplot (2, 2, 3), imagesc (RI_art_mart);
    colormap (gray);
    title 'Reconstructed Image (ART_-MART)'
    axis square
    axis off

    ID_art_mart(1:number_of_cycles+1)=ID_art_mart;
```

```
        subplot (2,2,4), plot (cycle, ID_art_mart, 'r-', cycle, ID_art_mart, 'bo')
        grid on;
        xlabel ('Cycle');
        ylabel ('Normalized Eucl. Distance');
        title ('Plot of Normalized Eucl.Distance Vs Cycle');
        axis ([0 number_of_cycles 0 1.1*max(ID_art_mart)])
        ID_art_mart'

    otherwise
            disp('Invalid Algorithm Type.')
            return;
end;

load handel
x=y(1:1000);
sound (x, 8192);
```

generate_paos_sas.m

```
%=============================================================================================
% Created By : Girish Tirunelveli
% Created On : January 01, 2003
% File Name  : generate_paos_sas.m
% Description: The generate_paos_sas function generates the sequential projection angles
%              ordering. It returns the projection angles in the same order as they
%              were received. Technically this program is not required. It is created
%              only to be consistent with the other projection angles ordering schemes.
%=============================================================================================
function [R_angles] = generate_paos_sas(theta_in_degrees)
    R_angles          = theta_in_degrees;
```

generate_paos_faas.m

```
%=============================================================================================
% Created By : Girish Tirunelveli
% Created On : January 01, 2003
% File Name  : generate_paos_faas.m
% Description: The generate_paos_faas function generates the projection angles ordering
%              scheme such that the projection angles are as orthogonal as possible.
%              For example a sequence of angles 0, 30, 60, 90, 120, 150, 180  is
%              changed to 0, 90, 30, 120, 60, 150, 180.
%=============================================================================================
function [R_angles] = generate_paos_faas(theta_in_degrees)
    %-------------------------------------------------------------------------------
    % Step1. Initialize all the appropriate variables. Also initialize the output
    %        parameter, just in case the program abhends in the middle.
    %-------------------------------------------------------------------------------
    number_of_angles  = length(theta_in_degrees);
    R_angles          = zeros(size(theta_in_degrees));
    angle_selected    = zeros(size(theta_in_degrees));

    %-------------------------------------------------------------------------------
    % Step2. The first orthogonal angle is the first one from the theta series.
    %-------------------------------------------------------------------------------
    R_angles (1)          = theta_in_degrees (1);
    angle_selected (1)    = 1;

    %-------------------------------------------------------------------------------
    % Step2. We need to assign the orthogonal angles from the 2nd element until the
    %        last.
    %-------------------------------------------------------------------------------
    for oan = 2:number_of_angles
        if mod(oan, 2) == 1
            ref_angle_value = R_angles(oan-1)-90;
          else
            ref_angle_value = R_angles(oan-1)+90;
        end;
        diff            =1000;
        previous_an     = [];

        for an=2:number_of_angles
            if (angle_selected(an) == 0)    % Indicates the angle has not been selected before.
                new_diff= abs(ref_angle_value - theta_in_degrees(an));
                if new_diff < diff
                    % Reset previously set angle
                    if (isempty (previous_an) == 0)
                        angle_selected(previous_an) = 0;
                    end;

                    % Set the current angle.
                    previous_an         = an;
                    diff                = new_diff;
                    R_angles(oan)       = theta_in_degrees(an);
```

```
                angle_selected(an)  = 1;

            end;
        end;
    end;   % for an

end;  % for oan
```

generate_paos_ras.m

```
%=============================================================================================
% Created By : Girish Tirunelveli
% Created On : January 01, 2003
% File Name  : generate_paos_ras.m
% Description: Random Access Scheme for ordering the projection angles.
%              Algorithm: -
%              1.
%=============================================================================================
function [R_angles] = generate_paos_ras(theta_in_degrees)
    %-------------------------------------------------------------------------------
    % Step1. Initialize all the appropriate variables. Also initialize the output
    %        parameter, just in case the program abhends in the middle.
    %-------------------------------------------------------------------------------
    number_of_angles  = length(theta_in_degrees);
    R_angles          = zeros(size(theta_in_degrees));
    angle_selected    = zeros(size(theta_in_degrees));
    next_angle_index  = 0;

    %-------------------------------------------------------------------------------
    % Step2. We need to assign the random angles from the 1st element until the
    %        last.
    %        Algorithm: -
    %          a) Select an angle in random.
    %          b) Make sure that the angle_index has not been selected before.
    %-------------------------------------------------------------------------------
    for ran = 1:number_of_angles
        angle_index = round(rand*(number_of_angles-1)) + 1;
        girish      = find (angle_selected == angle_index);

        while isempty(girish) == 0
            angle_index = round(rand*(number_of_angles-1)) + 1;
            girish      = find (angle_selected == angle_index);
        end;

        %-------------------------------------------------------------------------
        % If the program control comes to this point then it indicates that
        % the angle index has not been selected before. Assign the next_angle
        % of the output variable as the angle from the input set with the
        % same index. Also mark the index as selected so that it will not be
        % selected again.
        %-------------------------------------------------------------------------
        next_angle_index           = next_angle_index+1;
        R_angles(next_angle_index) = theta_in_degrees(angle_index);
        angle_selected(next_angle_index) = angle_index;

    end;  % for ran
```

generate_paos_mlsas.m

```
%=============================================================================================
% Created By : Girish Tirunelveli
% Created On : January 01, 2003
% File Name  : generate_paos_mlsas.m
% Description: The generate_paos_mlsas function generates the projection angles ordering
%              scheme based on MLS (Multi Level Acess Scheme) based on Guan And Gordon.
%=============================================================================================
function [R_angles] = generate_paos_mlsas(theta_in_degrees)
    %-------------------------------------------------------------------------------
    % Step1. Initialize all the appropriate variables. Also initialize the output
    %        parameter, just in case the program abhends in the middle.
    %-------------------------------------------------------------------------------
    number_of_angles  = length(theta_in_degrees);
    R_angles          = zeros(size(theta_in_degrees));
    angle_selected    = zeros(size(theta_in_degrees));

    %-------------------------------------------------------------------------------
    % Step2. The first orthogonal angle is the first one from the theta series.
    %-------------------------------------------------------------------------------
    R_angles (1)          = theta_in_degrees (1);
    angle_selected (1)    = 1;
```

```
    %-------------------------------------------------------------------------------
    % Step2. For the second angle, the difference from the first angle must be
    %        close to 90.
    %-------------------------------------------------------------------------------
    for oan = 2:number_of_angles
        if (oan==2)
            preferred_distance = 90;
            ref_angle_value = 90;
          elseif mod(oan, 2) == 1      % Odd
            preferred_distance = (R_angles(oan-1)+ R_angles(oan-2))/2;
            ref_angle_value = R_angles(oan-1)-preferred_distance;
          else
            preferred_distance = 90;
            ref_angle_value = R_angles(oan-1)+preferred_distance;
        end;
        diff            =1000;
        previous_an     = [];

        for an=2:number_of_angles
            if (angle_selected(an) == 0)    % Indicates the angle has not been selected before.
                new_diff= abs(ref_angle_value - theta_in_degrees(an));
                if new_diff < diff
                    % Reset previously set angle
                    if (isempty (previous_an) == 0)
                        angle_selected(previous_an) = 0;
                    end;

                    % Set the current angle.
                    previous_an         = an;
                    diff                = new_diff;
                    R_angles(oan)       = theta_in_degrees(an);
                    angle_selected(an)  = 1;

                end;
            end;
        end;   % for an

    end;  % for oan
```

generate_paos_wdas.m

```
%================================================================================================
% Created By : Girish Tirunelveli
% Created On : January 01, 2003
% File Name  : generate_paos_wdas.m
% Description: The generate_paos_wdas function generates the projection angles ordering
%              scheme as proposed by Mueller. The code is written by Chris Badea and
%              is reproduced here with his permission. The code is changed slightly
%              to fit the standards of my coding and style.
%================================================================================================
function [R_angles] = generate_paos_wdas(theta_in_degrees)
    %-------------------------------------------------------------------------------
    % Step1. Initialize all the appropriate variables. Also initialize the output
    %        parameter, just in case the program abhends in the middle.
    %-------------------------------------------------------------------------------
    number_of_angles  = length(theta_in_degrees);
    wds               = m_order(number_of_angles, 1, '2');
    R_angles          = theta_in_degrees(wds);
```

m_order.m (Code written by Chris Badea Email: chris@orion.mc.duke.edu)

```
%-------------------------------------------------------------------
% m_order function used wit WDS for the ART/MART reconstruction
% C. Badea Nov. 2k2
%-------------------------------------------------------------------
function m_order=create_order(n_of_pr, n_of_it, ordering)


     s_pr=0; %* index of selected projection *%

     %S; %* length of circular queue *%
     %Theta;%* circular queue *%
      %M; %* number of projections *%
      %Lambda; %* List of projections not yet used *%
      %temp;
      %L; %* length of projections not yet used *%
      %Q; %* number of proj currently in Theta *%
      %count_wds;
      first=1;
      order_ind=1;
      %i, iteration;
      %m;
%* Initialisation for wds *%
    M=n_of_pr;
    S=M; %* This is optimal number from the paper *%
%* First projection for reconstruction is 0. *%
```

```
    Q=1; %* Theta filled with first proj *%

    m_order(1)=1;

    Theta(1)=0;

    L=M-1;

% Fill in projection pool */
    for  i = 1: L
    Lambda(i)=i;
    end


    % Iteration loop */
    for iteration=1: n_of_it

        if (first==0)
                  L=M;
                  for  i = 1: L
                      Lambda(i)=i;
            end
        end


     %----
     % Projection loop */
          for m=1:n_of_pr


        %/* choice for ordering */
              if (ordering == '2')

                        if (first==1) first=0;
                        else
            %/* Select projection from Lambda */
                            order_ind = order_ind + 1;
                            s_pr = SelectProj(Lambda, L ,Theta, Q, S);
                            m_order(order_ind)=s_pr;
                %/* Change Lambda and Theta */
                % /* Cancel from Lambda */
                            count_wds = 1;

                            for i=1: L
                                if (Lambda(i)~= s_pr)
                                    temp(count_wds) = Lambda(i);
                                    count_wds=count_wds+1;
                    end
                end
                            L=L-1;
                            for i =1:L
                                    Lambda(i)=temp(i);
                end

                %/* Add to Theta */
                             if (Q < S)
                                 Q=Q+1;
                                 Theta(Q)= s_pr; %%%%% HERE !!!!

                             else   %/* with replacement */

                                      for i = 2:Q
                                          Theta(i-1)=Theta(i);
                    end
                                      Theta(Q)= s_pr;  %%%%% HERE !!!!
                 end
           end%/* for else first*/

          %/* for if ordering */
                else
                     m_order(order_ind)=m;
               order_ind = order_ind + 1;
        end
    end  %/* for m */

    end % iterations
```

SelectProj.m (Code written by Chris Badea Email: chris@orion.mc.duke.edu)

```
%-------------------------------------------------------------------------------
% sel_proj function used with wds for the ART/MART reconstruction
% C. Badea Nov. 2k2
%-------------------------------------------------------------------------------
function sel_prj=SelectProj(Lambda, L, Theta, Q, S)


%float *mu;%// Repulsive force

%float *sigma;
```

```
%float *sigmanorm;

%float *munorm;

%float *D;

sel_prj=0;

%int l,r,index_sel_pr;

%int dlq;

%float wq;

%float sum1, sum2, sum_dlq, dl_avg, min_temp, max_temp;


for  l = 1: L

    sum1=0.0;

    sum2=0.0;

    sum_dlq=0.0;


    for  r = 1:Q

        wq=(r+1)/Q;

        %// Compute minimal distance
        if abs(Theta(r)-Lambda(l)) < S-abs(Theta(r)-Lambda(l)) dlq=abs(Theta(r)-Lambda(l));
        else  dlq=  S-abs(Theta(r)-Lambda(l));
        end
            %dlq=lessval(abs(Theta(r)-Lambda(l)),S-abs(Theta(r)-Lambda(l)));

        sum_dlq = sum_dlq + dlq;

        %// Accumulate

        sum1 = sum1 + (wq * ( S /2.0-dlq));

        sum2 = sum2 + wq;

     end %//for r

    mu(l)=sum1/sum2;

    dl_avg = sum_dlq / Q;

    sum1=0;

    for  r = 1:Q

        wq=(r+1)/Q;
        if abs(Theta(r)-Lambda(l))<S-abs(Theta(r)-Lambda(l))  dlq= abs(Theta(r)-Lambda(l));
            else      dlq= S-abs(Theta(r)-Lambda(l));
        end

        %lessval(abs(Theta(r)-Lambda(l)),S-abs(Theta(r)-Lambda(l)));

        sum1 = sum1 + (wq*(dlq-dl_avg)*(dlq-dl_avg) );

    end    %// for r

    sigma(l)=sum1/sum2;

end   %//for l


%Normalize mu

%Find min of mu

min_temp=mu(1);

for  l = 2:L

    if (min_temp > mu(l)) min_temp = mu(l); end

end

%// Find max of mu

max_temp=mu(1);

for  l = 2:L

    if (max_temp < mu(l)) max_temp = mu(l); end

end
```

```
%// Compute mu_norm
for  l = 1:L
    if (max_temp ~= min_temp) munorm(l)=(mu(l)-min_temp)/(max_temp-min_temp);
    else munorm(l)=0.0; end
end

%//Normalize sigma
%// Find min of sigma
min_temp=sigma(1);
for  l = 2:L
    if (min_temp > sigma(l)) min_temp = sigma(l); end
end
%Find max of sigma
max_temp=sigma(1);
for  l = 2:L
if (max_temp < sigma(l)) max_temp = sigma(l); end
end
%// Compute sigma_norm
for  l = 1:L
if (max_temp ~= min_temp) sigmanorm(l)=(sigma(l)-min_temp)/(max_temp-min_temp);
    else sigmanorm(l)= 0.0; end
end
if 0
for  l = 1:L
aa= sprintf('Mu[%d]= %f  Sigma = %f ',l,munorm(l),sigmanorm(l));
%/printf(" Mu [%d] = %f  Sigma = %f  ",l,munorm[l],sigmanorm[l]);
disp(aa);
end
end
%// Compute D[]
for  l = 1:L
D(l)=munorm(l)*munorm(l)+0.5*sigmanorm(l)*sigmanorm(l);
end
%// Find minimal D[]
min_temp=D(1);
index_sel_pr=0;
for  l = 1:L
if (min_temp >= D(l))
   min_temp = D(l);
   index_sel_pr=l;
end
end
sel_prj=Lambda(index_sel_pr);
```

checkerboard.m

```
%=========================================================================================
% Created By : Girish Tirunelveli
% Created On : Januarty 21, 2003
% File Name  : checkerboard.m
% Description: This program asks for the size of the image and creates a checkerboard
%              image.
%=========================================================================================
```

