2005-07-22

Constraint-Based Interpolation

Daniel David Goggins
Brigham Young University - Provo

CONSTRAINT-BASED INTERPOLATION

by

Dan Goggins

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

August 2005

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Dan Goggins

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Bryan S. Morse, Chair
Date	William A. Barrett
Date	Irene Geary

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Dan Goggins in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Bryan S. Morse
Chair, Graduate Committee

Accepted for the Department

David W. Embley
Department Chair

Accepted for the College

G. Rex Bryce,
Associate Dean, College of Physical and Mathematical Sciences

ABSTRACT

CONSTRAINT-BASED INTERPOLATION

Dan Goggins

Department of Computer Science

Master of Science

Image reconstruction is the process of converting a sampled image into a continuous one prior to transformation and resampling. This reconstruction can be more accurate if two things are known: the process by which the sampled image was obtained and the general characteristics of the original image. We present a new reconstruction algorithm known as *Constraint-Based Interpolation*, which estimates the sampling functions found in cameras and analyzes properties of real world images in order to produce quality real-world image magnifications.

To accomplish this, Constraint-Based Interpolation uses a sensor model that pushes the pixels in an interpolation to more closely match the data in the sampled image. Real-world image properties are ensured with a level-set smoothing model that smooths "jaggies" and a sharpening model that alleviates blurring.

This thesis describes the three models, their methods and constraints. The effects of the various models and constraints are also shown, as well as a human observer

test. A variation of a previous interpolation technique, *Quad-based Interpolation*, and a new metric, *gradient weighted contour curvature*, is presented and analyzed.

ACKNOWLEDGMENTS

I would first like to thank Prof. Bryan Morse for all of his support, time, and patience. Without his efforts, this thesis would never have existed. I would also like to thank my wife for supporting me and helping me be a better person.

Contents

1 Introduction **1**

1.1 Interpolation 1

1.2 Motivation 2

1.3 Thesis Overview 3

2 Previous Work **5**

2.1 Function Fitting Methods 5

2.1.1 Pixel Replication 6

2.1.2 Bilinear Interpolation 6

2.1.3 Bicubic Interpolation 8

2.2 Filtering Methods 10

2.3 Edge-Directed Interpolation 11

2.4 Learning-Based Algorithms 11

2.5 Bayesian Statistics 13

3 Innovations and Methods **15**

3.1 Framework 15

3.2 Initial Interpolation 17

3.3 Sensor Model 21
3.4 Smoothing Model 26
3.5 Sharpening Model 27
3.6 Sharpening Curvature Constraint 30
3.7 Color 32
3.8 Drawbacks 33

4 Results **35**

4.1 Visual Comparison 36
4.2 Human User Study 66
4.3 Quantitative Analysis 67
4.4 Accuracy Measurements 68
4.5 Curvature Measurements 73

5 Summary and Future Work **75**

5.1 Summary 75
5.2 Future Work 76
5.2.1 Sharpening Curvature Constraint Limit 76
5.2.2 Stopping Criteria 77
5.2.3 Arbitrary Warps 77
5.2.4 Initial Interpolations and Weights 77

Bibliography **79**

A User Study Images **81**

A.1 Lena 81

A.2 Monarch 81
A.3 Frymire 81
A.4 Zebra 81

B Source Code Fragments **99**
B.1 Sensor Model 99
B.2 Sharpening Model - Calculate Nudge Flow 102
B.3 Sharpening Model - Sharpening Constraint 105
B.4 Quad-Based Interpolation 110
B.5 Quad-Based Interpolation Cost Function 113

List of Figures

1.1 Artifact Comparison 2

2.1 Pixel Replication Example 6
2.2 Linear Interpolation 7
2.3 Bilinear Interpolation 8
2.4 Pixel Replication vs Bilinear Interpolation 9
2.5 Linear vs Cubic Interpolation 9
2.6 Bilinear vs Bicubic Interpolation 10
2.7 Edge-Directed vs Bilinear Interpolation 12
2.8 Optimal Face Reconstruction Example 13

3.1 Constraint-Based Framework 16
3.2 Quadrilateral Selection 19
3.3 Quadrilateral Values 19
3.4 Quad-Based Interpolation 20
3.5 Sensor Model as Constraint 22
3.6 Sensor Model Bounding Box 23
3.7 Non-integer and Non-uniform Magnification Scales 25
3.8 Sharpening Model 28

3.9 Sharpening Model Results 29
3.10 Example of Curves. 30
3.11 Sharpening Curvature Constraint Results 32

4.1 Face Image Result - Original 37
4.2 Face Image Result - Bilinear 38
4.3 Face Image Result - Bicubic 39
4.4 Face Image Result - Levelset 40
4.5 Face Image Result - Constraint Based 41
4.6 Face Image Result - Comparisons 42
4.7 Lena Image Result - Original 43
4.8 Lena Image Result - Bilinear 44
4.9 Lena Image Result - Bicubic 45
4.10 Lena Image Result - Levelset 46
4.11 Lena Image Result - Constraint Based 47
4.12 Lena Image Result - Comparisons 48
4.13 Monarch Image Result - Original 49
4.14 Monarch Image Result - Bilinear 50
4.15 Monarch Image Result - Bicubic 51
4.16 Monarch Image Result - Levelset 52
4.17 Monarch Image Result - Constraint Based 53
4.18 Monarch Image Result - Comparisons 54
4.19 Baboon Image Result - Original 55
4.20 Baboon Image Result - Bilinear 56
4.21 Baboon Image Result - Bicubic 57

4.22 Baboon Image Result - Levelset 58
4.23 Baboon Image Result - Constraint Based 59
4.24 Baboon Image Result - Comparisons 60
4.25 Text Image Result - Original 61
4.26 Text Image Result - Bilinear 62
4.27 Text Image Result - Bicubic 63
4.28 Text Image Result - Levelset 64
4.29 Text Image Result - Constraint Based 65
4.30 Text Image Result - Comparisons 65
4.31 Mean Squared Error Results 69
4.32 Mean Absolute Error Results 69
4.33 Cross-Correlation Coefficient Results 70
4.34 Mean Contour Curvature Results 70
4.35 Mean Weighted Contour Curvature Results 71

A.1 Original Lena Image 82
A.2 8x Bilinear Lena Image 83
A.3 8x Level-set Reconstruction Lena Image 84
A.4 8x Constraint-Based Interpolation Lena Image 85
A.5 Original Monarch Image 86
A.6 4x Bilinear Interpolation Monarch Image 87
A.7 4x Level-set Reconstruction Monarch Image 88
A.8 4x Constraint-Based Interpolation Monarch Image 89
A.9 Original Frymire Image 90
A.10 8x Pixel Replication Frymire Image 91

A.11 8x Bilinear Interpolation Frymire Image 92

A.12 8x Level-set Reconstruction Frymire Image 93

A.13 8x Constraint-Based Interpolation Frymire Image 94

A.14 Original Zebra Image 95

A.15 8x Bilinear Interpolation Zebra Image 96

A.16 8x Level-set Reconstruction Zebra Image 97

A.17 8x Constraint-Based Interpolation Zebra Image 98

Chapter 1

Introduction

1.1 Interpolation

Image resolution, or the appearance of resolution, is important in many aspects of today's digital world. For instance, high-resolution cameras are able to digitize scenes at a much finer scale and thus capture much more detail than lower-resolution cameras. Unfortunately, not all pictures and images can be stored at high resolution due to equipment, memory, and in the case of the Internet, bandwidth limitations. Consumers still need low-resolution images to be enlarged to higher resolution for viewing, printing, and editing, creating a need for interpolation algorithms that give end-users these magnified images.

Image reconstruction is the process of converting a sampled image into a continuous one prior to transformation and resampling [2]. Image reconstruction can be a difficult problem for a variety of reasons. How does one create high-resolution information from a small set of low-resolution samples? The estimation in the reconstruction process of values between sampled points has a large effect on the final

(a) (b) Jaggies (c) Blurring (d) Ghosting

Figure 1.1: Zebra image (a) magnified by 3× for artifact comparison (b-d)

magnified image quality. A poor estimator will produce a poor high-resolution reconstruction. Current interpolation algorithms attempt to solve this estimation in a number of ways. A discussion of each of the methods will be further presented in Chapter 2.

1.2 Motivation

Many interpolation algorithms currently used in consumer products produce magnifications that include undesirable artifacts (flaws). These artificats include blurring, "jaggies", and "ghosting" (Figure 1.1). An explanation of these artifacts and in what interpolation techniques they occur are given in Chapter 2. The consumer, presented with the magnification, will be unsatisfied with the unrealistic image upon seeing the artifacts. As is known, sharpness of edges and freedom from artifacts are two critical factors in the perceived quality of the interpolated image [1]. More advanced magnification methods such as learning-based algorithms require specific training data, large running times, or extensive user input. There is a need for a more general interpolation technique that does not have these drawbacks.

The motivation of this research is to create a magnification algorithm that creates realistic real-world image magnifications that do not require a large amount of user input. A realistic real world image would be an image that is free from artifacts such as blurring and jaggies. The image must include smooth contours and also rapid edge transitions in areas of the image where sharp edges are found in the low-resolution image. Constraint-Based Interpolation can produce magnifications with these properties.

1.3 Thesis Overview

In order to provide the reader with a brief foundation in interpolation techniques, an overview of previous magnification algorithms is presented in Chapter 2. The algorithms that are discussed include function-fitting methods, filtering methods, edge-directed algorithms, and learning-based methods. An explanation of Bayesian statistics and how it applies in a magnification scheme will also be given.

After the background information is provided, an in-depth description of Constraint-Based Interpolation is given in Chapter 3. This chapter presents how Constraint-Based Interpolation models a Bayesian framework and explains how each of the components of the algorithm, the Sensor Model, Level-set Smoothness Model, and Sharpness Model fit into the framework. Also, Chapter 3 describes the overall algorithm, including a simple interpolation technique known as Quad-Based Interpolation, which is based on Data-Dependent Triangulation [15].

Chapter 4 presents qualitative and quantitative analysis of Constraint-Based Interpolation, comparing results with various other magnification techniques. The qualitative analysis is achieved through a human user study, and the results are

presented. The quantitative analysis is performed by a number of methods including Mean Squared Error, Mean Absolute Error, Cross-Correlation Coefficient, Mean Contour Curvature, and a new method known as Gradient Weighted Mean Contour Curvature.

Chapter 5 provides a summary of this thesis and describes possible future work that could improve the Constraint-Based algorithm. Example images and code fragments are included in appendices.

Chapter 2

Previous Work

Many algorithms have been created to construct high-resolution images. There are several general approaches that these algorithms take, including function-fitting, filter-based, edge-directed, learning-based, and statistical-based. These approaches are explained in this chapter, and examples of each algorithm are shown.

2.1 Function Fitting Methods

Interpolation is the estimation of values in a function between known points. When an image is magnified, the high-resolution grid contains pixels that need to be interpolated, or in other words, a value needs to be estimated for them. For instance, in 3x magnification, only 1/9 of the pixels in the high resolution grid are known from the low resolution grid. The remaining 8/9 of the pixels need to be estimated. There are several basic function-fitting methods, including Pixel Replication, Bilinear Interpolation, and Bicubic Interpolation.

(a) (b) 4× Pixel Replication (c) Original image

Figure 2.1: Pixel Replication Example. Original image downsampled by a scale of four (a) then magnified 4× by using Pixel Replication. As shown, images made with Pixel Replication are extremely jagged.

2.1.1 Pixel Replication

Pixel replication is by far the simplest and fastest function fitting method. In order to estimate the unknown pixels in the high resolution grid, it simply uses the value of the nearest neighbor, or in other words, the closest original pixel value. Hence the higher resolution image is blocky and jagged, since the original pixels have "grown" by exactly the magnification scale. An example of Pixel Replication is given in Figure 2.1.

2.1.2 Bilinear Interpolation

As can be inferred from the name, Bilinear Interpolation fits a piecewise linear function between known pixel values. While somewhat more complicated than Pixel Replication, the visual results are greatly improved. Figure 2.2 gives an example of linear interpolation in one dimension. As can be seen from the figure, unknown pixel values in the high resolution grid are estimated to exactly lie on the line that fits between two original pixel values. For example, in 2x magnification in one dimension,

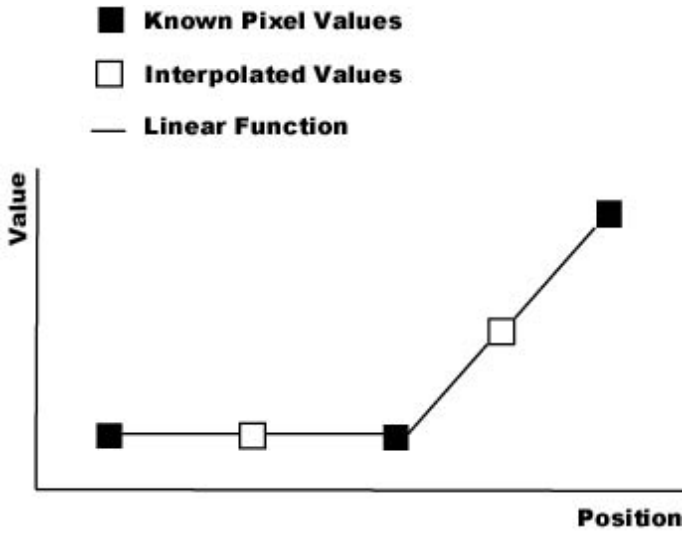


Figure 2.2: Linear Interpolation. A linear function is fit between known values and all interpolated values fall on the fitted line.

there would be only one unknown pixel between known pixel values. The pixel value would be calculated by taking half of the first known pixel plus half of the second known pixel. In 10x magnification, the first unknown pixel (from left to right) would be estimated by taking nine tenths of the first known pixel value plus one tenth of the second pixel value. This is how interpolation is performed in one dimension.

Linear interpolation extends easily into two dimensions. Bilinear Interpolation can be described as performing linear interpolation in one dimension followed by linear interpolation in the other. For instance, if we are estimating a pixel between a block of four original values, then two temporary values are first created: one, a linear interpolation between the top pair of pixels; and the second, a linear interpolation between the bottom pair of pixels. Lastly, a linear interpolation is performed between the two temporary values (Figure 2.3).

Bilinear Interpolation produces magnifications that are more visually appealing

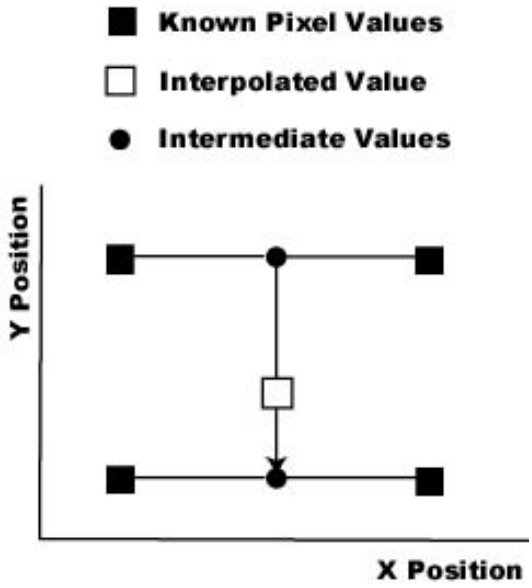


Figure 2.3: Bilinear Interpolation. A bilinear interpolation typically consists of three linear interpolations.

than Pixel Replication. This can be seen in Figure 2.4. However, due to the interpolation process, the image becomes slightly blurred since pixel values are spread throughout the high resolution grid.

2.1.3 Bicubic Interpolation

Bicubic Interpolation uses the same principle as Bilinear Interpolation, except using a cubic function instead of a linear function to estimate pixels between known values (Figure 2.5). This form of interpolation has advantages and drawbacks over Bilinear Interpolation. First, calculating the cubic polynomial in a specific area of the image is more computationally expensive than simple linear fits and also requires a larger neighborhood to calculate the curve. However, since Bicubic Interpolation utilizes a cubic curve, blurring is not as pronounced as in Bilinear Interpolation

(a) (b) 4× Pixel Replication (c) 4× Bilinear Interpolation (d) Original image

Figure 2.4: Pixel Replication vs Bilinear Interpolation. Original image downsampled by a scale of four (a) then magnified 4× by using Bilinear Interpolation. Bilinear Interpolation performs significantly better than Pixel Replication due to the increased accuracy and smoother contours.

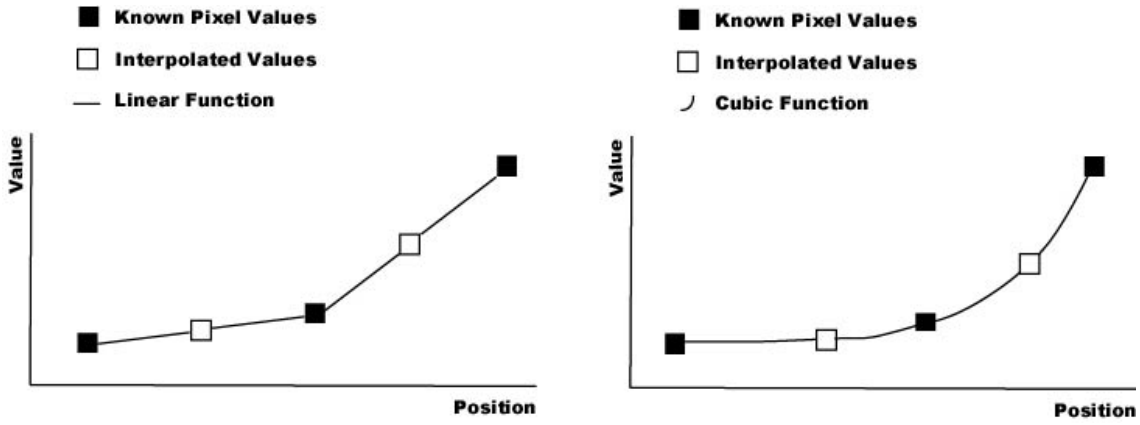


Figure 2.5: Linear vs Cubic Interpolation. A linear function fits straight lines between known points, and a cubic function fits cubic splines.

because pixel value transitions can be more rapid on the curve. On the other hand, jaggies are more distinguished since the image isn't as blurred. A comparison of Bilinear and Bicubic Interpolation is shown in Figure 2.6.

(a) (b) 4× Bilinear Interpolation (c) 4× Bicubic Interpolation (d) Original image

Figure 2.6: Bilinear vs Bicubic Interpolation. Original image downsampled by a scale of four (a) then magnified 4× using Bicubic Interpolation. Bicubic Interpolation produces magnifications which are sharper and more jagged (the eyes for example) than Bilinear Interpolation.

2.2 Filtering Methods

Another branch of magnification techniques uses filtering approaches to magnify low-resolution images. Filter-based methods use sampling theory to attempt to create perfect interpolations of images. A perfect reconstruction of sampled points is accomplished by convolving these points with a sinc function in the spatial domain. However, this is impossible due to the infinite extent of the sinc function. Filter-based methods overcome this by either truncating the sinc function, or even approximating a truncated sinc function with a cubic spline.

Due to these approximations, errors are introduced into the interpolation of the data, causing both blurring, jaggies, and ringing. Another limiting factor with filter based methods is the increased computational cost. Even with a truncated sinc function, the kernel can be quite large. For instance, the Lanczos [20] filter's kernel is 16×16, making the algorithm much more computationally expensive than Bicubic Interpolation.

2.3 Edge-Directed Interpolation

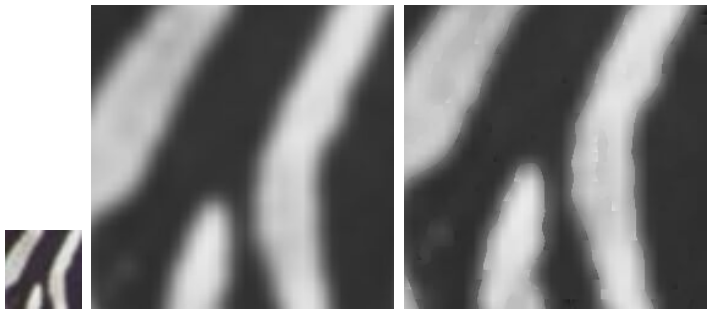
The basic idea of Edge-Directed Interpolation techniques is to analyze edge information in the low-resolution image in order to aid in the interpolation step. Edge information can be used in a variety of ways, whether to interpolate in the edge direction or to not allow interpolations to cross edges [8, 7, 13, 6, 1].

A good example of an algorithm that uses edge information to enhance the magnification is the appropriately titled "Edge-Directed Interpolation" by Allebach and Wong [1]. This algorithm uses a Laplacian-of-Gaussian filter to create a low-resolution edge map. This low-resolution map is then interpolated to a high-resolution edge map. In the rendering (interpolation) step, they modify Bilinear Interpolation in order to force interpolations to not cross edges in the edge map. The results of edge-directed interpolation show that edges remain sharp, as seen in Figure 2.7.

Edge-Directed interpolations unfortunately inherit all the downfalls of their constituent parts. For instance, Edge-Directed Interpolation by Allebach and Wong is limited to edges found by the Laplacian-of-Gaussian filter, which can detect edges in the middle of non-rapid edge transitions. This leads to "ghosting" in the final image. Also, enhancements to Bilinear Interpolation can only be seen where edges are detected, thus showing that a large portion of the image has the same problems as Bilinear Interpolation.

2.4 Learning-Based Algorithms

Another general area of interpolation methods are learning-based algorithms. These algorithms pull information from training data in order to aid in the interpolation and create high resolution data [10, 11, 12, 4, 17]. For example, "Optimal face



(a) (b) 8× Bilinear Interpolation (c) 8× Edge-Directed Interpolation

Figure 2.7: Edge-Directed vs Bilinear Interpolation. Original image downsampled by a scale of eight (a) then magnified by 8× using Edge-Directed Interpolation. Edge-Directed Interpolation produces strong edges but with ghosting.

reconstruction using training" by Muresan and Parks [11] uses pairs of low-resolution and high-resolution images of faces. This training data is then used to aid in the interpolation of low resolution face images (Figure 2.8). Data in the high resolution image is not simply interpolated but can also be "created" by analyzing similar areas of face pairs. This facet of data creation is unique to learning-based algorithms.

Unfortunately, these algorithms also have several drawbacks. First, analyzing training data can be computationally expensive. Second, the training data can require large amounts of storage memory. Also, specific training data is required for images. For instance, to make the most accurate face magnification, you need a large collection of face images to train with. Also, certain learning-based algorithms, such as Image Analogies [17], require extensive user interaction to identify regions in an image. These drawbacks hinder extensive use of learning-based algorithms for image magnification.

(a) (b) 4× Bicubic Magnified Image (c) 4× Magnified Image (d) Original Image

Figure 2.8: Optimal Face Reconstruction Example. Original image is downsampled by a scale of four (a) then magnified by 4× using Optimal Face Reconstruction [11]. Learning-Based algorithms can reconstruct areas such as the eyes far better than algorithms such as Bicubic Interpolation, but Learning-Based algorithms require specific training data in order to produce quality results.

2.5 Bayesian Statistics

Another branch of magnification techniques uses statistical frameworks, including Bayesian frameworks. These algorithms take into account prior knowledge of what the correct high resolution image should look like and of a corruption process such as Gaussian noise [3, 14]. Bayes Law is given by the following equation:

$$P(a|b) = \frac{P(b|a)P(a)}{P(b)} \tag{2.1}$$

where in a magnification model, a represents the high-resolution interpolated image and b is the original low-resolution image given to the algorithm. Algorithms based on a Bayesian framework attempt to maximize the likelihood of $P(a|b)$, which is the likelihood of the original high resolution image a being the correctly magnified image

given the low resolution data b. In order to do so, two other elements of the equation must be analyzed. $P(b|a)$ is a back analysis of an interpolation that is created through the algorithm. In particular, if we down-sampled a using the same process used to acquire b, how likely are we to get the original data b? $P(a)$ is the prior knowledge of what the high-resolution image should look like. If we are creating a real world image, our prior knowledge would tell us that our edges are sharp with smooth contours and that the image does not contain artifacts such as jaggies. It should be noted that $P(b)$ can be ignored in the equation because it is a constant, since it is the data that is obtained in the low-resolution image. Magnification algorithms based on the Bayesian framework must balance prior knowledge $P(a)$ and the original data $P(b|a)$ if a believable high resolution image is to be obtained.

As will be seen in Chapter 3, our proposed algorithm will be based on this Bayesian framework. The algorithm will remain true to the original data by relying on a sensor model that continually pushes the interpolation to match the original low-resolution image. Prior knowledge will be accounted for by a smoothing model based on level-set smoothing [2, 16], and a sharpening model that sharpens edges and hence alleviates blurring in the image.

Chapter 3

Innovations and Methods

Constraint-Based Interpolation utilizes several techniques to create realistic interpolations of images. By analyzing general properties of real-world images, several models have been implemented to ensure that an interpolation creates an image that maintains these properties. This chapter explains the framework of the Constraint-Based Interpolation algorithm in its entirety. Each step will be explained, including the initial interpolation and the iterative process of the sensor, smoothing, and sharpening models.

3.1 Framework

As stated, the Constraint-Based Interpolation algorithm was implemented in order to help ensure that image magnifications stayed true to general characteristics of real-world images. These characteristics include smooth contours with rapid edge transitions and freedom from obvious artifacts such as jaggies, ringing, and blurring. The framework of the algorithm is created in such a way as to maintain these properties of images. The algorithm is shown in Figure 3.1.

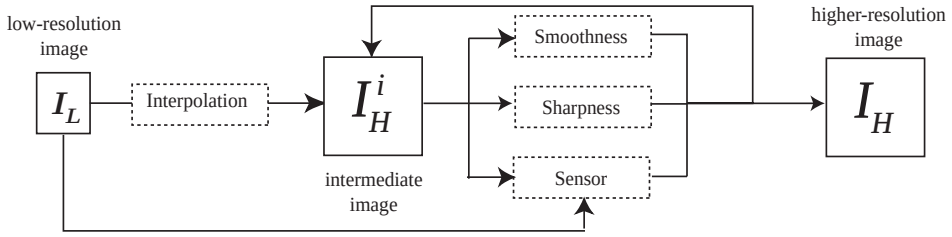


Figure 3.1: The Constraint-Based Interpolation algorithm. I_L is the low resolution image. I_H^i is the intermediate interpolated image. I_H is the final image.

The first step of our algorithm is to generate an image that is an initial approximation to the correct high-resolution reconstruction. Essentially, this step can be any number of accepted image magnification methods, such as Bilinear or Bicubic Interpolation. In fact, since we follow this initial interpolation with an iterative correction step that includes our sensor model, this interpolation can be anything. The iterative step will slowly correct the image to the desired result. We have observed that even a blank image can be iteratively corrected to create an accurate high-resolution image. However, if the initial interpolation is closer to an accurate high-resolution image, fewer iterations are needed in the correction step, improving computation time. For instance, the algorithm takes approximately twice as many iterations if given a blank initial image instead of an initial image created with bilinear interpolation.

This first interpolation is post-processed in an iterative correction step. This step attempts to smooth contours in the image and sharpen edges while preserving our original data. In each iteration, the current intermediate image I_H^i is given to the sensor, smoothing, and sharpening models. Each of these models then saves data about how much they would like each pixel in the image to change. The last step of the iteration involves combining the three models' information and creating a new

intermediate image I_H^i. To accomplish this, each of the models is given a user-defined weight determining how much force each model is able to exert on the intermediate image. Through testing, we found that appropriate weightings for the sensor model were between 0.5 and 0.7, weightings for the smoothing model range from 0.15 to 0.3, and the sharpening model weights are between 0.01 and 0.05. These weightings are allowed to be tuned by the user in order to accomodate user preference for a particular image. For instance, if the user desires that the final image be less smoothed and sharpened more, then they could use a smoothing weight of 0.15 and a sharpening weight of 0.05. The changes from each model are then combined and each pixel is updated according to the three models.

This iteration is continued until a user-specified number of iterations has lapsed. The most recent intermediate image then becomes the final higher-resolution image I_H and is presented to the user. Testing indicates that 30 iterations are usually sufficient to create an appealing magnification. If the user is still unsatisfied with the final image, then they can change weightings or increase the number of iterations.

Computation time for Constraint-Based Interpolation is linear in regards to image size and number of iterations. On a $128x128$ sized image, performing $4x$ magnification, the computation time is approximately 45 seconds running on a 2.8 Ghz Pentium 4 computer.