```
function [cb_image] = checkerboard (length_of_square, pixel_value1, pixel_value2)
    %-------------------------------------------------------------------------------
    % Initialize all the required variables. All intialize the output parameter
    % just in case the program abhends in the middle.
    %-------------------------------------------------------------------------------
    cb_image         = zeros (length_of_square);
    next_pixel_value = 0;

    for rn = 1:length_of_square
        for cn = 1:length_of_square
            if mod(rn+cn, 2) == 0
                cb_image(rn,cn) = pixel_value1;
             else
               cb_image(rn,cn) = pixel_value2;
            end;
        end;
    end;
```

shingles.m

```
%=================================================================================================
% Created By : Girish Tirunelveli
% Created On : December 31, 2002
% File Name  : shingles.m
% Description: This program asks for the size of the image and creates a shingles data.
%              The pixel values increases
%=================================================================================================
function [sb_image] = shingles (length_of_square)
    %-------------------------------------------------------------------------------
    % Initialize all the required variables. All intialize the output parameter
    % just in case the program abhends in the middle.
    %-------------------------------------------------------------------------------
    sb_image         = zeros (length_of_square);
    next_pixel_value = 0;

    for rn = 1:length_of_square
        for cn = 1:length_of_square
            next_pixel_value  = next_pixel_value + 1;
            sb_image (rn, cn) = next_pixel_value;
        end;
    end;
```

sinewave.m

```
%=================================================================================================
% Created By : Girish Tirunelveli
% Created On : December 20, 2002
% File Name  : sinewave.m
% Description: This program creates a sine wave at a particular frequency.
%=================================================================================================
function [sinewave_img] = sinewave (length_of_square_img, frequency)
    if nargin == 1
        frequency = 1;
    end;

    sinewave_img = zeros(length_of_square_img);
    for column_no=1:length_of_square_img
        theta_in_degrees = column_no * 360 * frequency / length_of_square_img;
        theta_in_radians = theta_in_degrees * pi / 180;
        sinewave_img(:, column_no) = 128 + sin (theta_in_radians) * 127;
    end;
```

stretch.m

```
%=================================================================================
% Created By : Girish Tirunelveli
% Created On : December 25, 2002
% File Name  : stretch.m
% Description: Stretch the original_image by the stretch factor and return the
%              stretched image back.
%=================================================================================
function [stretched_image] = stretch (original_image, stretch_factor)

    %-------------------------------------------------------------------------
    % Resize the image according to the stretch factor. After resizing, get
    % only the center pixels.
    %-------------------------------------------------------------------------
    [no_of_rows, no_of_columns] = size (original_image);
```

```
stretched_image = imresize (original_image,  ...
                            [no_of_rows*stretch_factor no_of_columns*stretch_factor], ...
                            'bicubic', 'crop');
range_of_rows    = no_of_rows * (stretch_factor-1) / 2 + 1   : no_of_rows * (stretch_factor+1)/2;
range_of_columns = no_of_columns * (stretch_factor-1) / 2 + 1: no_of_columns * (stretch_factor+1)/2;
stretched_image  = stretched_image(range_of_rows, range_of_columns);
OMin=min(min(original_image));
OMax=max(max(original_image));
ZMin=min(min(stretched_image));
ZMax=max(max(stretched_image));
Scaling_Factor=(OMax-OMin)/(ZMax-ZMin);
stretched_image = stretched_image * Scaling_Factor;
```

blur.m

```
%=============================================================================================
% Created By : Girish Tirunelveli
% Created On : January 21, 2003
% File Name  : blur.m
% Description: Blur the original image by the blur factor
%=============================================================================================
function [blurred_image] = blur (original_image, blur_factor)

    %-------------------------------------------------------------------------
    % Resize the image according to the blur factor. After resizing, resize it
    % back to the original_size
    %-------------------------------------------------------------------------
    [no_of_rows, no_of_columns] = size (original_image);
    blurred_image = imresize (original_image,  ...
                              [no_of_rows*blur_factor no_of_columns*blur_factor], ...
                                'bicubic', 'crop');
    blurred_image = imresize (blurred_image, size(original_image), 'bicubic', 'crop');
    OMin=min(min(original_image));
    OMax=max(max(original_image));
    ZMin=min(min(blurred_image));
    ZMax=max(max(blurred_image));
    Scaling_Factor=(OMax-OMin)/(ZMax-ZMin);
    blurred_image = blurred_image * Scaling_Factor;
```

generate_wm_bin.m

```
%=============================================================================================
% Created By : Girish Tirunelveli
% Created On : December 25, 2002
% File Name  : generate_wm_bin.m
% Description: Compute Weight Matrix required for *ART transform.
%              The Weight matrix is computed based on the binary_scheme. If the center
%              of the pixel falls in the ray, then the pixel is said to be contributing
%              100% (=1) in the ray, else 0.
%=============================================================================================
function [weight_matrix] = generate_wm_bin(original_image, theta_in_degrees, number_of_detectors, detector_width,
pixel_width)

    %-----------------------------------------------------------------------------
    % Step1. Initialize all the variables appropriately. Also initialize the output
    %        parameter just in case the program abhends in the middle.
    %-----------------------------------------------------------------------------
    Dmax=detector_width/2;
    [number_of_rows_in_image, number_of_columns_in_image] = size(original_image);
    weight_matrix = 0;
    if mod(number_of_rows_in_image, 2) == 0    % Indicates even
        Centre_Row = number_of_rows_in_image/2;
      else
        Centre_Row = number_of_rows_in_image/2+0.5;
    end;
    if mod(number_of_columns_in_image, 2) == 0 % Indicates even
        Centre_Column = number_of_columns_in_image/2;
      else
        Centre_Column = number_of_columns_in_image/2+0.5;
    end;

    Centre_Detector = (number_of_detectors+1)/2 ;
    theta_in_radians=theta_in_degrees.*pi/180;

    %-----------------------------------------------------------------------------------------
    % Step2. For each projection angle, get the contribution of each pixel in a particular
    %        detector. This is the weight matrix. Note that the weight matrix will increase
    %        rapidly as the size of the image, projection angles and number of detectors
    %        increases.
    %-----------------------------------------------------------------------------------------
    for angle_number=1:length(theta_in_radians)
        costheta = cos(theta_in_radians(angle_number));
```

```
        sintheta = sin(theta_in_radians(angle_number));
        %--------------------------------------------------------------------------
        % If no detector width is specified (0) then assume variable detector width.
        %--------------------------------------------------------------------------
        if Dmax == 0
            Dmax=max(sintheta,costheta)/2;
        end;

        for detector_number=1:number_of_detectors
            P=(detector_number-Centre_Detector)* max(sintheta, costheta);
            %-----------------------------------------------------------------------------------
            % Equation of line P(1) is given by xcos(theta)+ysin(theta) -P = 0;
            % In matlab terminology it becomes -
            %             row_number*cos(theta) + column_number*sin(theta) - P = 0;
            % Since the Centre pixel is not the origin the equation gets changed to -
            %        (row_number-Centre_Row)*cos(theta) +
            %                        (column_number-Centre_Column)*sin(theta) - P = 0
            %        row_number*cos(theta) + column_number*sin(theta) -
            %                        Centre_Row*cost(theta)-Centre_Column*sin(theta) - P= 0;
            %-----------------------------------------------------------------------------------
            C=-1*Centre_Row*costheta - Centre_Column*sintheta - P;
            A=costheta;
            B=sintheta;

            %-----------------------------------------------------------------
            % Perpendicular distance of (1,1) from the line Ax+By+C=0 is
            %      D=abs(A(1) + B(1) +C / sqrt (A^2+B^2));
            %-----------------------------------------------------------------
            for pixel_row_number = 1:number_of_rows_in_image
                for pixel_column_number = 1:number_of_columns_in_image
                    D=abs(A*pixel_row_number+B*pixel_column_number+C)/sqrt(A^2+B^2);
                    if D>Dmax
                        weight_factor = 0;
                      else
                        weight_factor = 1;
                    end;
                    weight_matrix(pixel_row_number, pixel_column_number, ...
                                  angle_number, detector_number) = weight_factor;

                end; % End for column_number = 1:number_of_columns_in_image
            end; % End for row_number = 1:number_of_rows_in_image


        end;  % End for detector_number=1:number_of_detectors
    end; % End for angle_number=1:length(theta)
```

generate_wm_int.m

```
%=========================================================================================
% Created By : Girish Tirunelveli
% Created On : December 25, 2002
% File Name  : generate_wm_int.m
% Description: Compute Weight Matrix required for *ART transform.
%              The Weight matrix is computed based on the length of the ray within
%              the pixel scheme. It is called "Int" scheme as it is the integration of
%              of the ray length within a pixel. In this scheme, the weight that  a
%              pixel can contribute to a ray can be greater than 1.
%=========================================================================================
function [weight_matrix] = generate_wm_int(original_image, theta_in_degrees, number_of_detectors, detector_width,
pixel_width)

    %--------------------------------------------------------------------------
    % Step1. Initialize all the variables appropriately. Also initialize the output
    %        parameter just in case the program abhends in the middle.
    %--------------------------------------------------------------------------
    Dmax=detector_width/2;
    [number_of_rows_in_image, number_of_columns_in_image] = size(original_image);
    weight_matrix = 0;
    if mod(number_of_rows_in_image, 2) == 0    % Indicates even
        Centre_Row = number_of_rows_in_image/2;
      else
        Centre_Row = number_of_rows_in_image/2+0.5;
    end;
    if mod(number_of_columns_in_image, 2) == 0 % Indicates even
        Centre_Column = number_of_columns_in_image/2;
      else
        Centre_Column = number_of_columns_in_image/2+0.5;
    end;

    Centre_Detector = (number_of_detectors+1)/2 ;
    theta_in_radians=theta_in_degrees.*pi/180;

    %-----------------------------------------------------------------------------------------
    % Step2. For each projection angle, get the contribution of each pixel in a particular
    %        detector. This is the weight matrix. Note that the weight matrix will increase
    %        rapidly as the size of the image, projection angles and number of detectors
    %        increases.
    %-----------------------------------------------------------------------------------------
```

```
for angle_number=1:length(theta_in_radians)
    costheta = cos(theta_in_radians(angle_number));
    sintheta = sin(theta_in_radians(angle_number));
    %-------------------------------------------------------------------------------
    % If no detector width is specified (0) then assume variable detector width.
    %-------------------------------------------------------------------------------
    if Dmax == 0
        Dmax=max(sintheta,costheta)/2;
    end;

    for detector_number=1:number_of_detectors
        P=(detector_number-Centre_Detector)* max(sintheta, costheta);
        %-------------------------------------------------------------------------------------------
        % Equation of line P(1) is given by xcos(theta)+ysin(theta) -P = 0;
        % In matlab terminology it becomes -
        %            row_number*cos(theta) + column_number*sin(theta) - P = 0;
        % Since the Centre pixel is not the origin the equation gets changed to -
        %       (row_number-Centre_Row)*cos(theta) +
        %                    (column_number-Centre_Column)*sin(theta) - P = 0
        %       row_number*cos(theta) + column_number*sin(theta) -
        %                    Centre_Row*cost(theta)-Centre_Column*sin(theta) - P= 0;
        %-------------------------------------------------------------------------------------------
        C=-1*Centre_Row*costheta - Centre_Column*sintheta - P;
        A=costheta;
        B=sintheta;

        %-----------------------------------------------------------------
        % If the center of the pixel is (1,1), the 4 lines that identifies
        % the pixel are x=1-pixel_width/2, x=1+pixel_width/2
        %          and y=1-pixel_width/2, y=1+pixel_width/2
        % Find the point of intersection of each of this lines with the
        % ray under consideration.
        % Point of intersection is calculated by simply substituting the
        % x or the y coordinate in the equation of the line.
        %-----------------------------------------------------------------
        for pixel_row_number = 1:number_of_rows_in_image
            for pixel_column_number = 1:number_of_columns_in_image
                % Point of intersection between ray line and line 1 is
                P1_y = pixel_row_number-pixel_width/2;
                P1_x = (- C - A*P1_y)/B;
                P3_y = pixel_row_number+pixel_width/2;
                P3_x = (- C - A*P3_y)/B;

                P2_x = pixel_column_number-pixel_width/2;
                P2_y = (-C - B*P2_x)/A;
                P4_x = pixel_column_number+pixel_width/2;
                P4_y = (-C - B*P4_x)/A;

                % Convert all points with precision multiplier of 10.
                P1_x = precision (P1_x);
                P1_y = precision (P1_y);
                P2_x = precision (P2_x);
                P2_y = precision (P2_y);
                P3_x = precision (P3_x);
                P3_y = precision (P3_y);
                P4_x = precision (P4_x);
                P4_y = precision (P4_y);

                % If the point of intersection is outside the pixel then
                % ignore it.
                d12 = distance (P1_x, P1_y, P2_x, P2_y);
                d13 = distance (P1_x, P1_y, P3_x, P3_y);
                d14 = distance (P1_x, P1_y, P4_x, P4_y);
                d23 = distance (P2_x, P2_y, P3_x, P3_y);
                d24 = distance (P2_x, P2_y, P4_x, P4_y);
                d34 = distance (P3_x, P3_y, P4_x, P4_y);
                if P1_x< pixel_column_number-pixel_width/2 | P1_x > pixel_column_number+pixel_width/2
                    d12 = 0;
                    d13 = 0;
                    d14 = 0;
                end;
                if P1_y< pixel_row_number-pixel_width/2 | P1_y > pixel_row_number+pixel_width/2
                    d12 = 0;
                    d13 = 0;
                    d14 = 0;
                end;

                if P3_x< pixel_column_number-pixel_width/2 | P3_x > pixel_column_number+pixel_width/2
                    d13 = 0;
                    d23 = 0;
                    d34 = 0;
                end;
                if P3_y< pixel_row_number-pixel_width/2 | P3_y > pixel_row_number+pixel_width/2
                    d13 = 0;
                    d23 = 0;
                    d34 = 0;
                end;
                if P2_x< pixel_column_number-pixel_width/2 | P2_x > pixel_column_number+pixel_width/2
                    d12 = 0;
                    d23 = 0;
                    d24 = 0;
                end;
                if P2_y< pixel_row_number-pixel_width/2 | P2_y > pixel_row_number+pixel_width/2
```

```
                    d12 = 0;
                    d23 = 0;
                    d24 = 0;
                end;
                if P4_x< pixel_column_number-pixel_width/2 | P4_x > pixel_column_number+pixel_width/2
                    d14 = 0;
                    d24 = 0;
                    d34 = 0;
                end;
                if P4_y< pixel_row_number-pixel_width/2 | P4_y > pixel_row_number+pixel_width/2
                    d14 = 0;
                    d24 = 0;
                    d34 = 0;
                end;
                weight_factor = max([d12,d13,d14, d23,d24, d34]);

                weight_matrix(pixel_row_number, pixel_column_number, ...
                              angle_number, detector_number) = weight_factor;

            end; % End for column_number = 1:number_of_columns_in_image
        end; % End for row_number = 1:number_of_rows_in_image


    end;  % End for detector_number=1:number_of_detectors
end; % End for angle_number=1:length(theta)