3.2 Initial Interpolation

As shown in Figure 3.1, the first step of the Constraint-Based Interpolation algorithm is to perform an initial interpolation. However, as we have verified experimentally, this step is not entirely necessary. The iterative correction step can create

from a blank image a higher-resolution image that is realistic if given enough iterations. Thus, this step is for efficiency. The closer the initial interpolation, the fewer iterations that are needed to correct errors in the interpolation. For example, an initial interpolation created using Pixel Replication requires about 33% more iterations than if created with Bilinear Interpolation. In our experiments, standard interpolation techniques were able to produce initial interpolations that were of sufficient quality that a reasonable (30) number of iterations were required to produce quality higher-resolution images.

As an innovation, a variation of an interpolation that was proposed by Wang and Ward [13] was created to produce initial interpolations that contained fewer "jaggies". In the variation, which we call Quad-Based Interpolation, interpolations are based on quadrilaterals that best fit the gradient direction, instead of using straight interpolations between pixels found to be partners using the gradient direction [13]. This algorithm is also similar to data-dependent triangulation [15], but instead of arbitrary triangles we use a small set of predetermined quadrilaterals and use a simple difference cost function based on gradient direction (Figure 3.2). A full explanation of this cost function and the source code fragment to calculate the cost function is given in Appendix B.

Quad-Based Interpolation consists of three steps. First, the gradient direction is calculated for every low resolution pixel. Next, for every $2x$ area in the high resolution grid, we analyze the nine quadrilateral candidates to find the best fit of the gradient direction. The nine quadrilaterals are shown in Figure 3.3. The final step in the algorithm is to perform Bilinear Interpolation within the best-fit quadrilateral, which we call Quad-Based Interpolation. To accomplish this interpolation within the

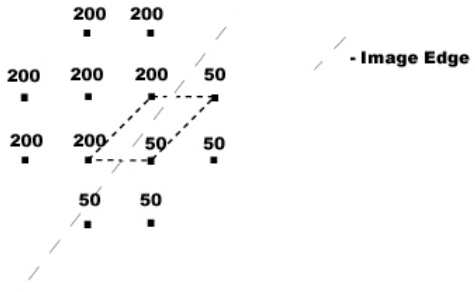


Figure 3.2: Quadrilaterals are chosen based on edge information. In this example, the quadrilateral is chosen because the gradient in the interpolation of the quadrilateral is aligned with the gradient in the image.

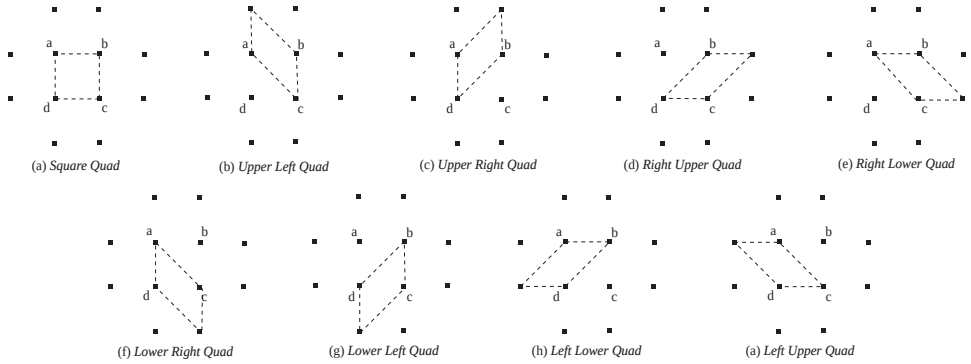
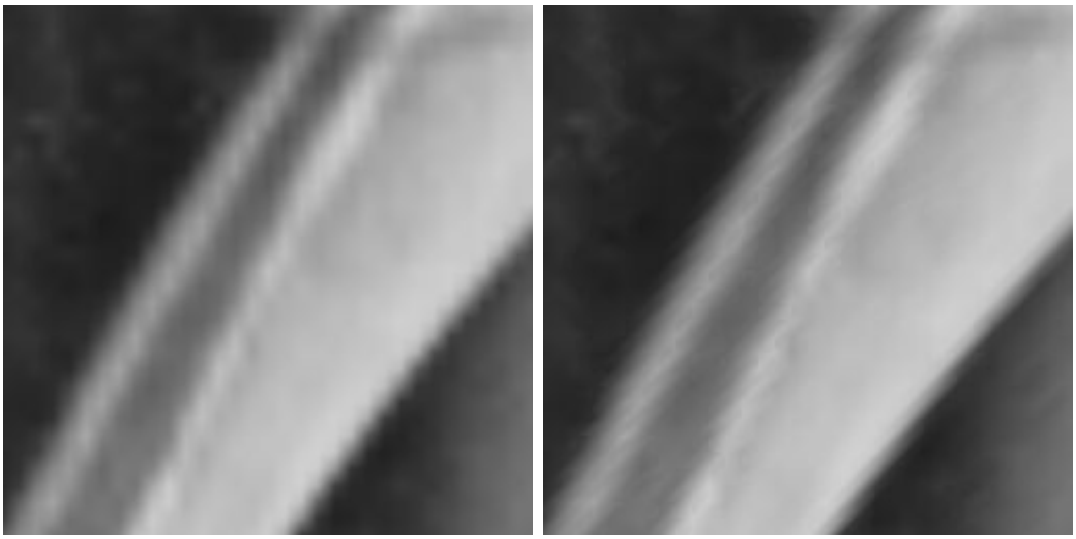


Figure 3.3: The nine possible quadrilateral values for interpolation in Quad-Based Interpolation. Gradient direction is calculated for pixels a,b,c and d. The quadrilateral that best fits the gradient direction is used to perform the interpolation.

quadrilateral, only either the x or y percentages need to be updated from standard Bilinear Interpolation. The source code for this process is also given in Appendix B. Standard Bilinear Interpolation is only able to accurately model 0 and 90 degree edge orientations, whereas Quad-Based Interpolation adds 45 degree increments as well, improving interpolations along edges (Figure 3.4).

As can be seen in Figure 3.4, Quad-Based Interpolation preserves edge direction



(a) Bilinear Interpolation (b) Quad-Based Interpolation

Figure 3.4: Quad-Based Interpolation compared to Bilinear Interpolation. Notice that the predominant edges in (b) have jaggies that are closer aligned with the edge direction instead of stepping artifacts along the edge as in Bilinear Interpolation (a).

better than Bilinear Interpolation. While Quad-Based Interpolation still produces "jaggies", these jaggies are more closely aligned with the edge when compared with the stepping artifacts in Bilinear Interpolation. Since Quad-Based Interpolation artifacts are aligned with the edge, fewer iterations are needed by the smoothing model in order to smooth them.

3.3 Sensor Model

The sensor constraint ensures that the higher-resolution image that is produced is still consistent with the data in the low-resolution image. The sensor model is similar to other constraints proposed in related magnification work [1, 14]. This model attempts to make the magnification *image-consistent*, meaning that if the high resolution image is downsampled using the same sensor function used to acquire the low-resolution original, it would exactly match the original data.

The sensor model is also used to ensure that the smoothing and sharpening models do not change the higher-resolution image too drastically. In image magnification using Level-Set Image Reconstruction [2], intensity anchors are used to constrain the diffusion process. These intensity anchors are pixels in the higher-resolution image that map directly to the low-resolution grid. These points are not allowed to change during the diffusion. The intensity anchors help to ensure that the higher-resolution image does not diffuse too greatly and thus be drastically different from the low-resolution image.

In much the same way, the sensor model ensures that the area around every pixel matches the low-resolution data. This "area anchor" is a more accurate representation of the relationship between the low-resolution and high-resolution images. The intensity anchors used in Level-Set Image Reconstruction rely on an underlying assumption that only a grid of pixel points in the high-resolution image should match the low-resolution data exactly. A more accurate assumption is used by the area anchors, indicating that any particular pixel in the low-resolution data should match a particular area in the high-resolution image exactly. In this area anchor model, if the area function is the same sensor function that was used to create the low-

(a) Face Magnification using sensor model (b) Face Magnification without sensor model

Figure 3.5: Without the sensor model, the smoothing and sharpening models are allowed to overly smooth along edges and to overly sharpen across edges

resolution image, and the area around every pixel matches the original data, then the magnification will be *image-consistent*. Such a claim can not be made by using intensity anchors alone. The sensor model also acts as a constraint for the smoothing and sharpening models. Without the sensor model, the image would become overly smooth along edges and overly sharp across edges. This is shown in Figure 3.5.

Two sensor functions were created for use in the sensor model, a Gaussian and a pillbox convolved with a box filter [14]. The latter function mimics the blurring in a camera when the optics and CCD elements capture the scene. The general algorithm of the sensor model is as follows. First, a kernel is created in order to compute area averages. This kernel is then passed over the high-resolution image. A difference calculation is then performed for pixels that map exactly to low-resolution pixels.

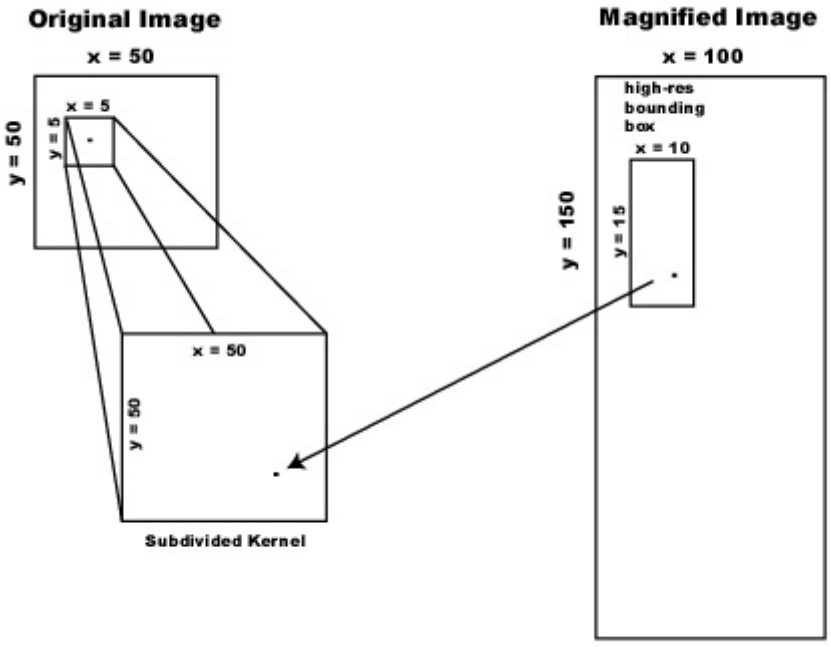


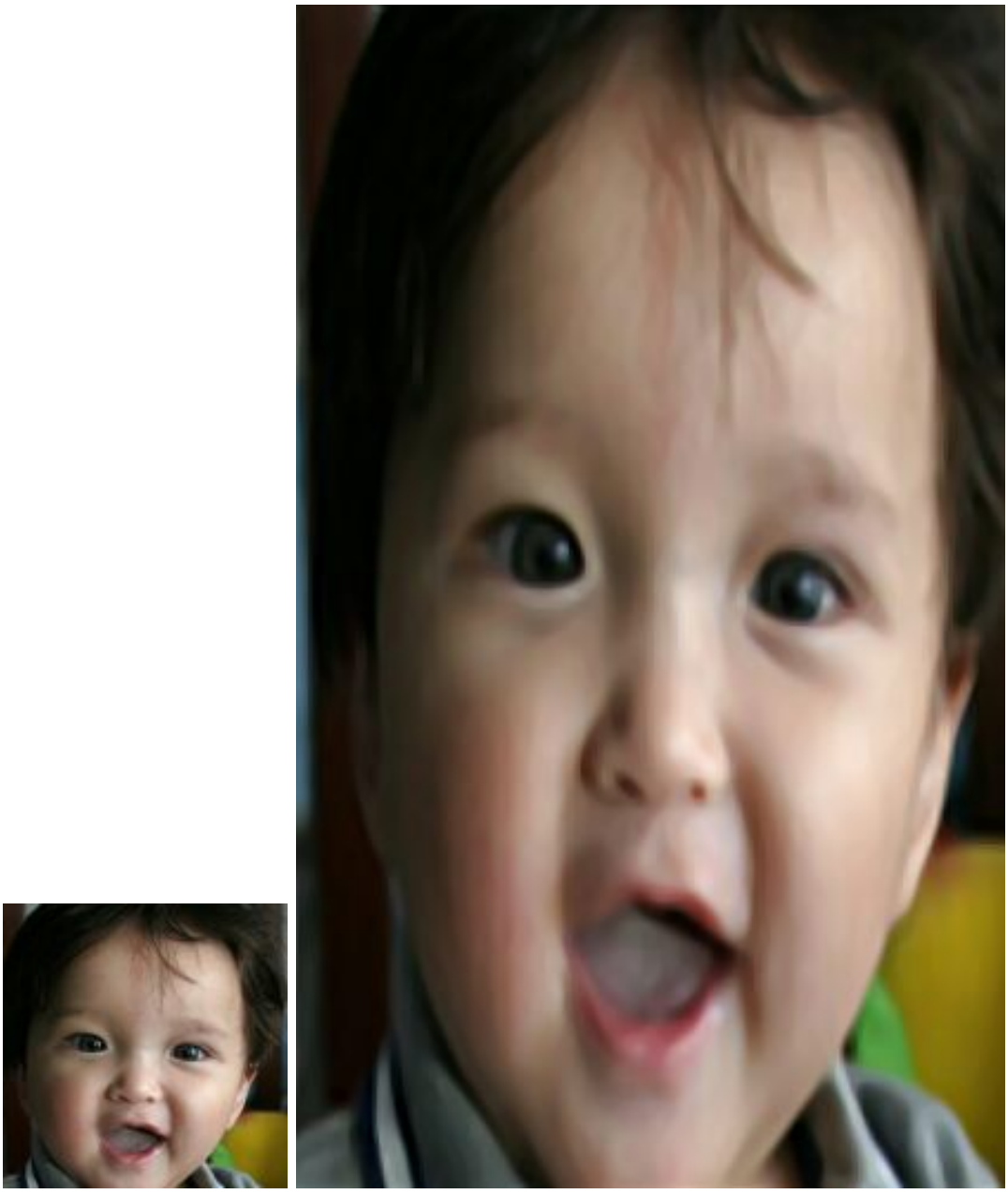
Figure 3.6: The Sensor Model Bounding Box. In 2×3 magnification, the original kernel is 5×5, which translates to a 10×15 high-resolution bounding box. Each high-resolution pixel in this bounding box is mapped to the closest value in the super-sampled kernel.

This difference value is then spread over the high-resolution area by utilizing the same kernel weights. These values are used as the sensor model.