%=============================================================================================
%  Sub-Function:  distance: Calculates the distance between two points, The
%                 x and y co-ordinates of the two points has to be supplied.
%                 If this subfunction is not able to calculate the distance,
%                 then 0 is returned.
%=============================================================================================
function [distance] = distance (x1, y1, x2, y2)
    distance = sqrt( (x1-x2)^2 + (y1-y2)^2);
    if (isfinite(distance) == 0)
        distance = 0;
    end;

%=============================================================================================
%  Sub-Function:  precision: Returns the value where the number of decimal points is
%                 equal to the number of 0's in the precision multiplier.
%=============================================================================================
function [R_value] = precision (I_value)
    R_value = round(I_value * 10000000000) / 10000000000;
```

generate_wm_dist.m

```
%=============================================================================================
% Created By : Girish Tirunelveli
% Created On : December 25, 2002
% File Name  : generate_wm_dist.m
% Description: Compute Weight Matrix required for *ART transform.
%              The Weight matrix is computed based on the distance of the center
%              of the ray from the center of the pixel logic.
%=============================================================================================
function [weight_matrix] = generate_wm_dist(original_image, theta_in_degrees, number_of_detectors, detector_width,
pixel_width)

    %---------------------------------------------------------------------------------
    % Step1. Initialize all the variables appropriately. Also initialize the output
    %        parameter just in case the program abhends in the middle.
    %---------------------------------------------------------------------------------
    Dmax=detector_width/2;
    [number_of_rows_in_image, number_of_columns_in_image] = size(original_image);
    weight_matrix = 0;
    if mod(number_of_rows_in_image, 2) == 0    % Indicates even
        Centre_Row = number_of_rows_in_image/2;
      else
        Centre_Row = number_of_rows_in_image/2+0.5;
    end;
    if mod(number_of_columns_in_image, 2) == 0 % Indicates even
        Centre_Column = number_of_columns_in_image/2;
      else
        Centre_Column = number_of_columns_in_image/2+0.5;
    end;

    Centre_Detector = (number_of_detectors+1)/2 ;
    theta_in_radians=theta_in_degrees.*pi/180;

    %-----------------------------------------------------------------------------------------
    % Step2. For each projection angle, get the contribution of each pixel in a particular
    %        detector. This is the weight matrix. Note that the weight matrix will increase
    %        rapidly as the size of the image, projection angles and number of detectors
    %        increases.
    %-----------------------------------------------------------------------------------------
```

```
    for angle_number=1:length(theta_in_radians)
        costheta = cos(theta_in_radians(angle_number));
        sintheta = sin(theta_in_radians(angle_number));
        %-------------------------------------------------------------------------------
        % If no detector width is specified (0) then assume variable detector width.
        %-------------------------------------------------------------------------------
        if Dmax == 0
            Dmax=max(sintheta,costheta)/2;
        end;

        for detector_number=1:number_of_detectors
            P=(detector_number-Centre_Detector)* max(sintheta, costheta);
            %-------------------------------------------------------------------------------------------
            % Equation of line P(1) is given by xcos(theta)+ysin(theta) -P = 0;
            % In matlab terminology it becomes -
            %              row_number*cos(theta) + column_number*sin(theta) - P = 0;
            % Since the Centre pixel is not the origin the equation gets changed to -
            %        (row_number-Centre_Row)*cos(theta) +
            %                        (column_number-Centre_Column)*sin(theta) - P = 0
            %       row_number*cos(theta) + column_number*sin(theta) -
            %                       Centre_Row*cost(theta)-Centre_Column*sin(theta) - P= 0;
            %-------------------------------------------------------------------------------------------
            C=-1*Centre_Row*costheta - Centre_Column*sintheta - P;
            A=costheta;
            B=sintheta;

            %-----------------------------------------------------------------
            % Perpendicular distance of (1,1) from the line Ax+By+C=0 is
            %      D=abs(A(1) + B(1) +C / sqrt (A^2+B^2));
            %-----------------------------------------------------------------
            for pixel_row_number = 1:number_of_rows_in_image
                for pixel_column_number = 1:number_of_columns_in_image
                    D=abs(A*pixel_row_number+B*pixel_column_number+C)/sqrt(A^2+B^2);
                    if D>Dmax
                        weight_factor = 0;
                      else
                        weight_factor = 1-D/Dmax;
                    end;
                    weight_matrix(pixel_row_number, pixel_column_number, ...
                                  angle_number, detector_number) = weight_factor;

                end; % End for column_number = 1:number_of_columns_in_image
            end; % End for row_number = 1:number_of_rows_in_image


        end;  % End for detector_number=1:number_of_detectors
    end; % End for angle_number=1:length(theta)
```

generate_wm_cont.m

```
%=============================================================================================
% Created By : Girish Tirunelveli
% Created On : December 25, 2002
% File Name  : generate_wm_cont.m
% Description: Compute Weight Matrix required for *ART transform. The weight matrix in
%              this program is computed by considering the contribution made by the
%              pixel in the neighbouring rays. It makes sure that the sum of the
%              contribution that the pixel makes in all rays equals one.
%=============================================================================================
function [weight_matrix] = generate_wm_cont(original_image, theta_in_degrees, number_of_detectors, detector_width,
pixel_width)

    %-------------------------------------------------------------------------------
    % Step1. Initialize all the variables appropriately. Also initialize the output
    %        parameter just in case the program abhends in the middle.
    %-------------------------------------------------------------------------------
    Dmax1=detector_width/2;
    Dmax2=sqrt(1+1)/2;
    [number_of_rows_in_image, number_of_columns_in_image] = size(original_image);
    weight_matrix = 0;
    if mod(number_of_rows_in_image, 2) == 0    % Indicates even
        Centre_Row = number_of_rows_in_image/2;
      else
        Centre_Row = number_of_rows_in_image/2+0.5;
    end;
    if mod(number_of_columns_in_image, 2) == 0 % Indicates even
        Centre_Column = number_of_columns_in_image/2;
      else
        Centre_Column = number_of_columns_in_image/2+0.5;
    end;

    Centre_Detector = (number_of_detectors+1)/2 ;
    theta_in_radians=theta_in_degrees.*pi/180;

    %-------------------------------------------------------------------------------------------
```

```
% Step2. For each projection angle, get the contribution of each pixel in a particular
%        detector. This is the weight matrix. Note that the weight matrix will increase
%        rapidly as the size of the image, projection angles and number of detectors
%        increases.
%-------------------------------------------------------------------------------------------
for angle_number=1:length(theta_in_radians)
    costheta = cos(theta_in_radians(angle_number));
    sintheta = sin(theta_in_radians(angle_number));

    for detector_number=1:number_of_detectors
        P=detector_number-Centre_Detector;
        %-------------------------------------------------------------------------------
        % Equation of line P(1) is given by xcos(theta)+ysin(theta) -P = 0;
        % In matlab terminology it becomes -
        %             row_number*cos(theta) + column_number*sin(theta) - P = 0;
        % Since the Centre pixel is not the origin the equation gets changed to -
        %        (row_number-Centre_Row)*cos(theta) +
        %                        (column_number-Centre_Column)*sin(theta) - P = 0
        %        row_number*cos(theta) + column_number*sin(theta) -
        %                        Centre_Row*cost(theta)-Centre_Column*sin(theta) - P= 0;
        %-------------------------------------------------------------------------------
        C_previous =-1*Centre_Row*costheta - Centre_Column*sintheta - P-1;
        C_current  =-1*Centre_Row*costheta - Centre_Column*sintheta - P;
        C_next     =-1*Centre_Row*costheta - Centre_Column*sintheta - P+1;
        A          = costheta;
        B          = sintheta;

        %-----------------------------------------------------------------
        % Perpendicular distance of (1,1) from the line Ax+By+C=0 is
        %      D=abs(A(1) + B(1) +C / sqrt (A^2+B^2));
        %-----------------------------------------------------------------
        for pixel_row_number = 1:number_of_rows_in_image
            for pixel_column_number = 1:number_of_columns_in_image
                %-------------------------------------------------------------------------
                % How much has this pixel contributed in the previous detector
                % "contribution_previous
                % How much is this pixel going to contribute in the next detector
                % "contribution_next
                % The amount that this pixel is going to contribute in this ray is
                %   contribution_current = 1-(contribution_previous+contribution_next)
                %-------------------------------------------------------------------------
                D_previous = abs(A*pixel_row_number+B*pixel_column_number+C_previous)/sqrt(A^2+B^2);
                D_current  = abs(A*pixel_row_number+B*pixel_column_number+C_current)/sqrt(A^2+B^2);
                D_next     = abs(A*pixel_row_number+B*pixel_column_number+C_next)/sqrt(A^2+B^2);

                D_previous_edge = D_previous/2;
                D_next_edge = D_next/2;

                if (D_previous_edge > Dmax2)
                    contribution_previous = 0;
                else
                    contribution_previous = 1-D_previous_edge/Dmax1;
                    if contribution_previous < 0
                        contribution_previous = 0;
                    end;
                end;
                if (D_next_edge > Dmax2)
                    contribution_next = 0;
                else
                    contribution_next = 1-D_next_edge/Dmax1;
                    if contribution_next < 0
                        contribution_next = 0;
                    end;
                end;
                if (D_current > Dmax2)
                    weight_factor = 0;
                  else
                    weight_factor = 1- (contribution_previous+contribution_next);
                end;
                weight_matrix(pixel_row_number, pixel_column_number, ...
                              angle_number, detector_number) = weight_factor;

            end; % End for column_number = 1:number_of_columns_in_image
        end; % End for row_number = 1:number_of_rows_in_image


    end;  % End for detector_number=1:number_of_detectors
end; % End for angle_number=1:length(theta)
```

generate_projections.m

```
%=============================================================================================
% Created By : Girish Tirunelveli
% Created On : December 25, 2002
% File Name  : generate_projections.m
% Description: Compute Forward Simulation. i.e. Project the image on the detectors
%              and determine the projected value.
%=============================================================================================
function [projected_value] = generate_projections (original_image, theta_in_degrees, ...
                                                   weight_matrix, number_of_detectors)
```

```
    for angle_number=1:length(theta_in_degrees)
        for detector_number=1:number_of_detectors
            projected_value(angle_number, detector_number) = ...
                            sum(sum(weight_matrix(:,:,angle_number,detector_number).* original_image));
        end;
    end;
```

ifbp.m

```
%=============================================================================================
% Created By : Girish Tirunelveli
% Created On : January 03, 2002
% File Name  : ifbp.m
% Description: Compute inverse FBP (Fourier Backprojection Technique) transform.
%              The iFBP function computes the inverse FBP transform, and reconstructs
%              the result_image.
%=============================================================================================
function [result_image, image_difference] = ifbp(seed_image, original_image, projection_values, theta_in_degrees,
number_of_cycles, number_of_detectors, weight_matrix, precision_multiplier, relaxation_factor)
    %---------------------------------------------------------------------------------------
    % Step1. Initialize all appropriate variables. Also initialize the output parameters,
    %        just in case the job abhends in the middle.
    %---------------------------------------------------------------------------------------
    warning off   % To suppress the Divide by zero warning.
    result_image        = zeros(size(seed_image));
    RI2=0;
    [number_of_rows_in_image, number_of_columns_in_image] = size(original_image);

    %-------------------------------------------------------------------------
    % Step2. Since FBP considers radon transform and not the weight matrix
    %        for generating forward projections get the forward projections
    %        and compute the backward reconstruction.
    %-------------------------------------------------------------------------
    projection_values = radon(original_image,theta_in_degrees);
    result_image      = iradon(projection_values,theta_in_degrees,'nearest','Hann', 1, number_of_rows_in_image);

    %-------------------------------------------------------------------------
    % Step3. Since the algorithm is not dependent on the seed and the number of
    %        cycles, the image difference is the difference between the
    %        original image and the result image and is the same for all the
    %        cycles. We do not have to do based on cycles, but is done
    %        to be consistent with our other algorithms.
    %-------------------------------------------------------------------------
    image_difference(1) = image_difference_ed (original_image, result_image);
    for cycle_number=1:number_of_cycles
        image_difference(cycle_number+1) = image_difference(1);
    end;

    %=============================================================================================
    %  Sub-Function:  image_difference_ed: Calculates the image difference between two
    %                 images based on Euclidean distance.
    %=============================================================================================
    function [ID] = image_difference_ed (image1, image2)
        diff_img = image1 - image2;
        diff_img = diff_img.^2;
        [rn, cn] = size (image1);
        ID       = sqrt (sum(sum(diff_img)))/ (sqrt(rn*cn)*255);
```

isrt.m

```
%=============================================================================================
% Created By : Girish Tirunelveli
% Created On : January 03, 2002
% File Name  : isrt.m
% Description: Compute inverse SRT (Summation Reconstruction Technique) transform.
%              The iSRT function computes the inverse SRT transform, and reconstructs
%              the result_image.
%=============================================================================================
function [result_image, image_difference] = isrt(seed_image, original_image, projection_values, theta_in_degrees,
number_of_cycles, number_of_detectors, weight_matrix, precision_multiplier, relaxation_factor)
    %---------------------------------------------------------------------------------------
    % Step1. Initialize all appropriate variables. Also initialize the output parameters,
    %        just in case the job abhends in the middle.
    %---------------------------------------------------------------------------------------
    warning off   % To suppress the Divide by zero warning.
    result_image  = zeros(size(seed_image));
    RI2=0;
    [number_of_rows_in_image, number_of_columns_in_image] = size(original_image);

    %-------------------------------------------------------------------
    % Step2. Add the contribution that each pixel makes in all the rays.
    %-------------------------------------------------------------------
    for row_number=1:number_of_rows_in_image
        for column_number=1:number_of_columns_in_image
```

```
            for angle_number=1:length(theta_in_degrees)
                for detector_number=1:number_of_detectors
                    RI2=RI2+weight_matrix(row_number, column_number, angle_number,detector_number) * ...
                                projection_values(angle_number, detector_number);
                end;
            end;
            Z(row_number, column_number) = RI2;
            RI2=0;
        end;
    end;

    %-------------------------------------------------------------------------
    % Step3. Since the algorithm adds up all the contribution, the result is too
    %        large. Scaling is required. Scale according to the original.
    %-------------------------------------------------------------------------
    OMin=min(min(original_image));
    OMax=max(max(original_image));
    ZMin=min(min(Z));
    ZMax=max(max(Z));
    Scaling_Factor=(OMax-OMin)/(ZMax-ZMin);
    result_image = Z * Scaling_Factor;

    %-------------------------------------------------------------------------
    % Step4. Since the algorithm is not dependent on the seed and the number of
    %        cycles, the image difference is the difference between the
    %        original image and the result image and is the same for all the
    %        cycles. We do not have to do based on cycles, but is done
    %        to be consistent with our other algorithms.
    %-------------------------------------------------------------------------
    image_difference(1) = image_difference_ed (original_image, result_image);
    for cycle_number=1:number_of_cycles
        image_difference(cycle_number+1) = image_difference(1);
    end;