In order to allow for arbitrary non-integer and non-uniform magnification scale values, this algorithm has to be modified. When an image is magnified by arbitrary magnification scale values, the high-resolution grid becomes distorted. For instance, in 2×3 magnification, the high-resolution grid no longer has the same aspect ratio as the low-resolution grid (Figure 3.6). One solution is to alter the kernel function to include this distortion, which can be a difficult task for an arbitrary warp. An

explanation of a simpler solution is as follows. The sensor model requires that we compute the difference between an area average in the high-resolution image and the corresponding pixel in the low-resolution image. However, arbitrary non-integer and non-uniform magnification scale values produce a high-resolution coordinate system that does not have the same aspect ratio as the low-resolution coordinate system. Yet no matter what the magnification scale values are, each pixel in the high-resolution image is an interpolation of a sub-pixel point in the low-resolution image. So, the algorithm is modified to account for this. For each pixel in the low-resolution image we analyze the footprint of the sensor function, since we need to find all high-resolution pixels that fall within this footprint. As an optimization, we create a bounding box in the high-resolution image coordinate system that includes all pixels that were sampled from within the footprint. To calculate the weighting for each pixel in this bounding box, we could use a continuous kernel. However, this would be inefficient. To optimize this, we discretize the kernel by greatly oversampling it. In our implementation we use $10\times$ oversampling. Each high-resolution pixel is then matched with the nearest super-sampled weight in the discretized kernel grid. In order to form the area average, each high-resolution pixel is multiplied by its respective weight, and these weighted values are added together, then divided by the sum of the weights that were used. The difference between this area average and the original low-resolution pixel is then spread throughout the high-resolution bounding box to be stored as the corrective values for each high-resolution pixel. In this way, any kernel can still be used in its original form for any non-integer and non-uniform magnification scale.

An example of a magnification using arbitrary non-integer and non-uniform magnification scales is given in Figure 3.7.



(a) Original Image (b) Constraint-Based Interpolation

Figure 3.7: Non-integer and Non-uniform Magnification Scales. The Face image is magnified by 2.43× in x and 4× in y.

3.4 Smoothing Model

The smoothness constraint attempts to smooth the contours in the image in order to remove what are known as the "jaggies". This work is largely based on Level-Set Image Reconstruction [2]. The principles and techniques are thoroughly explained in Level-Set Image Reconstruction [2], yet a small introduction will be given here.

Level curves are curves of constant intensity in an image, and are especially perceptible around edges. Many interpolation techniques produce "jaggies" around edges, thus producing jagged level curves. However, real-life images generally have smooth level curves. Hence, our algorithm attempts to smooth these curves as much as possible, eliminating obvious artifacts near edges. Fortunately, explicitly finding the level curves in order to manipulate them is not necessary, since it has been shown that a level curve through a pixel can be moved by changing the intensity at that pixel. As shown in [18] and subsequent work, a flow equation can be used as a mechanism for this movement:

$$I_t = F|| \bigtriangledown I|| \tag{3.1}$$

where F is the speed of movement of the level curve in the direction of its normal. If we use the negative isophote curvature as this speed parameter, the curve will contract proportionally to its curvature. Thus areas of high curvature (jagged edges) will contract more quickly than places of low curvature, and the curve will contract inward. The flow equation now becomes

$$I_t = -\kappa|| \bigtriangledown I|| \tag{3.2}$$

The negative isophote curvature $-\kappa$ can be calculated using local derivatives of the

intensity surface by the following equation:

$$\kappa = \frac{I_x^2 I_{yy} - 2I_x I_y I_{xy} + I_y^2 I_{xx}}{(I_x^2 + I_y^2)^{\frac{3}{2}}} \tag{3.3}$$

Recognizing that $|| \bigtriangledown I|| = (I_x^2 + I_y^2)^{\frac{1}{2}}$, the desired flow is thus:

$$I_t = \frac{I_x^2 I_{yy} - 2I_x I_y I_{xy} + I_y^2 I_{xx}}{I_x^2 + I_y^2} \tag{3.4}$$

The pixel intensities are then modified according to Equation (3.4), smoothing the image. However, in typical level-set smoothing, many of the actual image features can also be removed. Level-Set Image Reconstruction [2] proposed several constraints on its implementation, such as intensity anchors. As shown in our results, the sensor model is able to ensure that many of the features of the image remain, serving as a constraint to this smoothing. Thus, this model removes much of the reconstruction artifacts that occur with our initial interpolation step by smoothing the jagged edges, while the sensor model ensures features are not smoothed away.

3.5 Sharpening Model

As can be seen in real-life images, most edges are sharp and smooth. Using this qualitative prior, we can know that in most cases our high-resolution magnification should also have sharp edges. With this prior knowledge we use a sharpening model on our iterative image, similar to other magnification algorithms [13]. In order to accomplish this, we use a shock filter [19]. The general idea of shock filters is to change pixel intensities to be closer to those of pixels in the direction away from

edges. The general form of the shock equation is given as follows:

$$I_t = -F(I_{ww})|| \bigtriangledown I|| \tag{3.5}$$

where F is a function of I_{ww} used in various implementations of the shock equation to control the flow of the filter. In our implementation of the shock filter, we use the following equation:

$$I_t = - \bigtriangledown || \bigtriangledown I|| \cdot \bigtriangledown_{upwind} I \tag{3.6}$$

This equation, once analyzed, is mathematically equivalent to the general form of the shock equation where $F(I_{ww}) = I_{ww}$. The flow of our algorithm is shown in Figure 3.8.

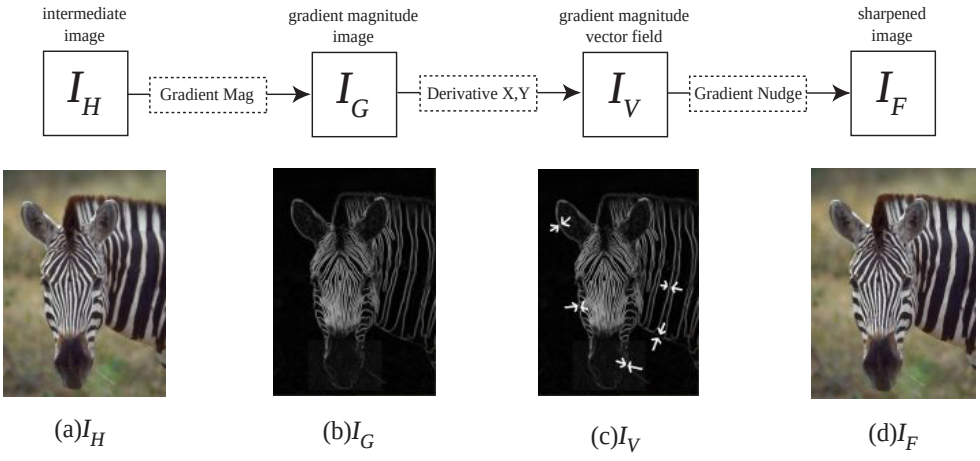


Figure 3.8: Overview of the sharpening model

The steps of the sharpening model are now explained. First, derivatives in the x and y direction of I_G (image holding the gradient magnitude values) are calculated to give a vector field indicating the direction of increasing gradient magnitude, which is generally towards edges. Upwind derivatives of the image are calculated based

on the direction indicated by the values in the vector field. Pixels intensities are then adjusted to be closer to pixels away from the edge, thus causing image level curves to move closer to the edge. The result is an image with more rapid edge transitions, increasing the overall sharpness of the image. The sensor model acts as a constraint to ensure that the filtering does not allow the high resolution image to appear "cartoonish" or overly diffused. Results of the sharpening model are shown in Figure 3.9.

(a) Original Zebra Image (b) Processed Image

Figure 3.9: The Zebra image is processed by the sharpening model alone.

3.6 Sharpening Curvature Constraint

Unfortunately, shock filters can also produce "stair-stepping" in areas of the image that have small changes in the gradient, which the shock equation is known to produce. For instance, a picture of a person's cheek can be sharpened in such a way as to create stair-stepping in the generally smooth gradient of the cheek. In order to eliminate this stairstepping, a further enhancement to the algorithm was needed. This constraint is called the sharpening curvature constraint. The purpose of this constraint is to force sharpening to be at a much slower rate in areas of the image that have smooth gradients as in case A of Figure 3.10, or no sharpening at all in areas that shouldn't be sharpened (case B). If two points fall under case C, then sharpening is allowed, but is weighted in order to allow more or less sharpening.

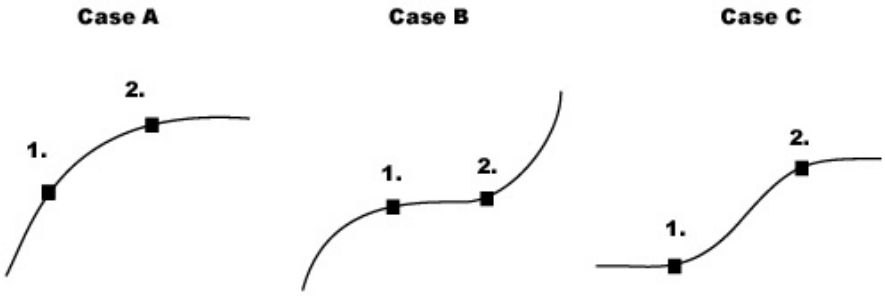


Figure 3.10: Three cases that the sharpening curvature constraint analyzes. These cases are given in 1-D for simplification.

The algorithm for the sharpening constraint is as follows. First, the cross-contour curvature image I_{ww} is calculated for the low-resolution image as follows:

$$I_{ww} = I_x^2 I_{xx} + I_x I_y I_{xy} + I_y^2 I_{yy} \tag{3.7}$$

Then, for each pixel in the high resolution image, the gradient direction w is computed, which is the potential edge normal. Next, the algorithm finds two of the four low-resolution neighbors that most closely align with w. Then, the I_{ww} values are analyzed for each of these two points. If the two values are the same sign, don't allow sharpening. This is case A of Figure 3.10. Since the sharpening model is attempting to eliminate blurring due to interpolation blur, we don't want to sharpen at a rate greater than what is present in the low resolution image. If this check is passed, then we calculate the difference between the two I_{ww} values. If the sign of the difference D is the same as the sign of the actual pixel values in the low resolution image, then sharpening is not performed. This indicates the rare case of an inflection point in the image (case B of Figure 3.10). If these two constraints are passed, then case C of Figure 3.10 is found, so $|D|$ is passed to a function that determines the rate of sharpening depending on the value of $|D|$. This function allows for greater sharpening for large $|D|$ values and little sharpening for smaller $|D|$ values. The function used asymptotically approaches 1.0 in the form:

$$|x|/(1+|x|) \tag{3.8}$$

This value is then used as a speed parameter to the sharpening model. Since the asymptotic function produces values between 0 and 1, the filter value is simply multiplied by the constraint value to produce a constrained shock filter value. Results of this constraint can be seen in Figure 3.11. Notice how the face image without the constraint has stairstepping around the eyes and nose, whereas the image with the constraint avoids this while still creating sharp edges in areas of high curvature, such as the eyes.

(a) Face without constraint (b) Face with constraint

Figure 3.11: Face image processed by the sharpening model with and without the sharpening curvature constraint.

3.7 Color

The Constraint-Based Algorithm is also easily extended to color images. This is accomplished by working on each of the three color planes separately and merging the results to produce a final tri-plane color image. However, the computation time scales linearly with the number of color channels, which in this case is three. Color separation can also become an issue when working with color images. However, the sensor model ensures that the area around every pixel remains true to the original data, thus color separation is minimal, if any. Examples of color image magnification using the Constraint-Based Algorithm are shown in Chapter 4 and in the appendices.

3.8 Drawbacks

There are several current drawbacks to the Constraint-Based Interpolation algorithm. First, if the sensor model does not accurately portray the function which was used to create the image, then errors can be introduced into the sensor model. For instance, if the image was not directly created from a digital camera, but was first printed and then scanned, then several different sensing models were used during the creation of the image and the sensor model will recreate an imperfect image.

Second, the Constraint-Based Interpolation algorithm does not currently converge to a final image. Thus, at an excessive number of iterations, for example 300, errors are introduced into the image, such as overshooting and oversmoothing errors. It is then up to the user to ensure that the algorithm does not run with too many iterations.

Lastly, certain high-details areas of an image can be lost due to the smoothing model if the smoothing weight is too large. On the other hand, if the smoothing weight is too small, then the image can retain jaggies or be sharper than expected due to the sharpening model.

However, if these drawbacks are taken into consideration and appropriate numbers of iterations and weights are chosen, then the Constraint-Based Interpolation algorithm can produce realistic high-resolution magnifications.

Chapter 4

Results

Only through extensive testing can an algorithm be considered a success. For any image reconstruction algorithm to be considered successful, it needs to excel in both a qualitative and a quantitative analysis when compared with other magnification techniques.

Constraint-Based Interpolation was tested and compared with several common magnificaton techniques. These include Pixel Replication, Bilinear Interpolation, Bicubic Interpolation, and Level-Set Image Reconstruction [2]. Qualitative analysis is provided in two ways. First, a set of image comparisons is presented to the reader for personal analysis. Second, a result of a human user study is given to provide a discrete approximation (rank) to the question of "does it look good?". A quantitative analysis is done utilizing several methods, including mean squared error, mean absolute error, cross-correlation coefficient, mean contour curvature, and mean weighted contour curvature.

As is shown in the results, Constraint-Based Interpolation performs well in both

qualitative and quantitive measurements, and can be considered successful in producing realistic higher resolution interpolations of real world images.

4.1 Visual Comparison

The first qualitative analysis of Constraint-Based Interpolation is a series of image comparisons that are presented to the reader. The reader can judge for themselves if Constraint-Based Interpolation produces results that are superior to common magnification techniques. The following pages include these image comparisons, showing various types of images and labeling their corresponding interpolation algorithm.

All images were produced using 30 iterations, with a sensor weight of 0.6, a smoothing weight of 0.2, and a sharpening weight of 0.01. The examples are magnified using a range of magnification scales in order to show variety and for space considerations. Magnifications with large scaling values (8×) are shown to demonstrate robustness of the Constraint-Based Interpolation algorithm.