    %=========================================================================================
    %  Sub-Function:  image_difference_ed: Calculates the image difference between two
    %                 images based on Euclidean distance.
    %=========================================================================================
    function [ID] = image_difference_ed (image1, image2)
        diff_img = image1 - image2;
        diff_img = diff_img.^2;
        [rn, cn] = size (image1);
        ID       = sqrt (sum(sum(diff_img)))/ (sqrt(rn*cn)*255);
```

iart.m

```
%=========================================================================================
% Created By : Girish Tirunelveli
% Created On : December 25, 2002
% File Name  : iart.m
% Description: Compute inverse ART (Additive Reconstruction Technique) transform.
%              The iART function computes the inverse ART transform, and reconstructs
%              the result_image.
%=========================================================================================
function [result_image, image_difference] = iart(seed_image, original_image, projection_values, theta_in_degrees,
number_of_cycles, number_of_detectors, weight_matrix, precision_multiplier, relaxation_factor)

    %-------------------------------------------------------------------------------------
    % Step1. Initialize all appropriate variables. Also initialize the output parameters,
    %        just in case the job abhends in the middle.
    %-------------------------------------------------------------------------------------
    warning off   % To suppress the Divide by zero warning.
    result_image        = zeros(size(seed_image));
    image_difference(1) = image_difference_ed (original_image, seed_image);

    for cycle_number = 1:number_of_cycles
        new_seed_image = seed_image;
        for angle_number = 1:length(theta_in_degrees)
            %disp (sprintf('********************NEW CYCLE for angle %d*********************', angle_number));

            for detector_number=1:number_of_detectors
                PV_seed(angle_number, detector_number) = ...
                                sum(sum(weight_matrix(:,:,angle_number,detector_number).*seed_image));

                number_of_contributing_pixels=length(find(weight_matrix(:,:, angle_number, detector_number)));
                adding_factor = (projection_values(angle_number, detector_number) - ...
                                        PV_seed(angle_number, detector_number))/ ...
                                        number_of_contributing_pixels;

                if isfinite(adding_factor) == 1
                    W1=weight_matrix(:,:,angle_number, detector_number);
                    z=find(W1);
                    seed_image(z) = seed_image(z)+relaxation_factor*adding_factor;
                end;


            end; % End for detector_number=1:number_of_detectors
```

```
        %disp (sprintf('********************END OF CYCLE for angle%d********************', angle_number));

    end;  % End for angle_number = 1:length(theta_in_degrees)

    image_difference(cycle_number+1) = image_difference_ed(original_image, seed_image);
    %----------------------------------
    % Apply the convergence criteria.
    %----------------------------------
    if image_difference(cycle_number+1) - image_difference(cycle_number) >= 0
        seed_image = new_seed_image;
        image_difference(cycle_number+1) = image_difference(cycle_number);
        for new_cycle_number = cycle_number:number_of_cycles
            image_difference(new_cycle_number+1) = image_difference(new_cycle_number);
        end;
        break;
    end;
    seed_image = abs(seed_image);  % Positivity Constraint

end; % End for cycle_number = 1:number_of_cycles

result_image=seed_image;


%=============================================================================================
%  Sub-Function:  image_difference_ed: Calculates the image difference between two
%                 images based on Euclidean distance.
%=============================================================================================
function [ID] = image_difference_ed (image1, image2)
    diff_img = image1 - image2;
    diff_img = diff_img.^2;
    [rn, cn] = size (image1);
    ID       = sqrt (sum(sum(diff_img)))/ (sqrt(rn*cn)*255);
```

imart.m

```
%=============================================================================================
% Created By : Girish Tirunelveli
% Created On : December 25, 2002
% File Name  : imart.m
% Description: Compute inverse MART transform.
%              The iMART function computes the inverse MART transform, and reconstructs
%              the result_image.
%=============================================================================================
function [result_image, image_difference] = imart(seed_image, original_image, projection_values, theta_in_degrees,
number_of_cycles, number_of_detectors, weight_matrix, precision_multiplier, relaxation_factor)


    %---------------------------------------------------------------------------------------
    % Step1. Initialize all appropriate variables. Also initialize the output parameters,
    %        just in case the job abhends in the middle.
    %---------------------------------------------------------------------------------------
    warning off   % To suppress the Divide by zero warning.
    result_image        = zeros(size(seed_image));
    image_difference(1) = image_difference_ed (original_image, seed_image);

    for cycle_number = 1:number_of_cycles
        new_seed_image = seed_image;
        for angle_number = 1:length(theta_in_degrees)
            %disp (sprintf('********************NEW CYCLE for angle %d********************', angle_number));

            for detector_number=1:number_of_detectors
                PV_seed(angle_number, detector_number) = ...
                                sum(sum(weight_matrix(:,:,angle_number,detector_number).*seed_image));

                multiplying_factor = projection_values(angle_number, detector_number)./ ...
                                                PV_seed(angle_number, detector_number);
                multiplying_factor = multiplying_factor.^0.1;

                if isfinite(multiplying_factor) == 1
                    W1=weight_matrix(:,:,angle_number, detector_number);
                    z=find(W1);
                    seed_image(z) = seed_image(z)* relaxation_factor*multiplying_factor;
                end;


            end; % End for detector_number=1:number_of_detectors

            %disp (sprintf('********************END OF CYCLE for angle%d********************', angle_number));

        end;  % End for angle_number = 1:length(theta_in_degrees)

        image_difference(cycle_number+1) = image_difference_ed(original_image, seed_image);
        %----------------------------------
        % Apply the convergence criteria.
        %----------------------------------
        if image_difference(cycle_number+1) - image_difference(cycle_number) >= 0
            seed_image = new_seed_image;
            image_difference(cycle_number+1) = image_difference(cycle_number);
```

```
            for new_cycle_number = cycle_number:number_of_cycles
                image_difference(new_cycle_number+1) = image_difference(new_cycle_number);
            end;
            break;
        end;
        seed_image = abs(seed_image);  % Positivity Constraint


    end; % End for cycle_number = 1:number_of_cycles


    result_image=seed_image;


    %=============================================================================================
    %  Sub-Function:  image_difference_ed: Calculates the image difference between two
    %                 images based on Euclidean distance.
    %=============================================================================================
    function [ID] = image_difference_ed (image1, image2)
        diff_img = image1 - image2;
        diff_img = diff_img.^2;
        [rn, cn] = size (image1);
        ID       = sqrt (sum(sum(diff_img)))/ (sqrt(rn*cn)*255);
```

isirt.m

```
%=============================================================================================
% Created By : Girish Tirunelveli
% Created On : December 25, 2002
% File Name  : isirt.m
% Description: Compute inverse SIRT (Simultaneous Iterative Reconstruction Technique)
%              transform. The iSIRT function computes the inverse SIRT transform,
%              and reconstructs the result_image.
%=============================================================================================
function [result_image, image_difference] = isirt(seed_image, original_image, projection_values, theta_in_degrees,
number_of_cycles, number_of_detectors, weight_matrix, precision_multiplier, relaxation_factor)

    %---------------------------------------------------------------------------------------
    % Step1. Initialize all appropriate variables. Also initialize the output parameters,
    %        just in case the job abhends in the middle.
    %---------------------------------------------------------------------------------------
    warning off   % To suppress the Divide by zero warning.
    result_image        = zeros(size(seed_image));
    image_difference(1) = image_difference_ed (original_image, seed_image);

    for cycle_number = 1:number_of_cycles
        new_seed_image = seed_image;
        correction_matrix = zeros(size(seed_image));
        corrections_done_matrix = zeros(size(seed_image));
        for angle_number = 1:length(theta_in_degrees)
            %disp (sprintf('********************NEW CYCLE for angle %d*********************', angle_number));

            for detector_number=1:number_of_detectors
                PV_seed(angle_number, detector_number) = ...
                                sum(sum(weight_matrix(:,:,angle_number,detector_number).*seed_image));

                number_of_contributing_pixels=length(find(weight_matrix(:,:, angle_number, detector_number)));
                adding_factor = (projection_values(angle_number, detector_number) - ...
                                        PV_seed(angle_number, detector_number))/ ...
                                        number_of_contributing_pixels;

                if isfinite(adding_factor) == 1
                    W1=weight_matrix(:,:,angle_number, detector_number);
                    z=find(W1);
                    correction_matrix(z) = correction_matrix(z)+adding_factor;
                    corrections_done_matrix(z) = corrections_done_matrix(z)+1;
                    %seed_image(z) = seed_image(z)+adding_factor;
                end;


            end; % End for detector_number=1:number_of_detectors

            %disp (sprintf('********************END OF CYCLE for angle%d*********************', angle_number));

        end;  % End for angle_number = 1:length(theta_in_degrees)

        seed_image=seed_image+relaxation_factor*correction_matrix./corrections_done_matrix;
        image_difference(cycle_number+1) = image_difference_ed(original_image, seed_image);
        %-----------------------------------
        % Apply the convergence criteria.
        %-----------------------------------
        if image_difference(cycle_number+1) - image_difference(cycle_number) >= 0
            seed_image = new_seed_image;
            image_difference(cycle_number+1) = image_difference(cycle_number);
            for new_cycle_number = cycle_number:number_of_cycles
                image_difference(new_cycle_number+1) = image_difference(new_cycle_number);
            end;
            break;
```

```
        end;
        seed_image = abs(seed_image);  % Positivity Constraint

    end; % End for cycle_number = 1:number_of_cycles

    result_image=seed_image;


    %=============================================================================================
    %  Sub-Function:  image_difference_ed: Calculates the image difference between two
    %                 images based on Euclidean distance.
    %=============================================================================================
    function [ID] = image_difference_ed (image1, image2)
        diff_img = image1 - image2;
        diff_img = diff_img.^2;
        [rn, cn] = size (image1);
        ID       = sqrt (sum(sum(diff_img)))/ (sqrt(rn*cn)*255);
```

isart.m

```
%=============================================================================================
% Created By : Girish Tirunelveli
% Created On : December 25, 2002
% File Name  : isart.m
% Description: Compute inverse SART (Simultaneous Additive Reconstruction Technique)
%              transform. The iSART function computes the inverse SART transform,
%              and reconstructs the result_image.
%=============================================================================================
function [result_image, image_difference] = isart(seed_image, original_image, projection_values, theta_in_degrees,
number_of_cycles, number_of_detectors, weight_matrix, precision_multiplier, relaxation_factor)

    %---------------------------------------------------------------------------------------------
    % Step1. Initialize all appropriate variables. Also initialize the output parameters,
    %        just in case the job abhends in the middle.
    %---------------------------------------------------------------------------------------------
    warning off   % To suppress the Divide by zero warning.
    result_image        = zeros(size(seed_image));
    image_difference(1) = image_difference_ed (original_image, seed_image);

    [x,y]=size(seed_image);
    for cycle_number = 1:number_of_cycles
        new_seed_image = seed_image;
        correction_matrix = zeros(size(seed_image));
        corrections_done_matrix = zeros(size(seed_image));
        for angle_number = 1:length(theta_in_degrees)
            %disp (sprintf('********************NEW CYCLE for angle %d*********************', angle_number));

            for detector_number=1:number_of_detectors
                PV_seed(angle_number, detector_number) = ...
                                sum(sum(weight_matrix(:,:,angle_number,detector_number).*seed_image));

                difference = projection_values(angle_number, detector_number) - ...
                                            PV_seed(angle_number, detector_number);
                denominator = sum(sum(weight_matrix(:,:,angle_number, detector_number)));
                if denominator ~= 0
                    W1=weight_matrix(:,:,angle_number, detector_number);
                    z=find(W1);
                    correction_matrix(z) = correction_matrix(z)+ ...
                                                difference/denominator.*W1(z);
                    corrections_done_matrix(z) = corrections_done_matrix(z)+1;
                end;

            end; % End for detector_number=1:number_of_detectors

            %disp (sprintf('********************END OF CYCLE for angle%d*********************', angle_number));

        end;  % End for angle_number = 1:length(theta_in_degrees)

        seed_image=seed_image+relaxation_factor*correction_matrix./corrections_done_matrix;
        image_difference(cycle_number+1) = image_difference_ed(original_image, seed_image);
        %-----------------------------------
        % Apply the convergence criteria.
        %-----------------------------------
        if image_difference(cycle_number+1) - image_difference(cycle_number) >= 0
            seed_image = new_seed_image;
            image_difference(cycle_number+1) = image_difference(cycle_number);
            for new_cycle_number = cycle_number:number_of_cycles
                image_difference(new_cycle_number+1) = image_difference(new_cycle_number);
            end;
            break;
        end;
        seed_image = abs(seed_image);  % Positivity Constraint