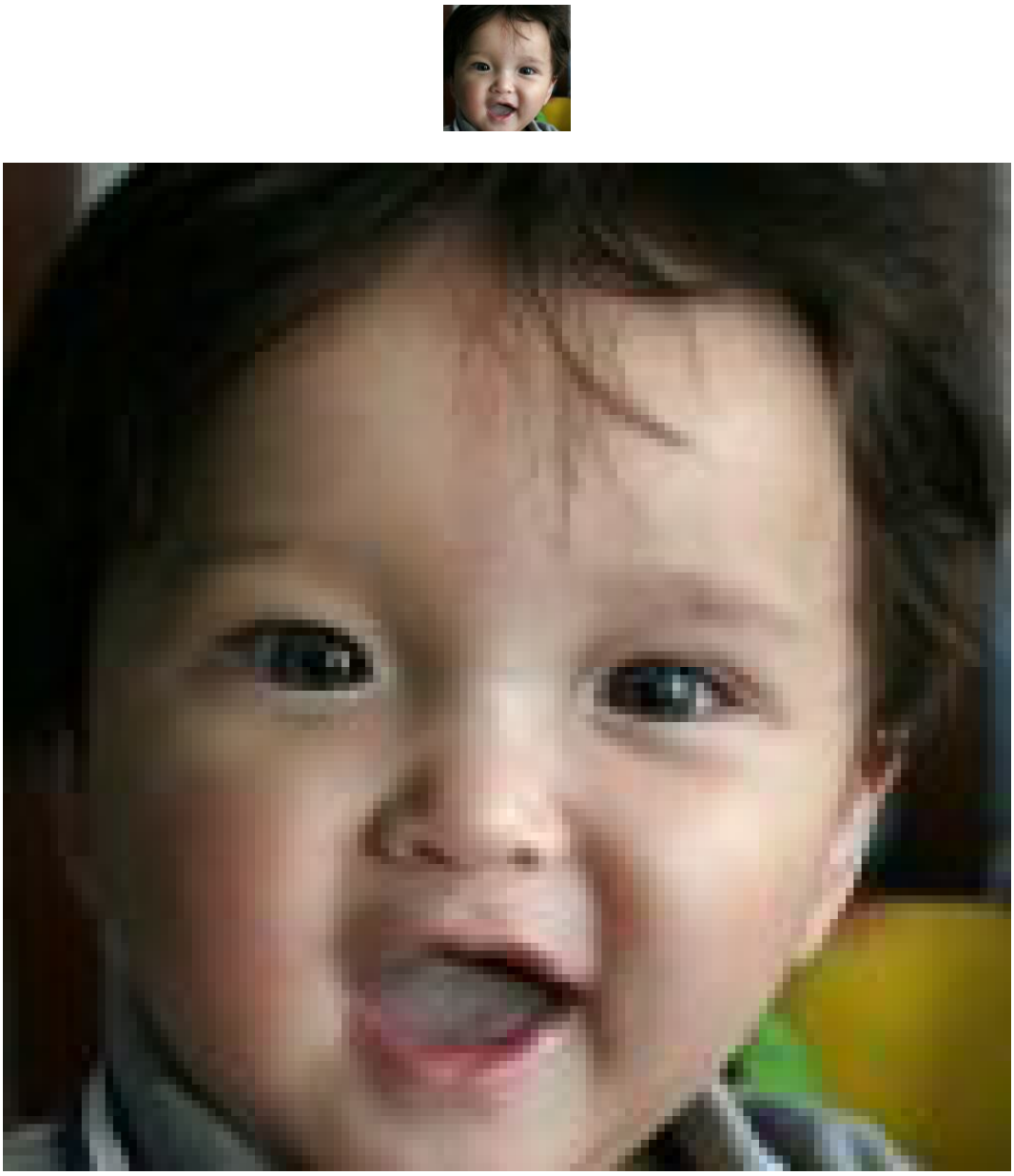


Figure 4.1: Face Image. The original face image and magnified 8× with Pixel Replication to show coarseness of data.

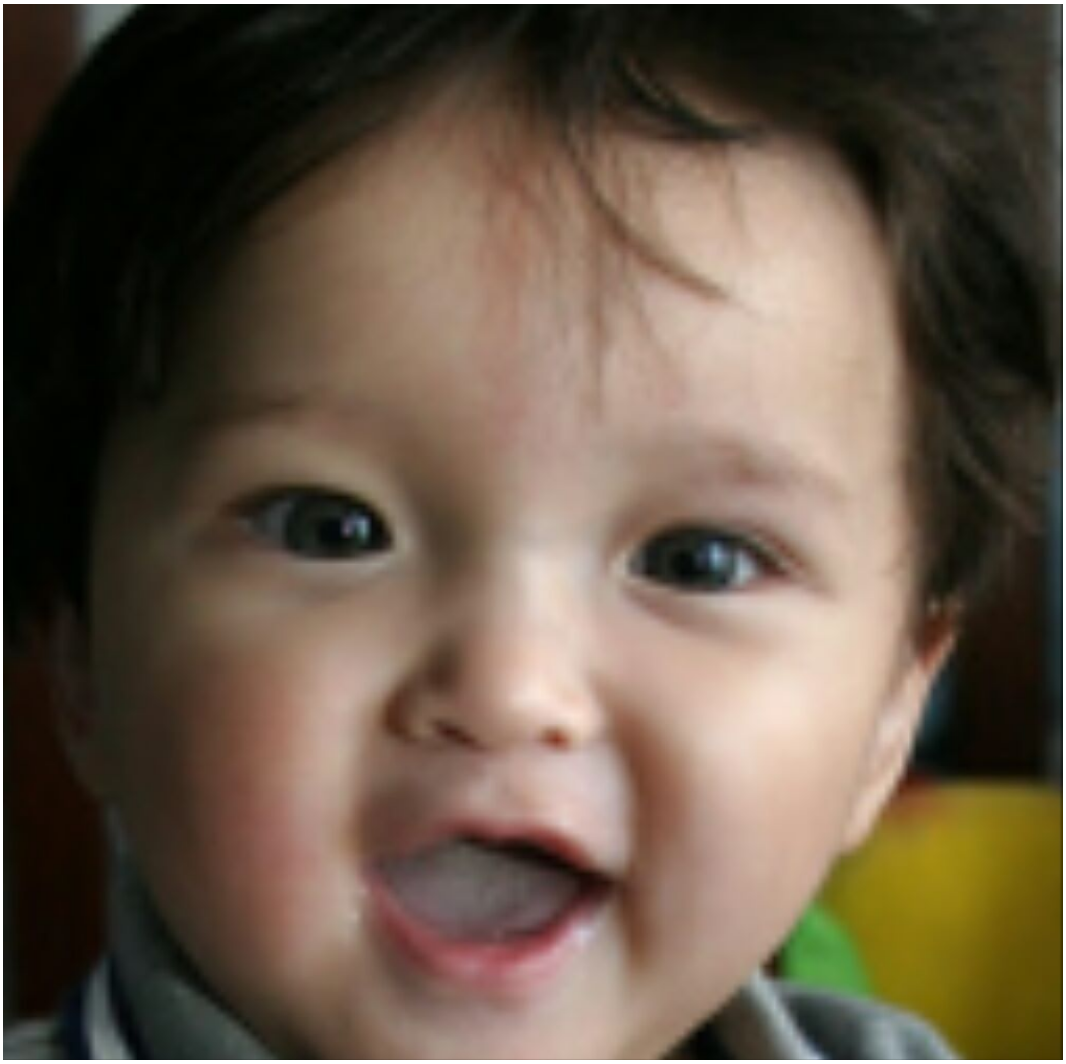


Figure 4.2: The Face image magnified 8× using Bilinear Interpolation

Figure 4.3: The Face image magnified 8× using Bicubic Interpolation

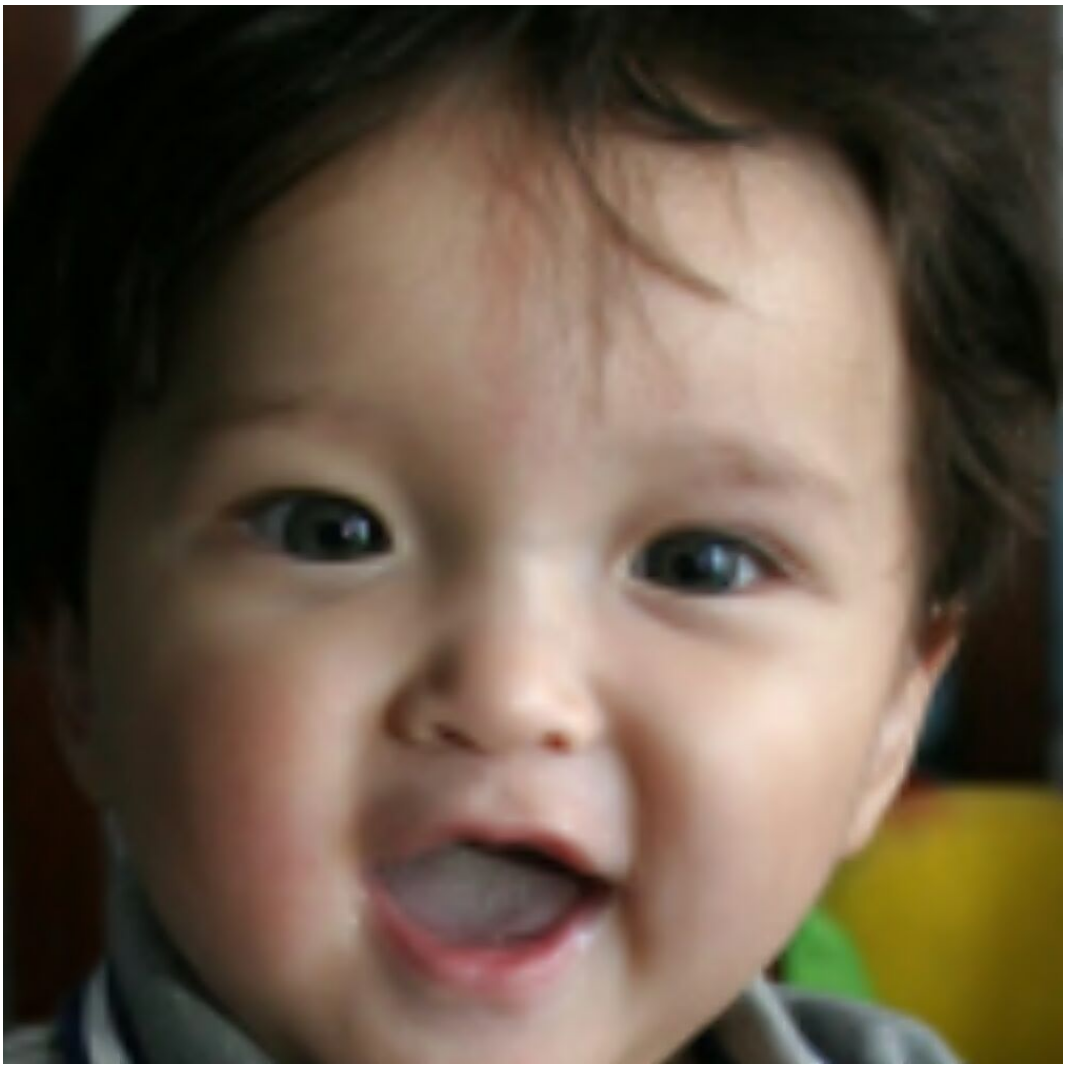


Figure 4.4: The Face image magnified 8× using Level-Set Reconstruction

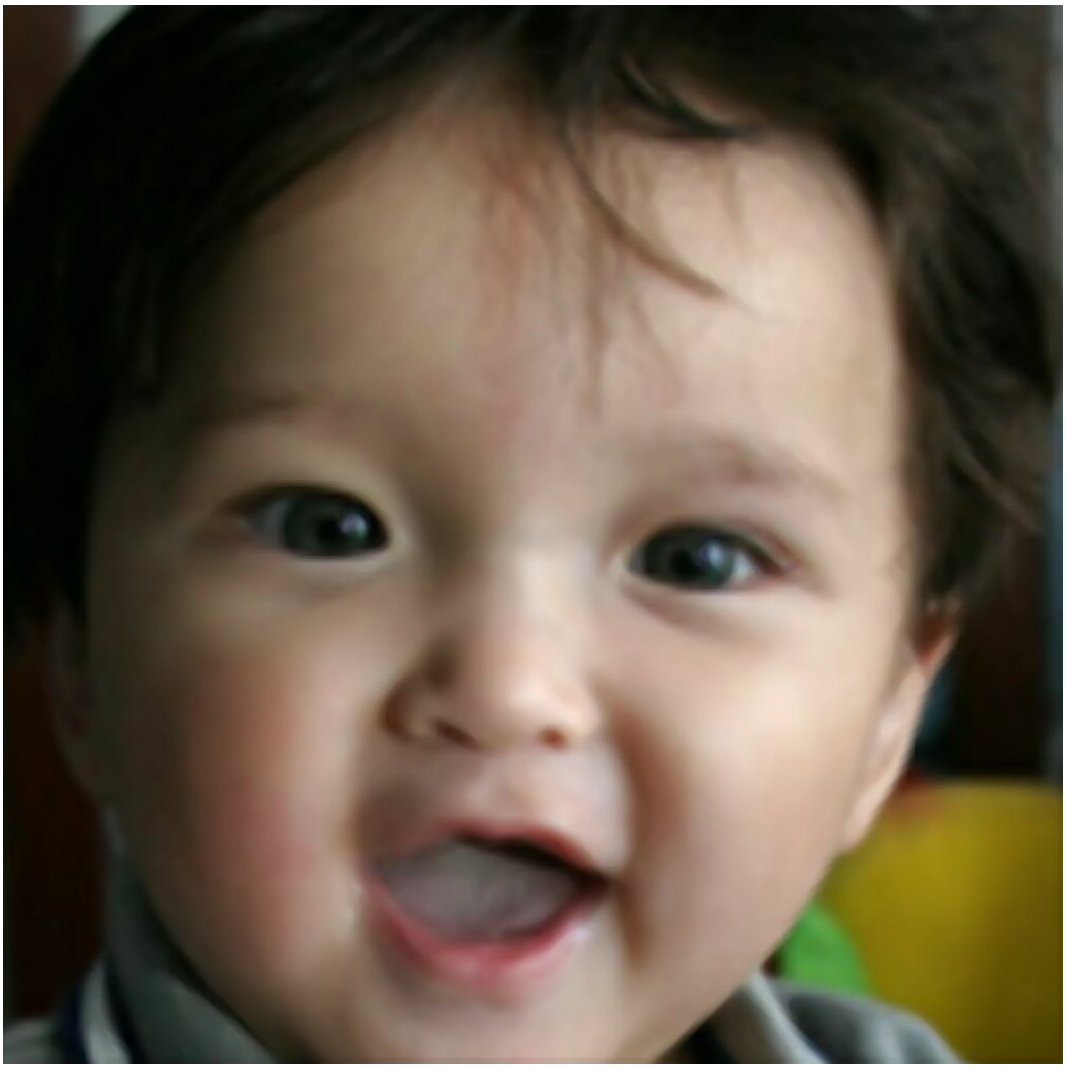


Figure 4.5: The Face image magnified 8× using Constraint-Based Interpolation. Notice the absence of jaggies around the eyelids and ears.

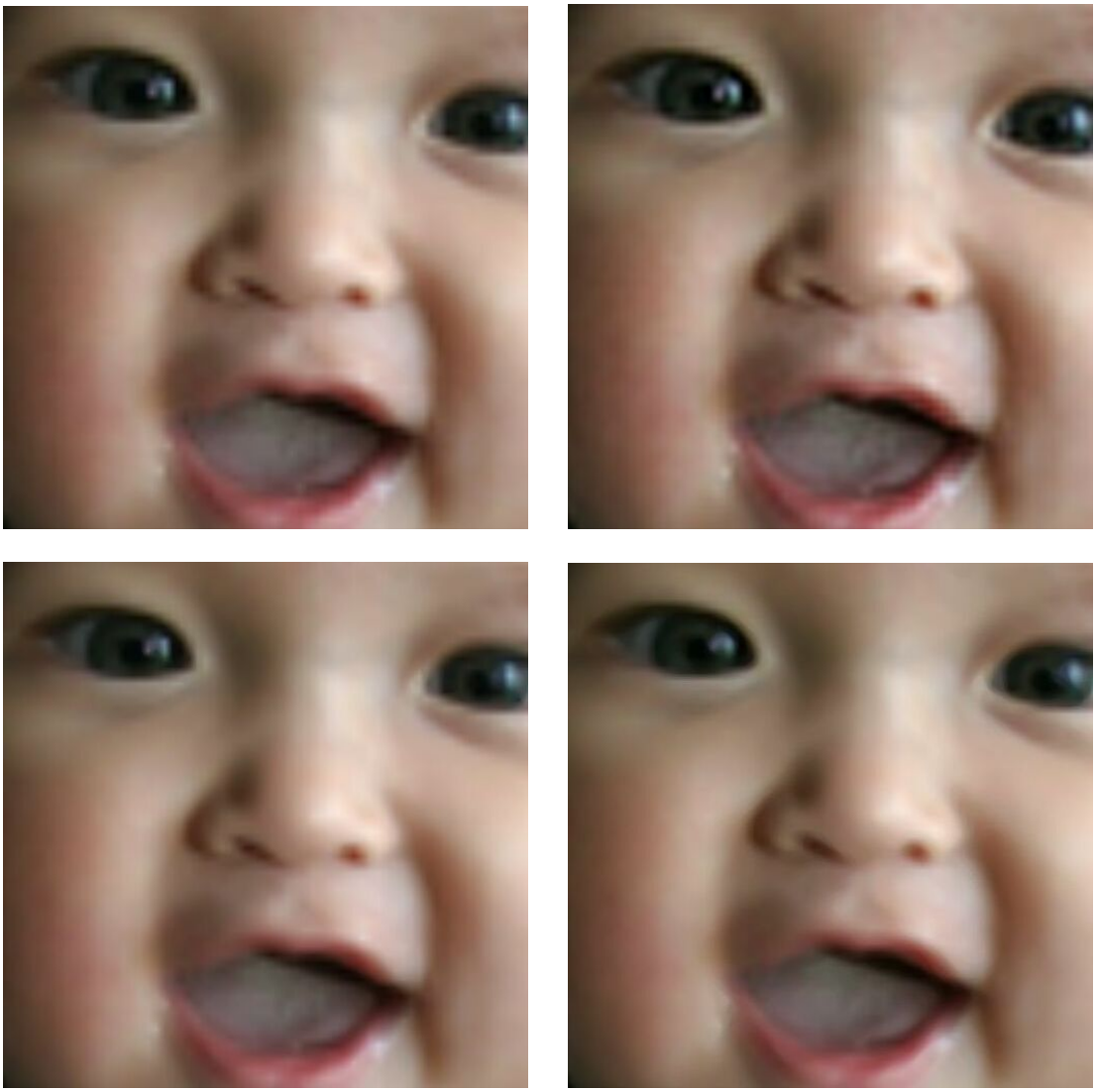


Figure 4.6: Face image comparisons. Clockwise from Upper Left: Bilinear Interpolation, Bicubic Interpolation, Constraint-Based Interpolation, Level-Set Reconstruction.

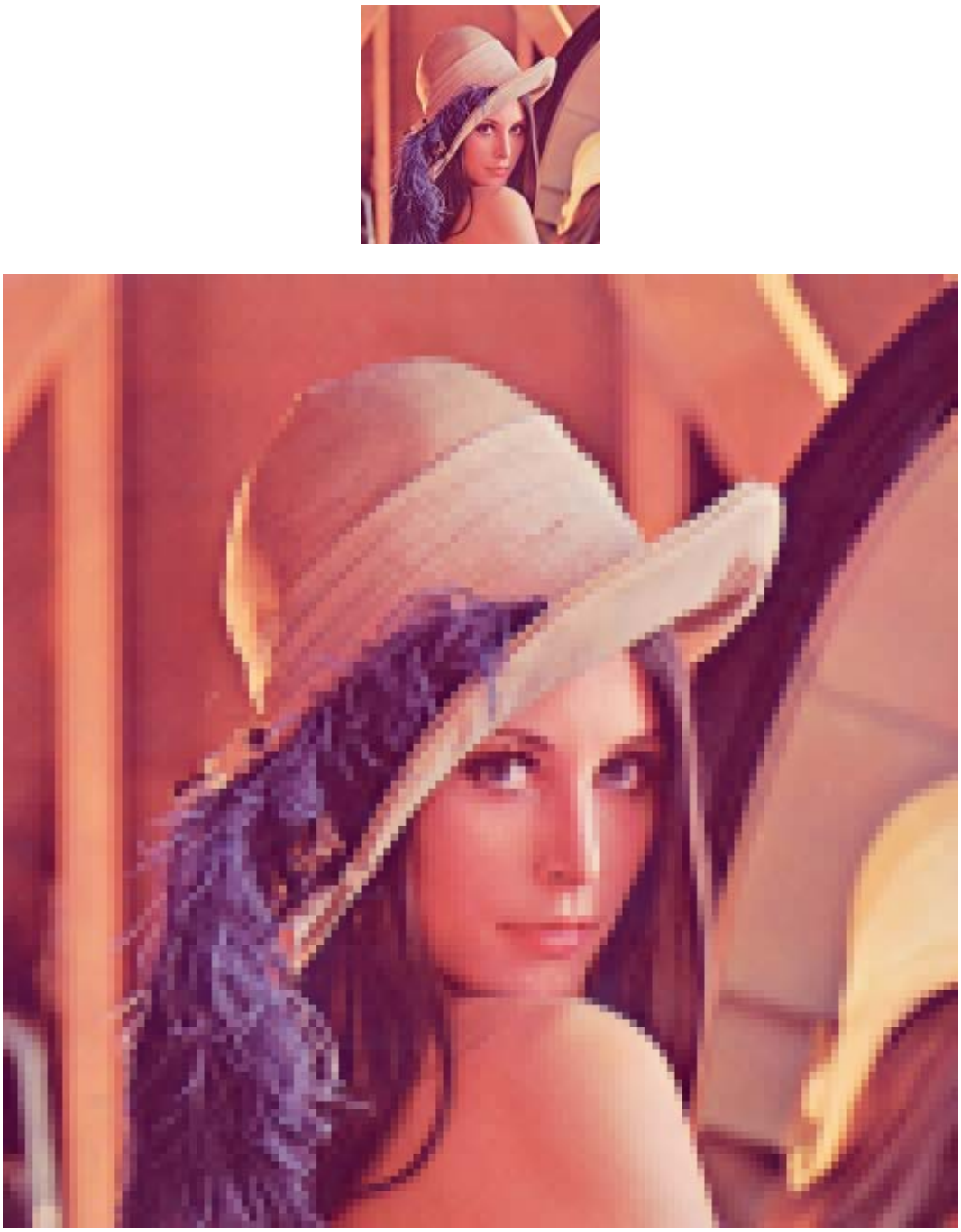


Figure 4.7: Lena Image. The original lena image and magnified 4× with Pixel Replication to show coarseness of data.

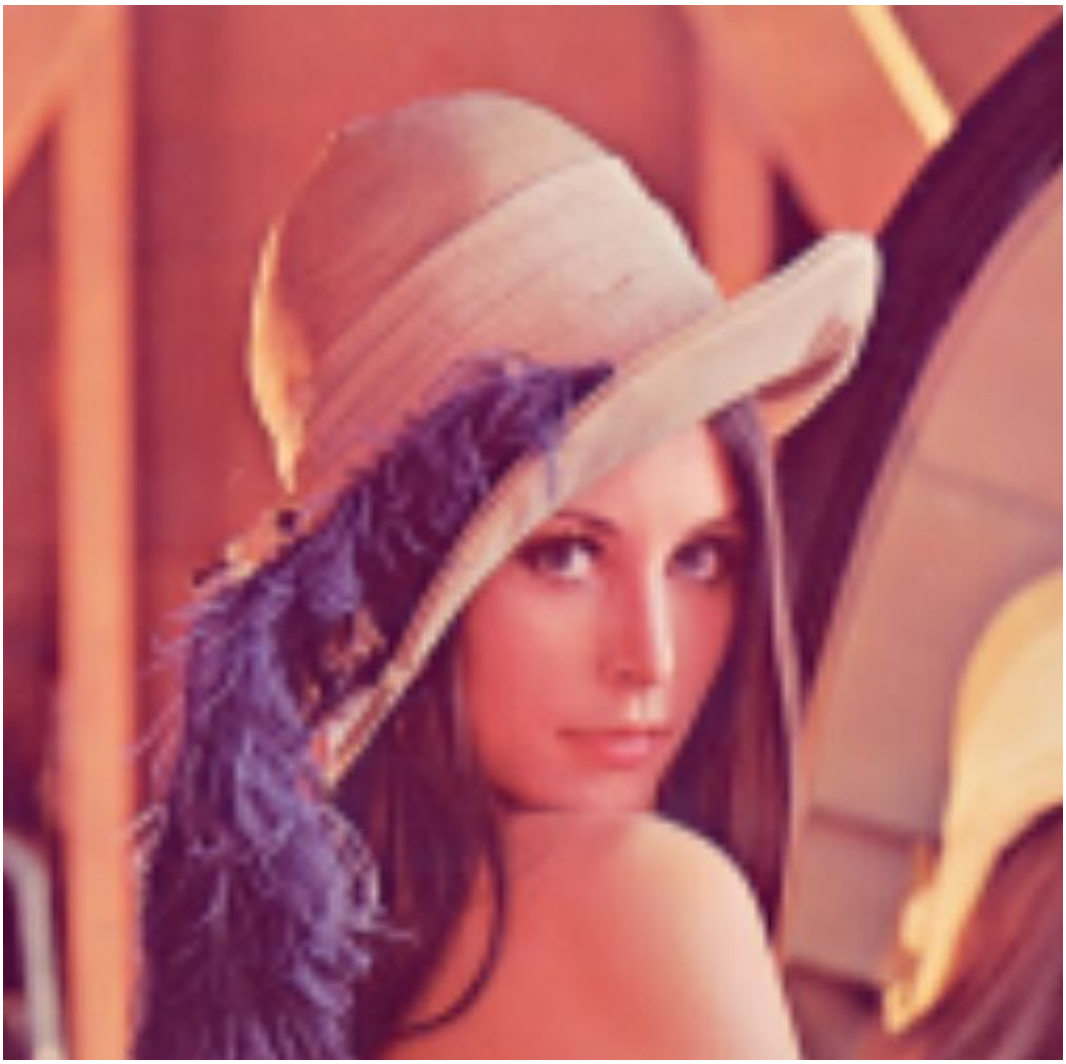


Figure 4.8: The Lena image magnified 4× using Bilinear Interpolation

Figure 4.9: The Lena image magnified 4× using Bicubic Interpolation

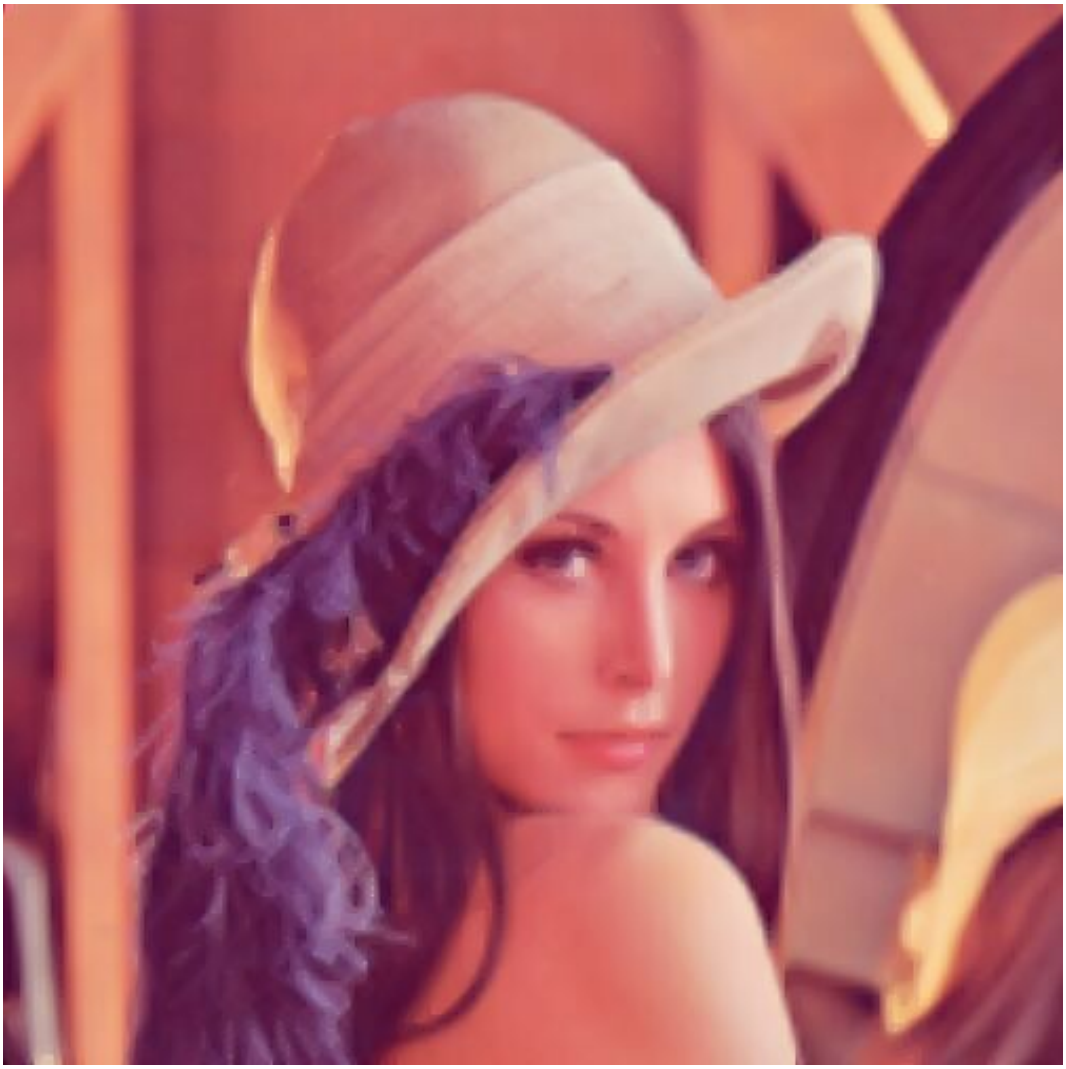


Figure 4.10: The Lena image magnified 4× using Level-Set Reconstruction

Figure 4.11: The Lena image magnified 4× using Constraint-Based Interpolation. This image has smooth, sharp edges along hat, shoulder, and cheek.