    end; % End for cycle_number = 1:number_of_cycles

    result_image=seed_image;
```

```
%=============================================================================================
%  Sub-Function:  image_difference_ed: Calculates the image difference between two
%                 images based on Euclidean distance.
%=============================================================================================
function [ID] = image_difference_ed (image1, image2)
    diff_img = image1 - image2;
    diff_img = diff_img.^2;
    [rn, cn] = size (image1);
    ID       = sqrt (sum(sum(diff_img)))/ (sqrt(rn*cn)*255);
```

Appendix B

Tabular Results of the Square pixel and Hexagonal pixel Resolution Comparison Experiment

This appendix contains the results of the square vs. hexagon pixel comparison experiment. The length of square pixel is 8, the length of hexagon pixel is 4.5 and the rotation is done from 0 to 360 degrees in intervals of 5 for all experiments. In case of MTF results, the data is tabulated based on frequency and not on the rotation angle.

B.1 Results for admin256.bmp image

Rotation in Degrees	SIQ_ed	HIQ_ed	SIQ_rb	HIQ_rb	SIQ_en	HIQ_en
0	0.0773	0.0782	0.9773	0.9771	0.9578	0.9604
5	0.0753	0.0767	0.9757	0.9753	0.9855	0.9875
10	0.0765	0.0764	0.9753	0.9753	0.9917	0.9885
15	0.0764	0.0765	0.9748	0.9752	0.996	0.9962
20	0.0762	0.0766	0.975	0.9752	0.9985	0.9982
25	0.0751	0.0766	0.9758	0.9754	1.0012	0.9967
30	0.0757	0.0762	0.9755	0.9755	1.0025	1.0018
35	0.076	0.0765	0.9755	0.9755	1.0056	0.9954
40	0.0758	0.0769	0.9755	0.9753	1.005	1.0027
45	0.0754	0.0752	0.9759	0.9758	0.9877	0.9977
50	0.0765	0.0758	0.9752	0.9758	1.0024	0.9945
55	0.0762	0.076	0.9752	0.9753	1.0043	1.003
60	0.0754	0.0758	0.9755	0.9753	1.0027	0.9962
65	0.0756	0.0751	0.9752	0.9757	0.9981	0.999
70	0.0763	0.0758	0.9747	0.9752	0.9993	0.996
75	0.0757	0.0756	0.9745	0.9753	0.9962	0.9976
80	0.0764	0.075	0.9746	0.9753	0.9932	0.9935
85	0.0755	0.0745	0.9753	0.9761	0.987	0.9848
90	0.0773	0.0755	0.9773	0.9779	0.9578	0.9625
95	0.0754	0.075	0.9757	0.9762	0.9855	0.9817
100	0.0765	0.0753	0.9753	0.9757	0.9917	0.9895
105	0.0765	0.0756	0.9748	0.9756	0.996	0.9963
110	0.0762	0.0747	0.975	0.9761	0.9985	0.9968
115	0.0751	0.0758	0.9758	0.9758	1.0012	0.9973
120	0.0757	0.0759	0.9755	0.9763	1.0025	0.9972

125	0.0761	0.0751	0.9755	0.9764	1.0056	0.9967
130	0.0759	0.0752	0.9755	0.9761	1.005	1.0018
135	0.0754	0.0756	0.9759	0.9758	0.9877	0.9974
140	0.0765	0.0755	0.9752	0.976	1.0024	0.9954
145	0.0763	0.0761	0.9752	0.9755	1.0043	0.998
150	0.0755	0.0761	0.9755	0.9756	1.0027	0.9934
155	0.0757	0.0752	0.9752	0.9758	0.9981	0.9973
160	0.0764	0.0772	0.9747	0.975	0.9993	0.9924
165	0.0758	0.0766	0.9745	0.9748	0.9962	0.9943
170	0.0765	0.0758	0.9746	0.9754	0.9932	0.9904
175	0.0756	0.077	0.9753	0.9752	0.987	0.9842
180	0.0774	0.0787	0.9773	0.9772	0.9578	0.961
185	0.0755	0.0764	0.9757	0.9756	0.9855	0.9889
190	0.0766	0.0768	0.9753	0.9751	0.9917	0.9913
195	0.0766	0.0763	0.9748	0.9751	0.996	0.9933
200	0.0763	0.0764	0.975	0.9758	0.9985	0.9955
205	0.0752	0.0765	0.9758	0.9758	1.0012	0.9962
210	0.0758	0.076	0.9755	0.9759	1.0025	1.0018
215	0.0761	0.0763	0.9755	0.976	1.0056	0.9914
220	0.0759	0.0768	0.9755	0.9755	1.005	1.0032
225	0.0755	0.0749	0.9759	0.9764	0.9877	1.0005
230	0.0766	0.076	0.9752	0.9757	1.0024	0.9966
235	0.0763	0.0762	0.9752	0.9754	1.0043	1.0036
240	0.0755	0.0756	0.9755	0.9757	1.0027	0.9938
245	0.0757	0.0757	0.9752	0.9757	0.9981	1.0009
250	0.0764	0.0746	0.9747	0.9758	0.9993	1.0007
255	0.0758	0.0758	0.9745	0.9751	0.9962	0.9971
260	0.0764	0.0759	0.9746	0.9753	0.9932	0.9965
265	0.0756	0.0753	0.9753	0.9757	0.987	0.9843
270	0.0774	0.0792	0.9773	0.9771	0.9578	0.964
275	0.0755	0.0751	0.9757	0.9757	0.9855	0.9825
280	0.0766	0.0752	0.9753	0.9759	0.9917	0.9944
285	0.0766	0.0747	0.9748	0.9759	0.996	0.9949
290	0.0763	0.0748	0.975	0.976	0.9985	0.9901
295	0.0752	0.0756	0.9758	0.9758	1.0012	0.9962
300	0.0758	0.0752	0.9755	0.9765	1.0025	1.0007
305	0.0761	0.075	0.9755	0.9766	1.0056	0.9955
310	0.0759	0.0754	0.9755	0.9759	1.005	0.9973
315	0.0755	0.0761	0.9759	0.9756	0.9877	0.9951
320	0.0765	0.0753	0.9752	0.9758	1.0024	0.9961
325	0.0763	0.0756	0.9752	0.9756	1.0043	0.9996
330	0.0754	0.0763	0.9755	0.9752	1.0027	0.9926
335	0.0757	0.076	0.9752	0.9754	0.9981	0.9922
340	0.0763	0.0771	0.9747	0.9746	0.9993	0.9968
345	0.0757	0.0761	0.9745	0.9752	0.9962	0.9913
350	0.0763	0.0753	0.9746	0.9754	0.9932	0.9943

355	0.0756	0.0763	0.9753	0.975	0.987	0.9845
360	0.0773	0.0782	0.9773	0.9771	0.9578	0.9604

B.2 Results for balcony256.bmp image

Rotation in Degrees	SIQ_ed	HIQ_ed	SIQ_rb	HIQ_rb	SIQ_en	HIQ_en
0	0.0627	0.0644	0.9920	0.9923	0.9748	0.9684
5	0.0637	0.0630	0.9883	0.9888	0.9800	0.9805
10	0.0629	0.0621	0.9877	0.9884	0.9878	0.9893
15	0.0619	0.0620	0.9876	0.9881	0.9951	0.9897
20	0.0609	0.0616	0.9878	0.9880	0.9963	0.9961
25	0.0634	0.0621	0.9874	0.9880	0.9950	0.9921
30	0.0623	0.0625	0.9874	0.9882	0.9962	0.9968
35	0.0623	0.0620	0.9873	0.9875	0.9975	0.9939
40	0.0625	0.0617	0.9872	0.9880	1.0008	1.0002
45	0.0626	0.0614	0.9878	0.9878	0.9915	0.9927
50	0.0622	0.0627	0.9873	0.9876	1.0012	0.9978
55	0.0623	0.0623	0.9873	0.9876	1.0003	1.0006
60	0.0623	0.0614	0.9873	0.9880	0.9993	0.9943
65	0.0631	0.0623	0.9875	0.9879	0.9939	0.9965
70	0.0623	0.0615	0.9876	0.9884	0.9980	0.9952
75	0.0627	0.0619	0.9876	0.9885	0.9968	0.9943
80	0.0621	0.0609	0.9878	0.9890	0.9907	0.9896
85	0.0615	0.0610	0.9888	0.9893	0.9856	0.9846
90	0.0627	0.0611	0.9920	0.9930	0.9748	0.9732
95	0.0638	0.0626	0.9883	0.9891	0.9800	0.9856
100	0.0629	0.0625	0.9877	0.9886	0.9878	0.9899
105	0.0620	0.0621	0.9876	0.9883	0.9951	0.9939
110	0.0610	0.0610	0.9878	0.9884	0.9963	0.9930
115	0.0635	0.0617	0.9874	0.9883	0.9950	0.9964
120	0.0625	0.0610	0.9874	0.9882	0.9962	0.9961
125	0.0623	0.0609	0.9873	0.9879	0.9975	0.9914
130	0.0626	0.0619	0.9872	0.9875	1.0008	0.9970
135	0.0626	0.0621	0.9878	0.9879	0.9915	0.9973
140	0.0623	0.0622	0.9873	0.9874	1.0012	0.9954
145	0.0624	0.0635	0.9873	0.9874	1.0003	1.0016
150	0.0624	0.0624	0.9873	0.9870	0.9993	0.9873
155	0.0631	0.0623	0.9875	0.9878	0.9939	0.9976
160	0.0624	0.0626	0.9876	0.9879	0.9980	0.9948
165	0.0628	0.0622	0.9876	0.9883	0.9968	0.9991
170	0.0622	0.0618	0.9878	0.9884	0.9907	0.9912
175	0.0615	0.0624	0.9888	0.9887	0.9856	0.9802
180	0.0628	0.0628	0.9920	0.9925	0.9748	0.9727
185	0.0638	0.0630	0.9883	0.9887	0.9800	0.9824
190	0.0630	0.0618	0.9877	0.9884	0.9878	0.9891
195	0.0621	0.0624	0.9876	0.9880	0.9951	0.9939

200	0.0609	0.0631	0.9878	0.9878	0.9963	0.9915
205	0.0635	0.0629	0.9874	0.9877	0.9950	0.9911
210	0.0624	0.0623	0.9874	0.9883	0.9962	1.0007
215	0.0623	0.0609	0.9873	0.9881	0.9975	0.9942
220	0.0626	0.0621	0.9872	0.9877	1.0008	0.9985
225	0.0626	0.0625	0.9878	0.9876	0.9915	0.9969
230	0.0623	0.0626	0.9873	0.9874	1.0012	0.9935
235	0.0623	0.0622	0.9873	0.9876	1.0003	1.0000
240	0.0624	0.0614	0.9873	0.9879	0.9993	0.9894
245	0.0631	0.0621	0.9875	0.9879	0.9939	0.9956
250	0.0623	0.0614	0.9876	0.9884	0.9980	0.9970
255	0.0628	0.0616	0.9876	0.9885	0.9968	0.9922
260	0.0622	0.0607	0.9878	0.9889	0.9907	0.9919
265	0.0615	0.0612	0.9888	0.9893	0.9856	0.9849
270	0.0627	0.0634	0.9920	0.9924	0.9748	0.9735
275	0.0638	0.0608	0.9883	0.9893	0.9800	0.9795
280	0.0629	0.0611	0.9877	0.9889	0.9878	0.9924
285	0.0620	0.0619	0.9876	0.9883	0.9951	0.9947
290	0.0609	0.0619	0.9878	0.9882	0.9963	0.9905
295	0.0634	0.0622	0.9874	0.9882	0.9950	0.9979
300	0.0623	0.0608	0.9874	0.9879	0.9962	0.9960
305	0.0623	0.0621	0.9873	0.9877	0.9975	0.9917
310	0.0626	0.0616	0.9872	0.9877	1.0008	0.9950
315	0.0626	0.0620	0.9878	0.9879	0.9915	0.9944
320	0.0622	0.0623	0.9873	0.9874	1.0012	0.9930
325	0.0622	0.0618	0.9873	0.9871	1.0003	0.9963
330	0.0623	0.0631	0.9873	0.9870	0.9993	0.9877
335	0.0631	0.0630	0.9875	0.9875	0.9939	0.9964
340	0.0623	0.0614	0.9876	0.9882	0.9980	0.9972
345	0.0627	0.0630	0.9876	0.9881	0.9968	0.9937
350	0.0621	0.0625	0.9878	0.9884	0.9907	0.9897
355	0.0615	0.0632	0.9888	0.9891	0.9856	0.9852
360	0.0627	0.0644	0.9920	0.9923	0.9748	0.9684

B.3 Results for phantom256.bmp image

Rotation in Degrees	SIQ_ed	HIQ_ed	SIQ_rb	HIQ_rb	SIQ_en	HIQ_en
0	0.0937	0.0962	0.8327	0.8351	2.0607	2.0929
5	0.091	0.0927	0.8409	0.8348	1.3841	1.3558
10	0.0921	0.0928	0.8381	0.8413	1.4011	1.3673
15	0.0957	0.0912	0.8337	0.8523	1.3961	1.4075
20	0.0961	0.0913	0.8369	0.8419	1.404	1.3943
25	0.0932	0.0944	0.8465	0.8362	1.3832	1.3942
30	0.0932	0.0933	0.8442	0.8404	1.3782	1.3889
35	0.0951	0.0912	0.8325	0.845	1.3974	1.3867
40	0.0932	0.0958	0.8373	0.8366	1.4122	1.4051
45	0.0911	0.0951	0.8547	0.8358	1.3793	1.4120
50	0.0933	0.0936	0.8373	0.8459	1.3972	1.4158
55	0.0949	0.0929	0.8328	0.8463	1.4132	1.388
60	0.0935	0.0937	0.8436	0.8378	1.3765	1.3763
65	0.0932	0.0945	0.8465	0.8396	1.3932	1.4031
70	0.0961	0.0955	0.8368	0.8429	1.4069	1.421
75	0.0956	0.0972	0.8339	0.8371	1.4002	1.3909
80	0.092	0.0945	0.8384	0.8373	1.398	1.3823
85	0.0909	0.0929	0.8412	0.8459	1.3895	1.3984
90	0.0937	0.097	0.8327	0.8331	2.0607	2.1131
95	0.091	0.0956	0.8409	0.8336	1.3841	1.3967
100	0.0921	0.0948	0.8381	0.8435	1.4018	1.4134
105	0.0956	0.0922	0.8337	0.8451	1.3947	1.4231
110	0.096	0.0935	0.8369	0.84	1.404	1.3697
115	0.0929	0.0952	0.8465	0.8426	1.3839	1.3876
120	0.0932	0.0949	0.8442	0.839	1.3795	1.3700
125	0.0948	0.0941	0.8325	0.8374	1.3984	1.3977
130	0.093	0.0944	0.8373	0.8397	1.4121	1.3714
135	0.091	0.0924	0.8547	0.8397	1.3793	1.3694
140	0.093	0.0906	0.8373	0.8439	1.3968	1.3752
145	0.0947	0.093	0.8328	0.8458	1.4131	1.399
150	0.0933	0.095	0.8436	0.8376	1.3773	1.3791
155	0.093	0.0917	0.8465	0.8417	1.3932	1.3638
160	0.0959	0.0919	0.8368	0.8419	1.4067	1.3688
165	0.0952	0.0924	0.8339	0.8354	1.3998	1.3759
170	0.0917	0.0919	0.8384	0.8424	1.3996	1.3763
175	0.0906	0.0929	0.8412	0.846	1.3895	1.3995
180	0.0934	0.0961	0.8327	0.8296	2.0584	2.1043
185	0.0906	0.0923	0.8409	0.8354	1.3837	1.3695
190	0.0918	0.0913	0.8381	0.8468	1.4018	1.386
195	0.0953	0.09	0.8337	0.8521	1.3958	1.3608
200	0.0958	0.0922	0.8369	0.8376	1.403	1.3626

205	0.0927	0.0946	0.8465	0.8337	1.3839	1.3892
210	0.0932	0.0905	0.8442	0.8418	1.3785	1.3912