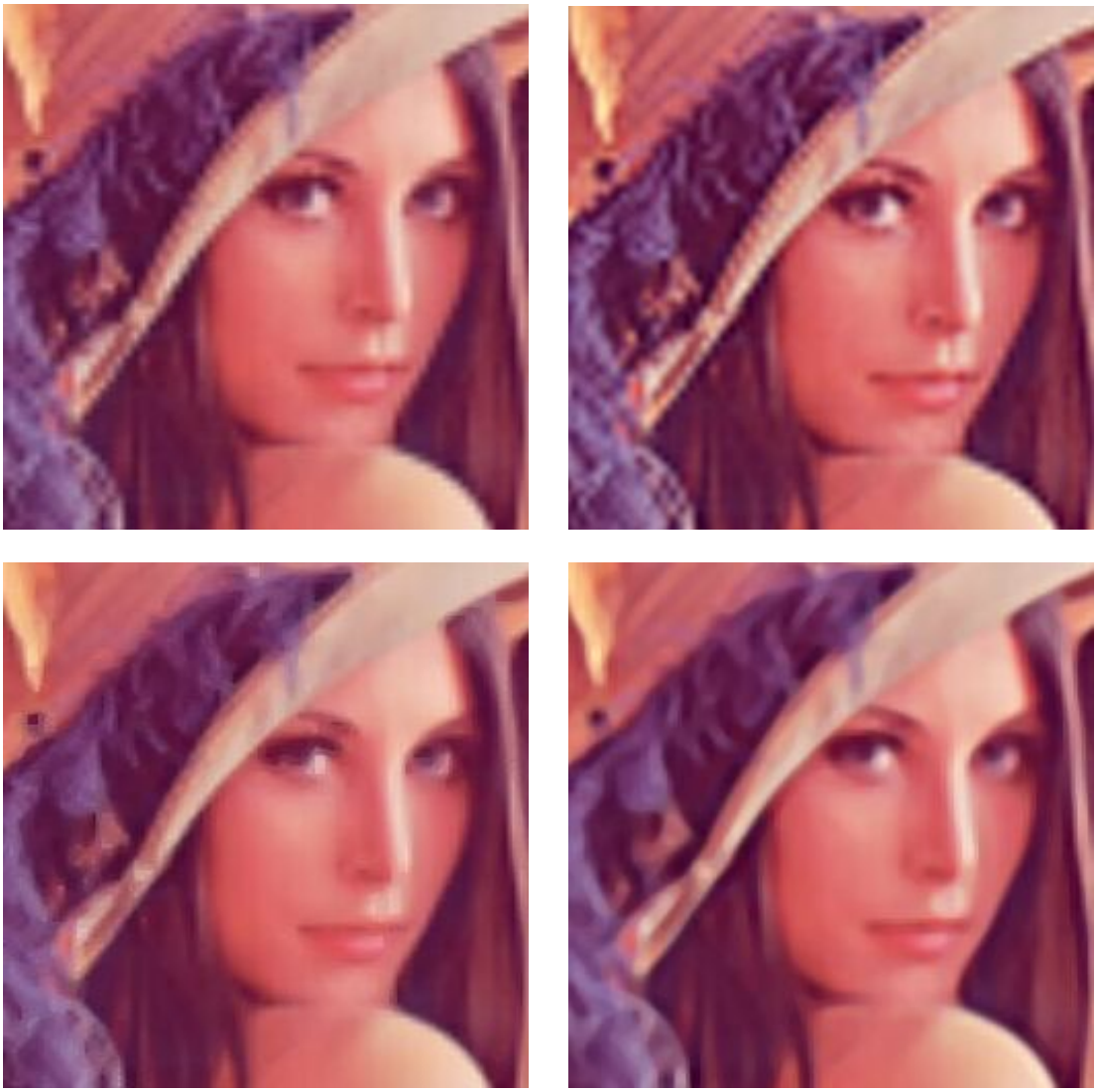


Figure 4.12: Lena image comparisons. Clockwise from Upper Left: Bilinear Interpolation, Bicubic Interpolation, Constraint-Based Interpolation, Level-Set Reconstruction.

Figure 4.13: Monarch Image. The original monarch image and magnified 3× with Pixel Replication to show coarseness of data.

Figure 4.14: The Monarch image magnified 3× using Bilinear Interpolation

Figure 4.15: The Monarch image magnified 3× using Bicubic Interpolation

Figure 4.16: The Monarch image magnified 3× using Level-Set Reconstruction

Figure 4.17: The Monarch image magnified 3× using Constraint-Based Interpolation. This image has freedom from artifacts such as jaggies, which were quite noticeable with the magnification produced with Bilinear Interpolation.

Figure 4.18: Monarch image comparisons. Clockwise from Upper Left: Bilinear Interpolation, Bicubic Interpolation, Constraint-Based Interpolation, Level-Set Reconstruction.

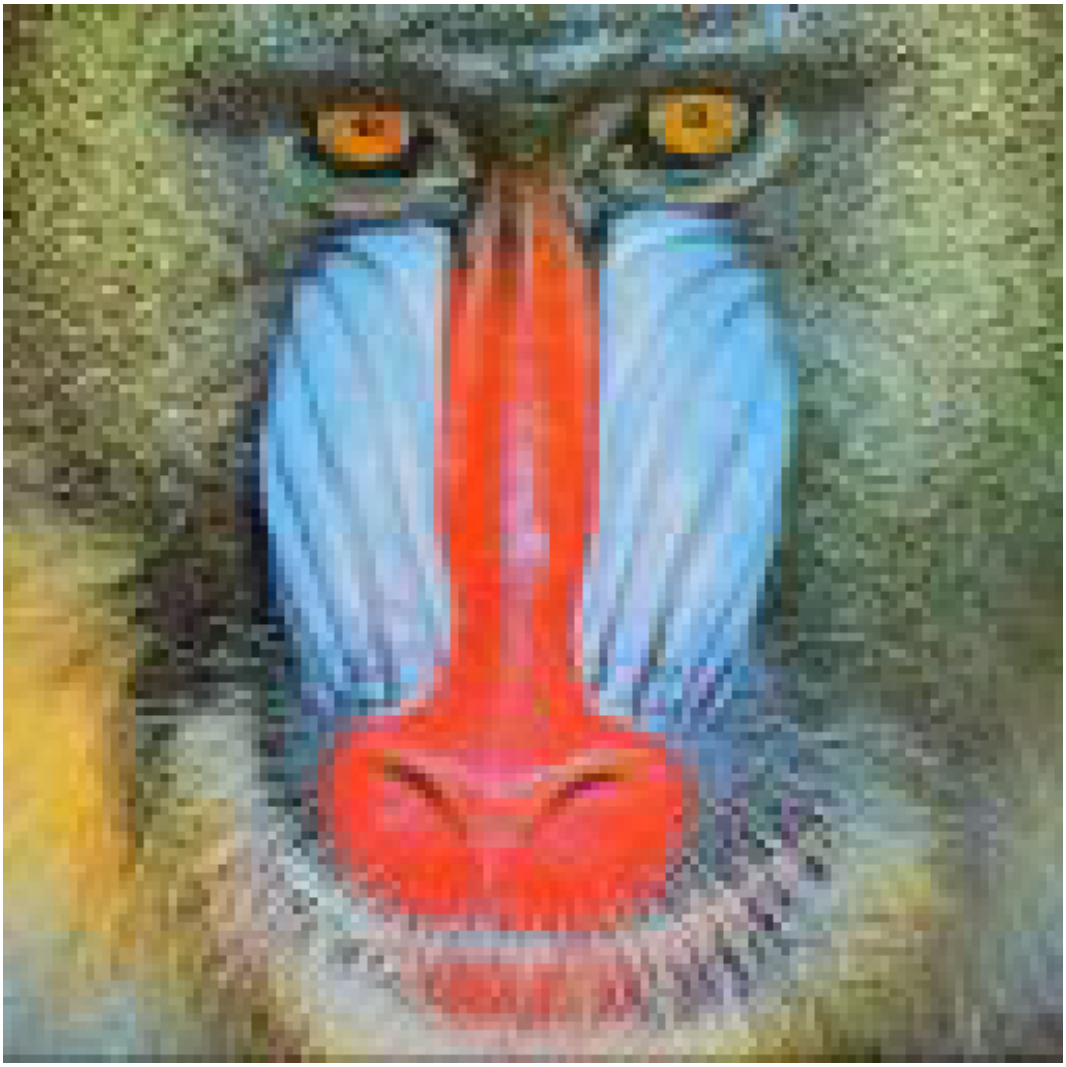


Figure 4.19: Baboon Image. The original baboon image and magnified 4× with Pixel Replication to show coarseness of data.

Figure 4.20: Baboon image magnified 4× using Bilinear Interpolation

Figure 4.21: Baboon image magnified 4× using Bicubic Interpolation

Figure 4.22: The Baboon image magnified 4× using Level-Set Reconstruction

Figure 4.23: The Baboon image magnified 4× using Constraint-Based Interpolation. This image has no prevalent artifacts such as jaggies along the nose.

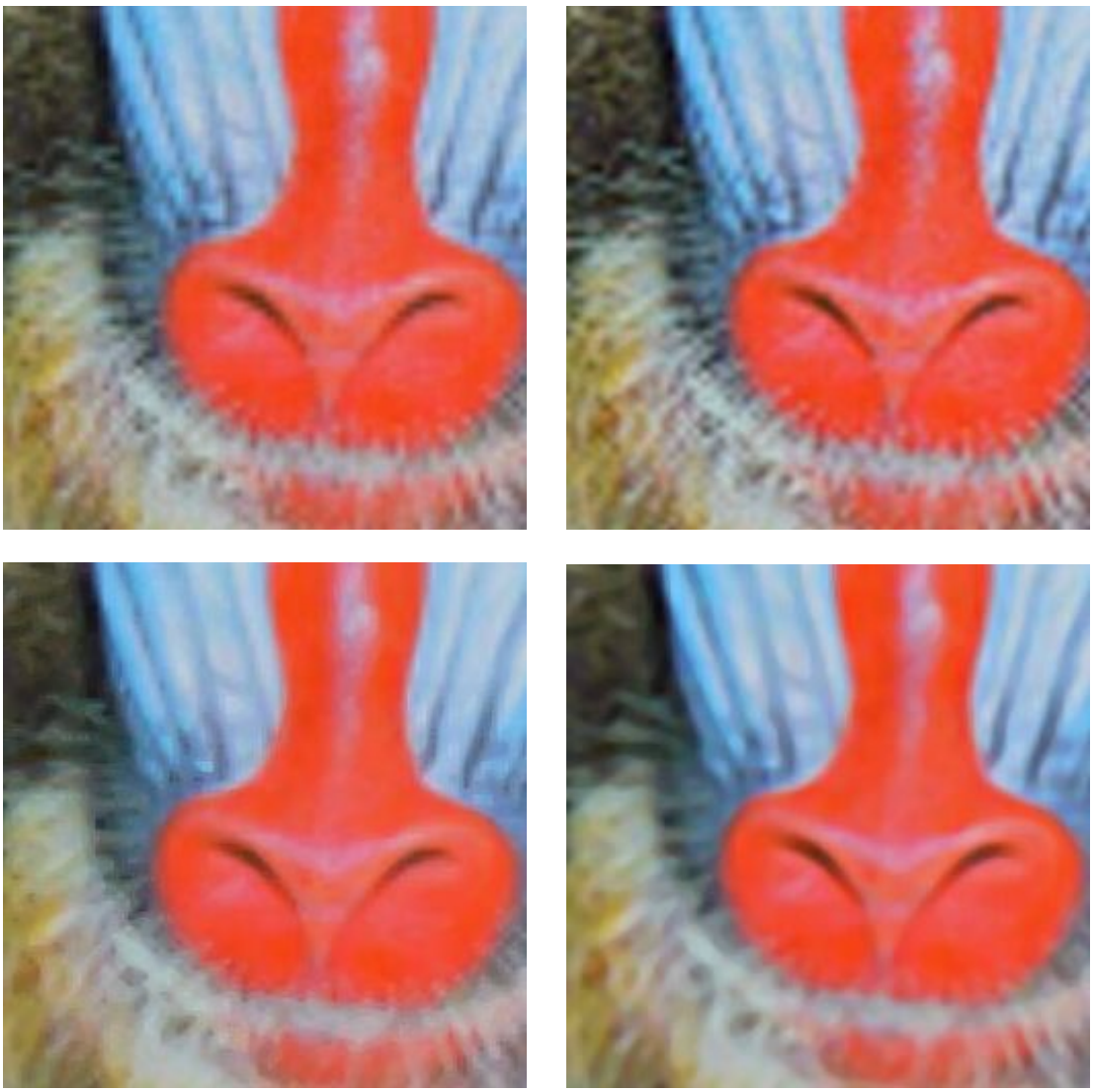


Figure 4.24: Baboon image comparisons. Clockwise from Upper Left: Bilinear Interpolation, Bicubic Interpolation, Constraint-Based Interpolation, Level-Set Reconstruction.