215	0.0947	0.0927	0.8325	0.8412	1.3984	1.3687
220	0.0928	0.096	0.8373	0.8291	1.4122	1.3892
225	0.0911	0.0941	0.8547	0.8394	1.3788	1.3964
230	0.0928	0.0927	0.8373	0.8471	1.3961	1.4193
235	0.0945	0.0926	0.8328	0.8434	1.4132	1.3911
240	0.0936	0.0933	0.8436	0.8401	1.3757	1.3847
245	0.0927	0.0945	0.8465	0.8407	1.393	1.4023
250	0.0959	0.0957	0.8368	0.8395	1.4065	1.4171
255	0.0952	0.0964	0.8339	0.8362	1.3993	1.3899
260	0.0916	0.0931	0.8384	0.8348	1.398	1.3907
265	0.0905	0.0937	0.8412	0.8434	1.3896	1.3773
270	0.0934	0.0971	0.8327	0.8325	2.0557	2.1257
275	0.0907	0.094	0.8409	0.8345	1.3841	1.399
280	0.0918	0.0955	0.8381	0.8403	1.4022	1.403
285	0.0953	0.0922	0.8337	0.8372	1.3955	1.3969
290	0.0959	0.0929	0.8369	0.8398	1.4047	1.3418
295	0.0929	0.0946	0.8465	0.8508	1.3839	1.3934
300	0.093	0.0946	0.8442	0.8407	1.3785	1.3739
305	0.0949	0.094	0.8325	0.8304	1.3977	1.4071
310	0.093	0.0945	0.8373	0.8381	1.4121	1.396
315	0.091	0.0921	0.8547	0.8446	1.3789	1.3826
320	0.093	0.0915	0.8373	0.8439	1.397	1.3708
325	0.0946	0.0935	0.8328	0.8498	1.4132	1.3813
330	0.0935	0.0935	0.8436	0.8428	1.3767	1.3891
335	0.0929	0.0941	0.8465	0.8395	1.3931	1.3864
340	0.096	0.0921	0.8368	0.8452	1.4061	1.3671
345	0.0955	0.0916	0.8339	0.8391	1.4009	1.3696
350	0.0919	0.0917	0.8384	0.8391	1.3995	1.3552
355	0.0908	0.0928	0.8412	0.8466	1.3895	1.4009
360	0.0937	0.0962	0.8327	0.8351	2.0527	2.087

B.4 Results for rand256.bmp image

Rotation in Degrees	SIQ_ed	HIQ_ed	SIQ_rb	HIQ_rb	SIQ_en	HIQ_en
0	0.1984	0.1983	0.8683	0.8703	0.6535	0.6685
5	0.1612	0.1613	0.9028	0.9044	0.6949	0.7063
10	0.1608	0.1609	0.9029	0.9043	0.7161	0.7266
15	0.1613	0.1613	0.9024	0.9037	0.7262	0.7276
20	0.1612	0.1613	0.9026	0.9036	0.7247	0.7310
25	0.1610	0.1612	0.9027	0.9035	0.7286	0.7252
30	0.1608	0.1607	0.9028	0.9043	0.7365	0.7375
35	0.1614	0.1612	0.9025	0.9041	0.7332	0.7378
40	0.1615	0.1616	0.9024	0.9036	0.7325	0.7374
45	0.1614	0.1616	0.9032	0.9038	0.7220	0.7363
50	0.1611	0.1612	0.9028	0.9036	0.7385	0.7327
55	0.1610	0.1611	0.9029	0.9035	0.7395	0.7402
60	0.1614	0.1614	0.9023	0.9034	0.7311	0.7396
65	0.1611	0.1611	0.9023	0.9032	0.7263	0.7355
70	0.1613	0.1611	0.9022	0.9034	0.7245	0.7372
75	0.1612	0.1612	0.9020	0.9032	0.7191	0.7252
80	0.1610	0.1609	0.9023	0.9038	0.7117	0.7198
85	0.1609	0.1611	0.9029	0.9042	0.7058	0.7063
90	0.1984	0.1984	0.8683	0.8704	0.6535	0.6739
95	0.1612	0.1611	0.9028	0.9044	0.6949	0.7119
100	0.1608	0.1611	0.9029	0.9037	0.7161	0.7153
105	0.1612	0.1613	0.9024	0.9037	0.7262	0.7194
110	0.1612	0.1613	0.9026	0.9037	0.7247	0.7275
115	0.1610	0.1610	0.9027	0.9040	0.7286	0.7338
120	0.1608	0.1608	0.9028	0.9039	0.7365	0.7371
125	0.1613	0.1612	0.9025	0.9037	0.7332	0.7324
130	0.1615	0.1614	0.9024	0.9036	0.7325	0.7398
135	0.1614	0.1613	0.9032	0.9037	0.7220	0.7393
140	0.1611	0.1609	0.9028	0.9039	0.7385	0.7373
145	0.1610	0.1612	0.9029	0.9035	0.7395	0.7356
150	0.1614	0.1614	0.9023	0.9034	0.7311	0.7359
155	0.1611	0.1610	0.9023	0.9034	0.7263	0.7376
160	0.1613	0.1612	0.9022	0.9032	0.7245	0.7301
165	0.1611	0.1611	0.9020	0.9035	0.7191	0.7285
170	0.1610	0.1609	0.9023	0.9038	0.7117	0.7205
175	0.1609	0.1610	0.9029	0.9043	0.7058	0.7037
180	0.1985	0.1984	0.8683	0.8704	0.6535	0.6710
185	0.1612	0.1611	0.9028	0.9044	0.6949	0.7111
190	0.1608	0.1609	0.9029	0.9041	0.7161	0.7188
195	0.1612	0.1613	0.9024	0.9035	0.7262	0.7230
200	0.1612	0.1611	0.9026	0.9039	0.7247	0.7297

205	0.1610	0.1612	0.9027	0.9037	0.7286	0.7266
210	0.1609	0.1608	0.9028	0.9042	0.7365	0.7377
215	0.1613	0.1613	0.9025	0.9039	0.7332	0.7317
220	0.1615	0.1615	0.9024	0.9035	0.7325	0.7407
225	0.1613	0.1614	0.9032	0.9038	0.7220	0.7393
230	0.1611	0.1611	0.9028	0.9038	0.7385	0.7331
235	0.1610	0.1610	0.9029	0.9038	0.7395	0.7386
240	0.1614	0.1614	0.9023	0.9033	0.7311	0.7324
245	0.1611	0.1610	0.9023	0.9032	0.7263	0.7397
250	0.1613	0.1614	0.9022	0.9032	0.7245	0.7306
255	0.1611	0.1612	0.9020	0.9033	0.7191	0.7234
260	0.1610	0.1609	0.9023	0.9037	0.7117	0.7200
265	0.1608	0.1609	0.9029	0.9044	0.7058	0.7072
270	0.1984	0.1984	0.8683	0.8704	0.6535	0.6703
275	0.1612	0.1613	0.9028	0.9042	0.6949	0.7045
280	0.1608	0.1610	0.9029	0.9036	0.7161	0.7173
285	0.1612	0.1614	0.9024	0.9036	0.7262	0.7164
290	0.1612	0.1612	0.9026	0.9038	0.7247	0.7328
295	0.1610	0.1611	0.9027	0.9038	0.7286	0.7297
300	0.1608	0.1608	0.9028	0.9041	0.7365	0.7330
305	0.1613	0.1611	0.9025	0.9039	0.7332	0.7358
310	0.1615	0.1614	0.9024	0.9037	0.7325	0.7353
315	0.1613	0.1614	0.9032	0.9035	0.7220	0.7367
320	0.1611	0.1608	0.9028	0.9040	0.7385	0.7469
325	0.1610	0.1611	0.9029	0.9039	0.7395	0.7434
330	0.1614	0.1613	0.9023	0.9036	0.7311	0.7362
335	0.1612	0.1611	0.9023	0.9031	0.7263	0.7354
340	0.1614	0.1613	0.9022	0.9031	0.7245	0.7258
345	0.1612	0.1612	0.9020	0.9035	0.7191	0.7248
350	0.1610	0.1610	0.9023	0.9037	0.7117	0.7175
355	0.1608	0.1609	0.9029	0.9043	0.7058	0.7075
360	0.1984	0.1983	0.8683	0.8703	0.6535	0.6685

B.5 Results for square256.bmp image

Rotation in Degrees	SIQ_ed	HIQ_ed	SIQ_rb	HIQ_rb	SIQ_en	HIQ_en
0	0.0000	0.0065	1.0000	0.9999	1.0000	1.0227
5	0.0062	0.0062	0.9944	0.9947	1.0173	1.0198
10	0.0062	0.0062	0.9939	0.9944	1.0177	1.0119
15	0.0062	0.0061	0.9938	0.9943	1.0317	0.9936
20	0.0062	0.0061	0.9940	0.9944	1.0234	1.0257
25	0.0062	0.0061	0.9940	0.9944	1.0334	1.0150
30	0.0062	0.0061	0.9940	0.9946	1.0357	1.0335
35	0.0062	0.0061	0.9941	0.9949	1.0183	1.0320
40	0.0062	0.0060	0.9941	0.9944	1.0432	1.0371
45	0.0062	0.0060	0.9945	0.9943	1.0390	1.0383
50	0.0062	0.0060	0.9941	0.9943	1.0406	1.0338
55	0.0062	0.0059	0.9941	0.9939	1.0262	1.0290
60	0.0062	0.0059	0.9940	0.9941	1.0380	1.0315
65	0.0062	0.0060	0.9940	0.9942	1.0337	1.0342
70	0.0062	0.0060	0.9940	0.9943	1.0361	1.0215
75	0.0062	0.0060	0.9938	0.9943	1.0326	1.0276
80	0.0062	0.0061	0.9939	0.9944	0.9577	0.9912
85	0.0062	0.0061	0.9944	0.9949	1.0195	1.0151
90	0.0000	0.0064	1.0000	0.9999	1.0000	1.0239
95	0.0062	0.0061	0.9944	0.9950	1.0173	1.0179
100	0.0062	0.0062	0.9939	0.9949	1.0177	1.0231
105	0.0062	0.0062	0.9938	0.9945	1.0317	1.0221
110	0.0062	0.0062	0.9940	0.9946	1.0234	0.8741
115	0.0062	0.0063	0.9940	0.9946	1.0334	1.0303
120	0.0062	0.0063	0.9940	0.9947	1.0357	1.0298
125	0.0062	0.0064	0.9941	0.9947	1.0183	1.0239
130	0.0062	0.0063	0.9941	0.9946	1.0432	1.0372
135	0.0062	0.0063	0.9945	0.9944	1.0390	1.0365
140	0.0062	0.0063	0.9941	0.9946	1.0406	1.0342
145	0.0062	0.0062	0.9941	0.9937	1.0262	1.0379
150	0.0062	0.0062	0.9940	0.9941	1.0380	1.0311
155	0.0062	0.0063	0.9940	0.9944	1.0337	0.9909
160	0.0062	0.0063	0.9940	0.9946	1.0361	1.0330
165	0.0062	0.0063	0.9938	0.9945	1.0326	1.0262
170	0.0062	0.0063	0.9939	0.9946	0.9577	1.0235
175	0.0062	0.0062	0.9944	0.9954	1.0195	0.9345
180	0.0000	0.0065	1.0000	0.9999	1.0000	1.0228
185	0.0062	0.0062	0.9944	0.9952	1.0173	1.0207
190	0.0062	0.0062	0.9939	0.9946	1.0177	1.0138
195	0.0062	0.0061	0.9938	0.9945	1.0317	0.9931
200	0.0062	0.0061	0.9940	0.9945	1.0234	1.0222

205	0.0062	0.0061	0.9940	0.9947	1.0334	1.0152
210	0.0062	0.0061	0.9940	0.9950	1.0357	1.0345
215	0.0062	0.0061	0.9941	0.9944	1.0183	1.0308
220	0.0062	0.0060	0.9941	0.9946	1.0432	1.0371
225	0.0062	0.0060	0.9945	0.9944	1.0390	1.0394
230	0.0062	0.0060	0.9941	0.9943	1.0406	1.0341
235	0.0062	0.0059	0.9941	0.9944	1.0262	1.0301
240	0.0062	0.0059	0.9940	0.9938	1.0380	1.0301
245	0.0062	0.0060	0.9940	0.9943	1.0337	1.0368
250	0.0062	0.0060	0.9940	0.9945	1.0361	1.0241
255	0.0062	0.0060	0.9938	0.9944	1.0326	1.0264
260	0.0062	0.0061	0.9939	0.9945	0.9577	0.9892
265	0.0062	0.0061	0.9944	0.9949	1.0195	1.0145
270	0.0000	0.0064	1.0000	0.9999	1.0000	1.0240
275	0.0062	0.0061	0.9944	0.9950	1.0173	1.0174
280	0.0062	0.0062	0.9939	0.9945	1.0177	1.0229
285	0.0062	0.0062	0.9938	0.9945	1.0317	1.0207
290	0.0062	0.0062	0.9940	0.9945	1.0234	0.8693
295	0.0062	0.0063	0.9940	0.9945	1.0334	1.0302
300	0.0062	0.0063	0.9940	0.9947	1.0357	1.0311
305	0.0062	0.0064	0.9941	0.9948	1.0183	1.0235
310	0.0062	0.0063	0.9941	0.9942	1.0432	1.0357
315	0.0062	0.0063	0.9945	0.9944	1.0390	1.0361
320	0.0062	0.0063	0.9941	0.9941	1.0406	1.0329
325	0.0062	0.0062	0.9941	0.9943	1.0262	1.0397
330	0.0062	0.0062	0.9940	0.9939	1.0380	1.0298
335	0.0062	0.0063	0.9940	0.9943	1.0337	0.9900
340	0.0062	0.0063	0.9940	0.9943	1.0361	1.0324
345	0.0062	0.0063	0.9938	0.9943	1.0326	1.0266
350	0.0062	0.0063	0.9939	0.9944	0.9577	1.0221
355	0.0062	0.0062	0.9944	0.9947	1.0195	0.9377
360	0.0000	0.0065	1.0000	0.9999	1.0000	1.0227

B.6 Results for hexagon256.bmp image

Rotation in Degrees	SIQ_ed	HIQ_ed	SIQ_rb	HIQ_rb	SIQ_en	HIQ_en
0	0.1657	0.0000	0.9363	1.0000	2.7411	1.0000
5	0.1581	0.1565	0.9374	0.9374	1.2747	1.2725
10	0.1582	0.1567	0.9378	0.9394	1.2785	1.2849
15	0.1583	0.1579	0.9371	0.9386	1.2759	1.2860
20	0.1578	0.1578	0.9376	0.9387	1.2792	1.2792
25	0.1572	0.1579	0.9371	0.9389	1.2770	1.2714
30	0.1564	0.1581	0.9368	0.9387	1.2745	1.2788
35	0.1562	0.1577	0.9373	0.9396	1.2769	1.2714
40	0.1579	0.1569	0.9373	0.9392	1.2745	1.2802
45	0.1581	0.1577	0.9379	0.9388	1.2641	1.2831
50	0.1586	0.1572	0.9378	0.9388	1.2757	1.2677
55	0.1604	0.1590	0.9378	0.9377	1.2808	1.2590
60	0.1601	0.1577	0.9383	0.9389	1.2772	1.2744
65	0.1595	0.1671	0.9374	0.9316	1.2720	1.2396
70	0.1593	0.1564	0.9376	0.9389	1.2754	1.2775
75	0.1584	0.1576	0.9375	0.9386	1.2829	1.2619
80	0.1588	0.1569	0.9375	0.9393	1.2724	1.2837
85	0.1583	0.1568	0.9374	0.9393	1.2703	1.2803
90	0.1657	0.1634	0.9363	0.9378	2.7411	3.1840
95	0.1582	0.1569	0.9374	0.9392	1.2747	1.2790
100	0.1581	0.1569	0.9378	0.9393	1.2785	1.2826
105	0.1583	0.1583	0.9371	0.9383	1.2759	1.2680
110	0.1577	0.1563	0.9376	0.9388	1.2792	1.2683
115	0.1573	0.1703	0.9371	0.9291	1.2770	1.2323
120	0.1565	0.1582	0.9368	0.9395	1.2745	1.2765
125	0.1562	0.1563	0.9373	0.9399	1.2769	1.2602
130	0.1578	0.1571	0.9373	0.9392	1.2745	1.2728
135	0.1581	0.1568	0.9379	0.9390	1.2641	1.2781
140	0.1586	0.1585	0.9378	0.9391	1.2757	1.2710
145	0.1602	0.1571	0.9378	0.9385	1.2808	1.2809
150	0.1600	0.1573	0.9383	0.9389	1.2772	1.2758
155	0.1594	0.1575	0.9374	0.9383	1.2720	1.2768
160	0.1592	0.1576	0.9376	0.9389	1.2754	1.2818
165	0.1583	0.1582	0.9375	0.9387	1.2829	1.2803
170	0.1586	0.1572	0.9375	0.9385	1.2724	1.2815
175	0.1581	0.1575	0.9374	0.9384	1.2703	1.2829
180	0.1657	0.1474	0.9363	0.9491	2.7411	1.0875
185	0.1580	0.1577	0.9374	0.9387	1.2747	1.2782
190	0.1581	0.1572	0.9378	0.9388	1.2785	1.2852
195	0.1583	0.1574	0.9371	0.9388	1.2759	1.2793
200	0.1576	0.1574	0.9376	0.9386	1.2792	1.2790

205	0.1572	0.1589	0.9371	0.9391	1.2770	1.2746
210	0.1564	0.1580	0.9368	0.9391	1.2745	1.2821
215	0.1560	0.1575	0.9373	0.9394	1.2769	1.2700
220	0.1576	0.1568	0.9373	0.9391	1.2745	1.2775
225	0.1580	0.1573	0.9379	0.9393	1.2641	1.2791
230	0.1585	0.1573	0.9378	0.9388	1.2757	1.2563
235	0.1601	0.1557	0.9378	0.9405	1.2808	1.2679
240	0.1600	0.1579	0.9383	0.9391	1.2772	1.2716
245	0.1594	0.1701	0.9374	0.9290	1.2720	1.2377
250	0.1592	0.1566	0.9376	0.9391	1.2754	1.2751
255	0.1584	0.1566	0.9375	0.9391	1.2829	1.2742
260	0.1586	0.1566	0.9375	0.9392	1.2724	1.2836
265	0.1581	0.1569	0.9374	0.9397	1.2703	1.2785
270	0.1657	0.1635	0.9363	0.9377	2.7411	3.1347
275	0.1579	0.1566	0.9374	0.9394	1.2747	1.2810
280	0.1582	0.1569	0.9378	0.9390	1.2785	1.2851
285	0.1583	0.1577	0.9371	0.9387	1.2759	1.2614
290	0.1578	0.1567	0.9376	0.9391	1.2792	1.2740
295	0.1571	0.1652	0.9371	0.9331	1.2770	1.2346
300	0.1564	0.1576	0.9368	0.9395	1.2745	1.2767
305	0.1561	0.1588	0.9373	0.9379	1.2769	1.2498
310	0.1577	0.1570	0.9373	0.9390	1.2745	1.2640
315	0.1580	0.1573	0.9379	0.9391	1.2641	1.2816
320	0.1586	0.1581	0.9378	0.9389	1.2757	1.2684
325	0.1602	0.1580	0.9378	0.9387	1.2808	1.2851
330	0.1601	0.1579	0.9383	0.9383	1.2772	1.2738
335	0.1594	0.1581	0.9374	0.9382	1.2720	1.2719
340	0.1592	0.1574	0.9376	0.9387	1.2754	1.2815
345	0.1585	0.1574	0.9375	0.9384	1.2829	1.2865
350	0.1587	0.1566	0.9375	0.9391	1.2724	1.2784
355	0.1583	0.1565	0.9374	0.9373	1.2703	1.2746
360	0.1657	0.0000	0.9363	1.0000	2.7411	1.0000

B.7 Results for sinewave01_256.bmp image

Rotation in Degrees	SIQ_ed	HIQ_ed	SIQ_rb	HIQ_rb	SIQ_en	HIQ_en
0	0.0128	0.0120	0.9995	0.9996	0.7272	0.7712
5	0.0129	0.0120	0.9954	0.9959	1.0121	1.0130
10	0.0129	0.0121	0.9951	0.9955	1.0091	0.9851
15	0.0129	0.0121	0.9951	0.9954	1.0197	1.0175
20	0.0129	0.0122	0.9952	0.9955	0.9961	1.0144
25	0.0129	0.0123	0.9952	0.9957	1.0136	0.9986
30	0.0129	0.0124	0.9953	0.9962	1.0237	1.0188
35	0.0129	0.0125	0.9954	0.9956	1.0024	1.0125
40	0.0129	0.0126	0.9954	0.9957	1.0198	1.0109
45	0.0129	0.0127	0.9957	0.9958	0.7873	1.0245
50	0.0129	0.0128	0.9954	0.9955	1.0198	1.0184
55	0.0129	0.0130	0.9954	0.9955	1.0024	1.0211
60	0.0129	0.0131	0.9953	0.9953	1.0237	0.9999
65	0.0129	0.0132	0.9952	0.9955	1.0136	1.0119
70	0.0129	0.0132	0.9952	0.9954	0.9961	1.0118
75	0.0129	0.0133	0.9951	0.9954	1.0197	0.9852
80	0.0129	0.0134	0.9951	0.9955	1.0091	1.0159
85	0.0129	0.0134	0.9954	0.9955	1.0121	1.0112
90	0.0128	0.0134	0.9995	0.9994	0.7272	0.8620
95	0.0129	0.0134	0.9954	0.9959	1.0121	1.0088
100	0.0129	0.0134	0.9951	0.9955	1.0091	1.0146
105	0.0129	0.0133	0.9951	0.9955	1.0197	0.9977
110	0.0129	0.0133	0.9952	0.9956	0.9961	1.0122
115	0.0129	0.0132	0.9952	0.9956	1.0136	1.0011
120	0.0129	0.0131	0.9953	0.9958	1.0237	0.9997
125	0.0129	0.0130	0.9954	0.9958	1.0024	0.9947
130	0.0129	0.0129	0.9954	0.9956	1.0198	1.0223
135	0.0129	0.0127	0.9957	0.9957	0.7873	1.0251
140	0.0129	0.0126	0.9954	0.9955	1.0198	0.9997
145	0.0129	0.0125	0.9954	0.9952	1.0024	1.0150
150	0.0129	0.0124	0.9953	0.9955	1.0237	1.0191
155	0.0129	0.0123	0.9952	0.9954	1.0136	0.9984
160	0.0129	0.0122	0.9952	0.9954	0.9961	1.0189
165	0.0129	0.0121	0.9951	0.9954	1.0197	1.0154
170	0.0129	0.0121	0.9951	0.9954	1.0091	1.0111
175	0.0129	0.0120	0.9954	0.9958	1.0121	1.0127
180	0.0128	0.0120	0.9995	0.9996	0.7272	0.7746
185	0.0129	0.0120	0.9954	0.9957	1.0121	1.0124
190	0.0129	0.0121	0.9951	0.9956	1.0091	1.0140
195	0.0129	0.0121	0.9951	0.9955	1.0197	1.0136
200	0.0129	0.0122	0.9952	0.9956	0.9961	1.0176

205	0.0129	0.0123	0.9952	0.9957	1.0136	1.0027
210	0.0129	0.0124	0.9953	0.9956	1.0237	1.0177
215	0.0129	0.0125	0.9954	0.9960	1.0024	1.0108
220	0.0129	0.0126	0.9954	0.9959	1.0198	1.0100
225	0.0129	0.0127	0.9957	0.9956	0.7873	1.0337
230	0.0129	0.0128	0.9954	0.9958	1.0198	1.0181
235	0.0129	0.0130	0.9954	0.9953	1.0024	1.0154
240	0.0129	0.0131	0.9953	0.9954	1.0237	1.0019
245	0.0129	0.0132	0.9952	0.9955	1.0136	1.0022
250	0.0129	0.0132	0.9952	0.9956	0.9961	1.0191
255	0.0129	0.0133	0.9951	0.9954	1.0197	0.9889
260	0.0129	0.0134	0.9951	0.9955	1.0091	1.0130
265	0.0129	0.0134	0.9954	0.9959	1.0121	1.0069
270	0.0128	0.0134	0.9995	0.9994	0.7272	0.8635
275	0.0129	0.0134	0.9954	0.9955	1.0121	1.0091
280	0.0129	0.0134	0.9951	0.9957	1.0091	1.0141
285	0.0129	0.0133	0.9951	0.9955	1.0197	0.9988
290	0.0129	0.0133	0.9952	0.9956	0.9961	1.0180
295	0.0129	0.0132	0.9952	0.9958	1.0136	1.0042
300	0.0129	0.0131	0.9953	0.9959	1.0237	1.0007
305	0.0129	0.0130	0.9954	0.9959	1.0024	0.9889
310	0.0129	0.0129	0.9954	0.9958	1.0198	1.0215
315	0.0129	0.0127	0.9957	0.9957	0.7873	1.0345
320	0.0129	0.0126	0.9954	0.9957	1.0198	1.0175
325	0.0129	0.0125	0.9954	0.9958	1.0024	1.0124
330	0.0129	0.0124	0.9953	0.9950	1.0237	1.0170
335	0.0129	0.0123	0.9952	0.9955	1.0136	1.0043
340	0.0129	0.0122	0.9952	0.9956	0.9961	1.0192
345	0.0129	0.0121	0.9951	0.9955	1.0197	1.0179
350	0.0129	0.0121	0.9951	0.9955	1.0091	0.9845
355	0.0129	0.0120	0.9954	0.9957	1.0121	1.0124
360	0.0128	0.0120	0.9995	0.9996	0.7272	0.7712

B.8 Results for sinewave10_256.bmp image

Rotation in Degrees	SIQ_ed	HIQ_ed	SIQ_rb	HIQ_rb	SIQ_en	HIQ_en
0	0.1292	0.1211	0.9518	0.9593	0.6791	0.7634
5	0.1293	0.1213	0.9475	0.9547	0.9923	0.9950
10	0.1292	0.1210	0.9474	0.9541	0.9737	1.0202
15	0.1288	0.1217	0.9474	0.9534	0.9975	1.0159
20	0.1289	0.1223	0.9476	0.9533	1.0147	1.0230
25	0.1285	0.1231	0.9477	0.9528	0.9796	1.0188
30	0.1286	0.1240	0.9478	0.9520	0.9955	1.0153
35	0.1283	0.1250	0.9480	0.9514	0.9522	1.0254
40	0.1283	0.1261	0.9482	0.9501	0.9982	0.9677
45	0.1284	0.1271	0.9484	0.9495	0.7945	1.0502
50	0.1283	0.1283	0.9482	0.9490	0.9982	1.0164
55	0.1283	0.1294	0.9480	0.9477	0.9522	1.0198
60	0.1286	0.1304	0.9478	0.9470	0.9955	1.0046
65	0.1285	0.1311	0.9477	0.9464	0.9796	1.0093
70	0.1289	0.1318	0.9476	0.9462	1.0147	1.0134
75	0.1288	0.1326	0.9474	0.9456	0.9975	0.9634
80	0.1292	0.1330	0.9474	0.9452	0.9737	1.0068
85	0.1293	0.1332	0.9475	0.9452	0.9923	0.9948
90	0.1292	0.1334	0.9518	0.9502	0.6791	0.9295
95	0.1293	0.1334	0.9475	0.9453	0.9923	0.9919
100	0.1292	0.1332	0.9474	0.9454	0.9737	1.0033
105	0.1288	0.1324	0.9474	0.9456	0.9975	0.9958
110	0.1289	0.1322	0.9476	0.9457	1.0147	1.0133
115	0.1284	0.1313	0.9477	0.9468	0.9796	1.0117
120	0.1286	0.1303	0.9478	0.9472	0.9955	1.0132
125	0.1282	0.1295	0.9480	0.9481	0.9522	1.0117
130	0.1282	0.1284	0.9482	0.9489	0.9982	1.0116
135	0.1284	0.1275	0.9484	0.9496	0.7945	1.0474
140	0.1282	0.1262	0.9482	0.9502	0.9982	1.0040
145	0.1282	0.1251	0.9480	0.9508	0.9522	1.0230
150	0.1286	0.1243	0.9478	0.9516	0.9955	1.0166
155	0.1284	0.1233	0.9477	0.9522	0.9796	1.0201
160	0.1289	0.1224	0.9476	0.9531	1.0147	1.0301
165	0.1288	0.1219	0.9474	0.9535	0.9975	1.0134
170	0.1292	0.1210	0.9474	0.9540	0.9737	1.0198
175	0.1293	0.1218	0.9475	0.9547	0.9923	0.9793
180	0.1292	0.1214	0.9518	0.9586	0.6791	0.8216
185	0.1293	0.1218	0.9475	0.9549	0.9923	0.9823
190	0.1292	0.1207	0.9474	0.9540	0.9737	1.0219
195	0.1288	0.1218	0.9474	0.9536	0.9975	1.0191
200	0.1289	0.1223	0.9476	0.9533	1.0147	1.0261

205	0.1285	0.1230	0.9477	0.9529	0.9796	1.0210
210	0.1286	0.1241	0.9478	0.9518	0.9955	1.0091
215	0.1283	0.1250	0.9480	0.9513	0.9522	1.0260
220	0.1283	0.1261	0.9482	0.9505	0.9982	0.9832
225	0.1284	0.1272	0.9484	0.9495	0.7945	1.0447
230	0.1283	0.1282	0.9482	0.9484	0.9982	1.0113
235	0.1283	0.1294	0.9480	0.9476	0.9522	1.0190
240	0.1286	0.1304	0.9478	0.9467	0.9955	1.0085
245	0.1285	0.1311	0.9477	0.9464	0.9796	1.0137
250	0.1289	0.1319	0.9476	0.9461	1.0147	1.0175
255	0.1288	0.1327	0.9474	0.9456	0.9975	0.9479
260	0.1292	0.1329	0.9474	0.9453	0.9737	1.0023
265	0.1293	0.1332	0.9475	0.9455	0.9923	0.9950
270	0.1292	0.1334	0.9518	0.9501	0.6791	0.9136
275	0.1293	0.1334	0.9475	0.9454	0.9923	0.9939
280	0.1292	0.1331	0.9474	0.9453	0.9737	1.0041
285	0.1288	0.1324	0.9474	0.9455	0.9975	0.9595
290	0.1288	0.1322	0.9476	0.9458	1.0147	1.0108
295	0.1284	0.1312	0.9477	0.9466	0.9796	1.0018
300	0.1286	0.1303	0.9478	0.9475	0.9955	1.0139
305	0.1282	0.1295	0.9480	0.9474	0.9522	1.0175
310	0.1282	0.1284	0.9482	0.9486	0.9982	1.0163
315	0.1284	0.1274	0.9484	0.9494	0.7945	1.0453
320	0.1282	0.1262	0.9482	0.9503	0.9982	0.9770
325	0.1282	0.1252	0.9480	0.9508	0.9522	1.0303
330	0.1286	0.1242	0.9478	0.9515	0.9955	0.9910
335	0.1284	0.1233	0.9477	0.9523	0.9796	1.0292
340	0.1288	0.1223	0.9476	0.9531	1.0147	1.0296
345	0.1288	0.1218	0.9474	0.9539	0.9975	1.0094
350	0.1292	0.1212	0.9474	0.9542	0.9737	1.0190
355	0.1293	0.1212	0.9475	0.9546	0.9923	0.9979
360	0.1292	0.1211	0.9518	0.9593	0.6791	0.7634

B.9 MTF Results

Frequency (No of cycles across the width of the image)	SIQ_mtf	HIQ_mtf
0	1.0000	1.0000
2	0.9874	0.9893
4	0.9504	0.9577
6	0.8912	0.9060
8	0.8132	0.8294
10	0.7209	0.7960
12	0.6195	0.6950
14	0.5142	0.6075
16	0.4105	0.5349
18	0.3131	0.7606
20	0.2259	0.2580
22	0.1519	0.2174
24	0.0927	0.1650
26	0.0490	0.1189
28	0.0202	0.0856
30	0.0046	0.0581
32	0.0000	0.0409
34	0.0036	0.0313
36	0.0125	0.0283
38	0.0239	0.0266
40	0.0352	0.0292
42	0.0445	0.0325
44	0.0505	0.0353
46	0.0525	0.0434
48	0.0506	0.0403
50	0.0453	0.0400
52	0.0376	0.0386
54	0.0285	0.0358
56	0.0194	0.0351
58	0.0113	0.0319
60	0.0051	0.0291
62	0.0013	0.0268
64	0.0000	0.0317
66	0.0011	0.0234
68	0.0042	0.0224
70	0.0084	0.0217
72	0.0131	0.0213
74	0.0174	0.0206
76	0.0207	0.0204
78	0.0225	0.0202
80	0.0226	0.0199

82	0.0211	0.0202
84	0.0181	0.0191
86	0.0143	0.0190
88	0.0100	0.0189
90	0.0060	0.0190
92	0.0028	0.0194
94	0.0007	0.0195
96	0.0000	0.0199
98	0.0007	0.0203
100	0.0026	0.0208
102	0.0053	0.0211
104	0.0085	0.0215
106	0.0116	0.0218
108	0.0142	0.0221
110	0.0158	0.0217
112	0.0162	0.0224
114	0.0155	0.0227
116	0.0136	0.0231
118	0.0110	0.0252
120	0.0079	0.0206
122	0.0048	0.0220
124	0.0023	0.0223
126	0.0006	0.0225
128	0.0000	0.0137

Appendix C

Tabular Results of *ART Experiments

This appendix contains the tabular results of the *ART experiment. Note that a large number of experiments were done to arrive the *ART results. However only the ones whose graphs are shown and discussed in chapter 4 are provided here.

C.1 Comparison of ART, MART, SIRT and SART

Test Image	Normalized Euclidean distance at convergence				Number of Iterations to Converge			
	ART	MART	SIRT	SART	ART	MART	SIRT	SART
A	0.0685	0.2197	0.0660	0.0542	11	3	250	250
B	0.0574	0.3463	0.0654	0.0565	10	2	183	250
C	0.0746	0.1223	0.0895	0.0688	35	3	250	250
D	0.2078	0.5006	0.2127	0.1805	9	3	233	250
E	0.3919	0.4645	0.3929	0.3669	12	2	250	250
F	0.0064	0.2918	0.0193	0.0184	27	6	250	250
G	0.1143	0.2733	0.1173	0.1010	9	1	210	250
H	0.0066	0.2661	0.0263	0.0236	23	7	250	250
I	0.2169	0.3395	0.2194	0.1866	9	3	206	250

C.2 Comparison of Seed Images in *ART Reconstruction

Test Image	Seed Image (Algorithm: MART)		
	flat	meshgrid	FBP
A	0.1393	0.1084	0.0789
B	0.1911	0.1291	0.1003
C	0.0975	0.0946	0.0885
D	0.3862	0.3343	0.2841
E	0.4418	0.4349	0.4227
F	0.1547	0.0898	0.0779
G	0.1986	0.1686	0.1424
H	0.1798	0.1341	0.1156
I	0.2969	0.2730	0.2518

C.3 Comparison of the different Projection Angle Ordering Schemes in *ART

Test Image	Projection Angle Ordering Schemes				
	Sequential Access Scheme (SAS)	Fixed Angle Access Scheme (FAAS-90)	Random Access Scheme (RAS)	Multi-level Access Scheme (MLSAS)	Weighted Distance Scheme (WDAS)
A	0.1559	0.1393	0.1453	0.1248	0.1255
B	0.2269	0.1911	0.1822	0.1589	0.1579
C	0.0991	0.0975	0.1011	0.0944	0.0950
D	0.4194	0.3862	0.4187	0.3635	0.3536
E	0.4457	0.4418	0.4553	0.4419	0.4376
F	0.1829	0.1547	0.1409	0.1221	0.1210
G	0.2153	0.1986	0.1903	0.1842	0.1818
H	0.2040	0.1798	0.1689	0.1552	0.1577
I	0.3102	0.2969	0.2913	0.2851	0.2827

C.4 Comparison of the different Weighting Schemes in *ART

Test Image	Weighting Schemes			
	Binary Scheme (SAS)	Length of ray within Pixel Scheme (INT)	Distance of Pixel-Center from Center of Ray (DIST)	Distance of center of pixel from farthest corner of adjacent Ray Scheme (CONT)
A	0.0735	0.0782	0.0789	0.0661
B	0.0917	0.0947	0.1003	0.0825
C	0.0622	0.0764	0.0885	0.0560
D	0.2724	0.2849	0.2841	0.2451
E	0.3730	0.4330	0.4227	0.3357
F	0.0723	0.0798	0.0779	0.0651
G	0.1349	0.1408	0.1424	0.1214
H	0.1062	0.1147	0.1156	0.0956
I	0.2347	0.2527	0.2518	0.2112

References

[1] A. H. Andersen, A. C. Kak, "*Simultaneous algebraic reconstruction technique (SART): A superior implementation of the art algorithm*", Ultrasonic. Imaging, vol. 6, pp. 81-94, Jan 1984.

[2] A. Macovski, "*Medical Imaging Systems*", Englewood Cliff, Prentice-Hall, New Jersey (1983).

[3] A. Macovski, "*Physical problems of Computerized Tomography*", Proceedings IEEE 71:373-378 (1983).

[4] A.C. Kak, M. Slaney, "*Principles of Computerized Tomographic Imaging*", IEEE Press, 1988.

[5] C. Hamaker, D.C. Solmon, "*The angles between the null spaces of X rays*", J. Math. Anal. Appl., vol. 62,pp. 1-23, 1978.

[6] Cornwell, Holdaway, Uson, "*Radio-interferometric imaging of very large objects: implications for array design*", A&A, 271, 697-713 (1993)

[7] D. Alexander, P. Sheridan, P.Bourke, "*An Algebraic-Geometric Model of the Receptive Field Properties of the Macaque Striate Cortex*", Proceedings Australian Neuroscience Society vol.8, February (1997).

[8] D. Ros, C. Falcon, I Juvells, J. Pavia, "*The influence of a relaxation parameter on SPECT iterative reconstruction algorithms*", Phys. Med. Biol., no. 41, pp. 925-937, 1996.

[9] D. Saint-Felix, Y. Trousset, C. Picard, C. Ponchut, R. Romeas, A. Rougee, "*In vivo evaluation of a new system for 3D computerized angiography*", Phys. Med. Biol, Vol. 39, pp. 583-595, 1994.

[10] D. Scharf, "*Magnifications: Photography with the Scanning Electron Microscope*". Schocken Books, New York (1977).

[11] E. Krestel, "*Imaging systems for medical diagnostics*", Siemens Medical Division (1991).

[12] E. Mazur, R. Gordon, "*Interpolative algebraic reconstruction techniques without beam partitioning for computed tomography*", Med Biol Eng Comput 33(1), 82-6, (1995)

[13] E.Buschbeck, B.Ehmer, R.Hoy, "*Chunk versus point sampling: visual imaging in a small insect*". Science 286(5442), 1178-1180 (1999).

[14] F. Noo, C. Bernard, F.X. Litt, P. Marchot, "*A comparison between filtered backprojection algorithm and direct algebraic method in fan beam CT*", Signal Processing, 51:191-199, 1996.

[15] G. Rote, "*Computing the minimum Hausdorff distance between two point sets on a line under translation*". Information Processing Letters, v. 38, pp. 123-127 (1991).

[16] G. Tirunelveli, R. Gordon, S. Pistorius, "*Comparison of Square-Pixel and Hexagonal-Pixel Resolution in Image Processing*", IEEE CCECE-2002 Proceedings.

[17] G. Wang, M.W. Vannier, P. Chong, "*Iterative X-ray Cone-Beam Tomography for Metal Artifact Reduction and Local Area Reconstruction*", Microscopy and Microanalysis, Microscopy Society of America (1999)

[18] G.T. Herman, A.Lent, S.Rowland, "*ART: Mathematics and Applications (a report on the mathematical foundations and on the applicability to real data of the Algebraic Reconstruction Techniques)*", J.Theor.Biol, (1973)

[19] G.T. Herman, L.B. Meyer, "*Algebraic reconstruction can be made computationally efficient*", IEEE Trans. Med. Img, vol. 12, no. 3, pp. 600-609, 1993.

[20] G.T.Herman, "*Image Reconstruction from Projections: The Fundamentals of Computed Tomography*", Academic Press, New York (1980).

[21] H. Guan, R. Gordon, "*A projection access order for speedy convergence of ART: a multilevel scheme for computed tomography*", Phys. Med. Biol., no. 39, pp. 1005-2022, 1994.

[22] H. U. Frey, S. Frey, "*Tomographic methods for magnetospheric applications*", in press in Ann.Geophys, 1998.

[23] H.E. Cline, W.E. Lorensen, S. Ludke, C.R. Crawford, B.C. Teeter, "*Two algorithms for the three-dimensional reconstruction of tomograms*", Med.Phys. 15, 320-327 (1988)

[24] Imaginis Breast Health Web Site, URL: http://imaginis.com/ct-scan/biopsy.asp, January 2003. Figure 1-1 is reproduced from URL: http://imaginis.com/ct-scan/how_ct.asp. Copyright © 1997-2001 Imaginis Corporation All rights reserved

[25] J. Foley, A. Van Dam, J.Hughes, "*Computer Graphics, Principle and Practice (Second Edition)*". Addison Wesley, Sydney (1990).

[26] K. Mueller, "*Fast and accurate three-dimensional reconstruction from cone-beam projection data using algebraic methods*", PhD thesis, The Ohio State University, 1998.

[27] K. Mueller, R. Yagel, J.F. Cornhill, "*The Weighted Distance Scheme: A Globally Optimizing Projection Ordering Method for* ART", Transactions on Medical Imaging, Vol.16, No.2, April 1997.

[28] K. Shoemaker, "*Animating Rotation with Quaternion Curves*", Computer Graphics (1985)

[29] L. Youping, X. Qingfen, B. Guoliang, "*Study on Quality Evalutaion of Compressed Remote Sensing Images*", Beijing Remote Sensing Information Institute, ACRS (1990)

[30] M. Goesele, W. Heldrich, H.Seidel, "*Entropy-Based Dark Frame Subtraction*", Image Processing, Image Quality, Image Capture Systems Conference (PICS) 2001.

[31] M.B. Katz, "*Questions of Uniqueness and Resolution in Reconstruction from Projections*", Lecture Notes in Biomathematics, Springer-Verlag, New York (1978)

[32] M.C. van Dijke, "*Iterative methods in image reconstruction*" Ph.D. Dissertation, Rijksuniversiteit Utrecht, The Netherlands, 1992.

[33] O. Magnan, P. Grangeat, R. Proksa, "*Real Time Motion Compensated Reconstruction and Visualization for Dynamic Computed Tomography*", DynCT Consortium – 2000

[34] P. Agarwal, M. Shamir, "*Efficient Algorithms for Geometric Optimization*", ACM Computing Surveys 30(4): 412-458 (1998)

[35] R. Gordon, "*A tutorial on ART (Algebraic Reconstruction Techniques)*", IEEE Trans. Nucl. Sci. NS-21, 78-93, 95, (1974).

[36] R. Gordon, "*Artifacts in reconstructions made from a few projections*", Proceedings of the First International Joint Conference on Pattern Recognition, IEEE Computer Society (1973).

[37] R. Gordon, G.T. Herman, "*Three-Dimensional Reconstruction from Projections: A Review of algorithms*", Proceedings of the Society of Photo-Optical Instrumentation Engineers, Vol. 47 (1975)

[38] R. Gordon, G.T. Herman, S.A. Johnson, "*Image Reconstruction from Projections*", Scientific American, Vol. 233, no.4, pp.56-68, (October 1975).

[39] R. Gordon, R. Bender, G.T. Herman, "*Algebraic reconstruction techniques (ART) for three-dimensional electron microscopy and X-ray photography*", J. Theoret. Biol., vol. 29, pp. 471-482, 1970.

[40] R.A. Robb et al, "*High-speed three-dimensional x-ray computed tomography: The Dynamic Spatial Reconstructor*", Proc. IEEE, 71:308-319 (1983)

[41] R.A. Robb, "*X-ray Computed Tomography: An engineering synthesis of multiscientific principles*", CRC Crit. Rev. Biomed. Eng, 7:264-333 (1982)

[42] R.M. Rangayyan, A.Kantzas, "*Image Reconstruction*", Wiley Encyclopedia of Electrical and Electronics Engineering, Supplement 1, Wiley-Interscience publication, New York, (2000)

[43] S Horbelt, M Liebling, M Unser, "*Filter design for filtered back-projection guided by the interpolation model*", CH-1015 Lausanne EPFL, Biomedical Imaging Group, Swiss Federal Institute of Technology.

[44] S. Matej, G.T. Herman, T.K. Narayan, S.S. Furuie, R.M. Lewitt, P.E. Kinahan, "*Evaluation of task-oriented performance of several fully 3D PET reconstruction algorithms*", Phys. Med. Biol, Vol. 39, pp. 355-367, 1994.

[45] S.A. Larsson, "*Gamma Camera Emission Tomography*", Acta Radiol. Suppl., 363 (1980).

[46] T. Yamada, "*A Progressive Scan CCD Image Sensor for DSC Applications*", IEEE Journal of Solid-State Circuits, vol. 35 (12), December 2000 pp. 2044-2054.

[47] Walker, I. K., J. A. T. Heaton, L. Kersley, C. N. Mitchell, S., E. Pryse, M. J. Williams, "*EISCAT verification in the development of ionospheric tomography*", Ann. Geophys, 14, 1413-1421, 1996.

[48] Z.H. Cho, J.P. Jones, M. Singh, "*Foundations of Medical Imaging*", Wiley, New York (1993).