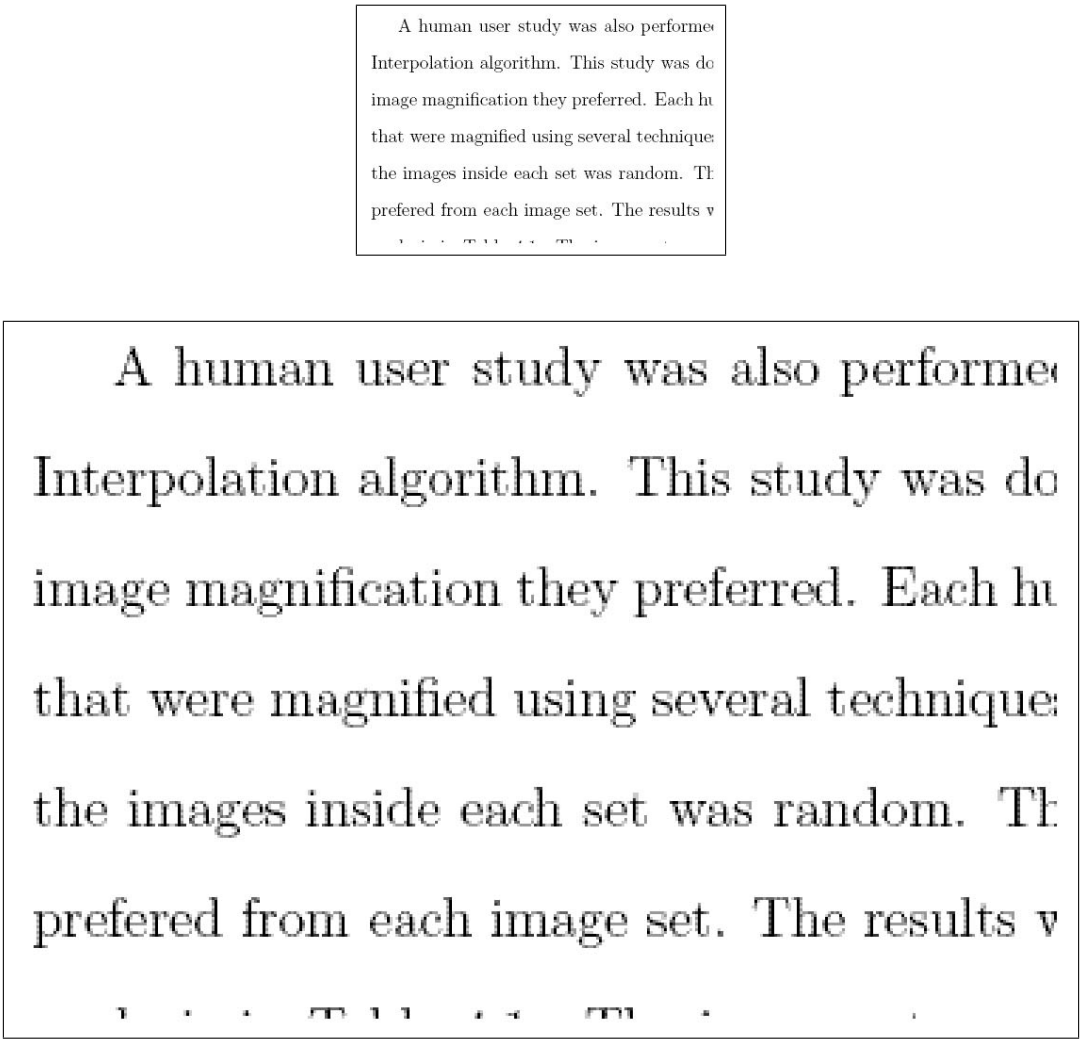


Figure 4.25: Text Image. The original text image and magnified 3× with Pixel Replication to show coarseness of data.

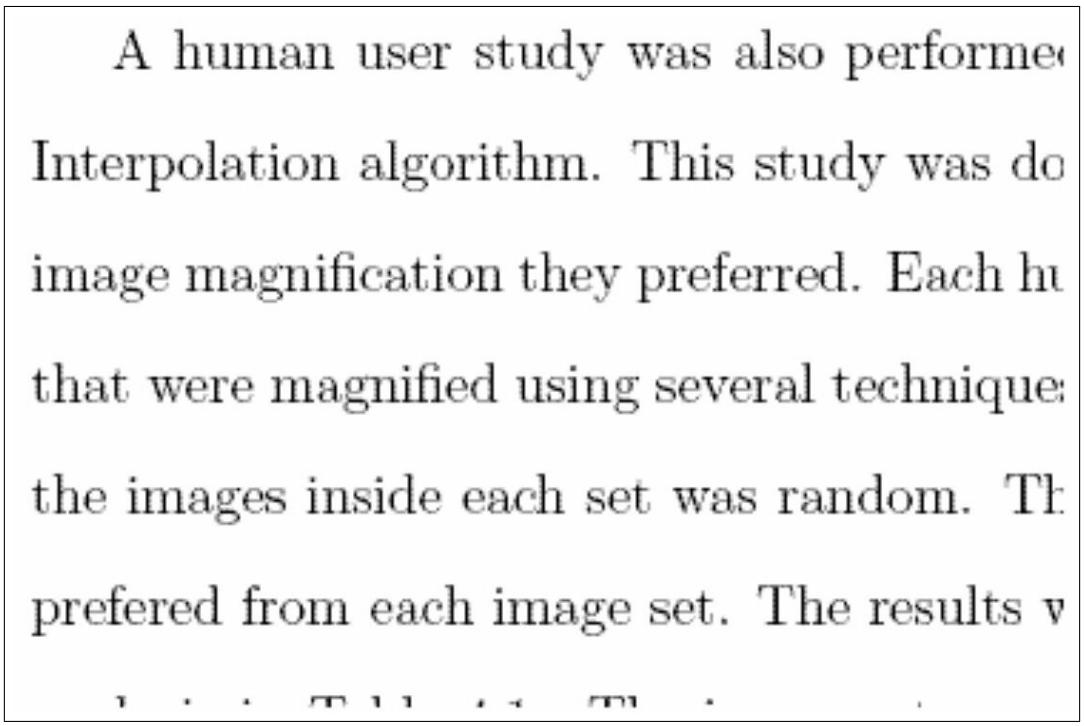


Figure 4.26: Text image magnified 3× using Bilinear Interpolation

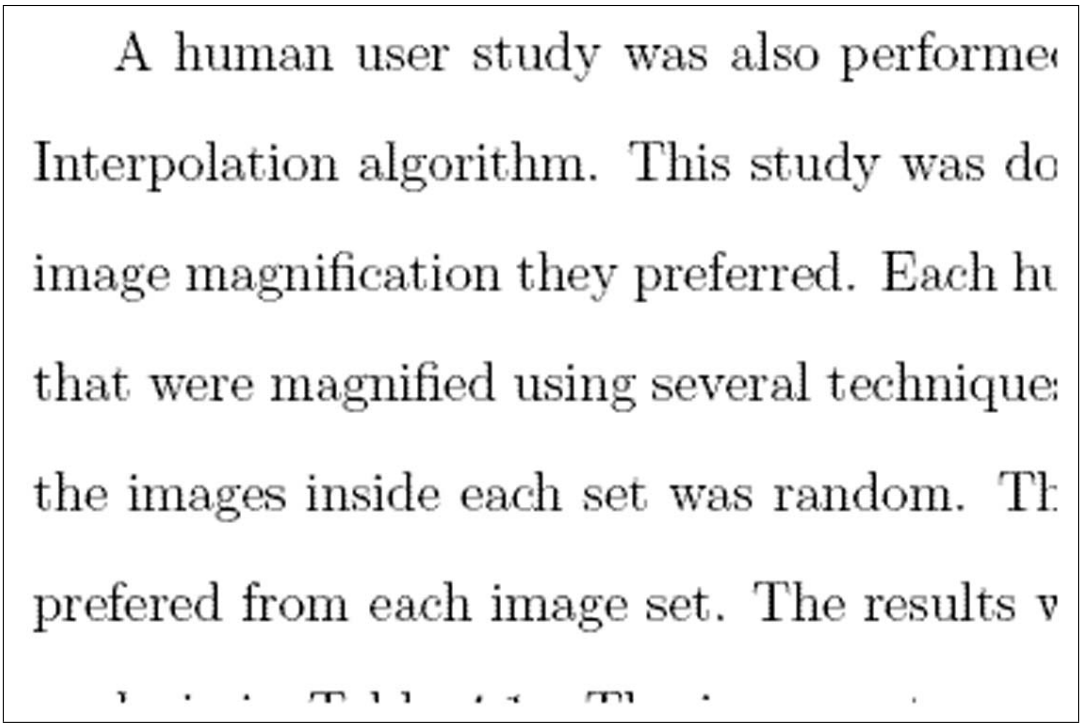


Figure 4.27: Text image magnified 3× using Bicubic Interpolation

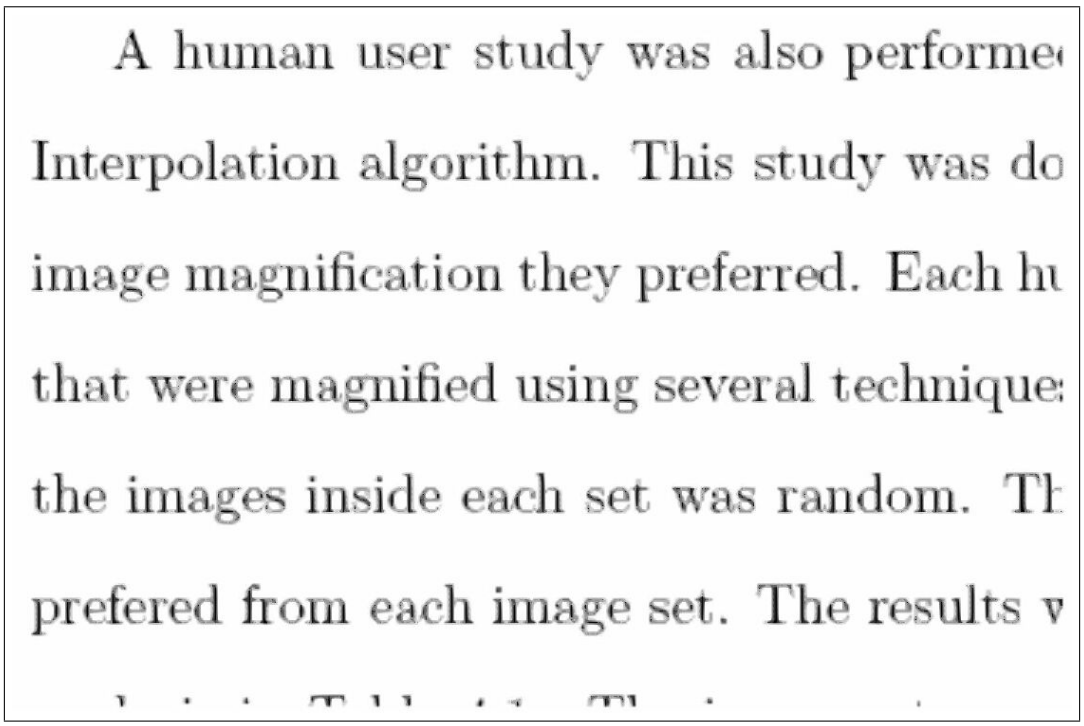


Figure 4.28: The Text image magnified 3× using Level-Set Reconstruction

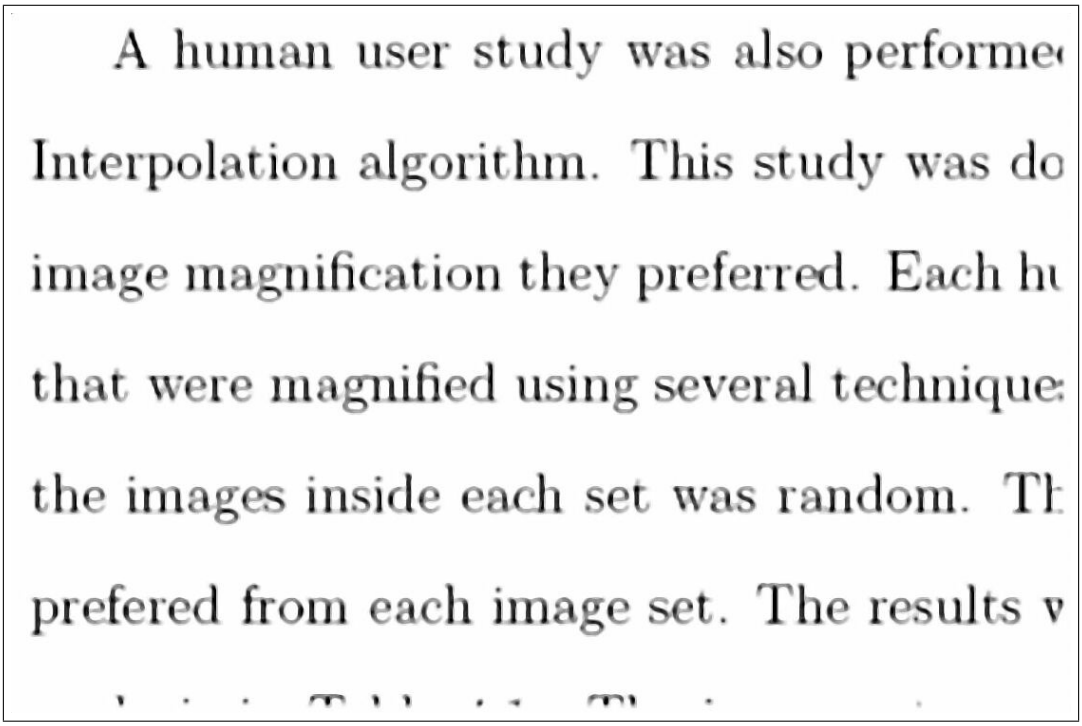


Figure 4.29: The Text image magnified 3× using Constraint-Based Interpolation. Text is sharp with no aliasing along edges.

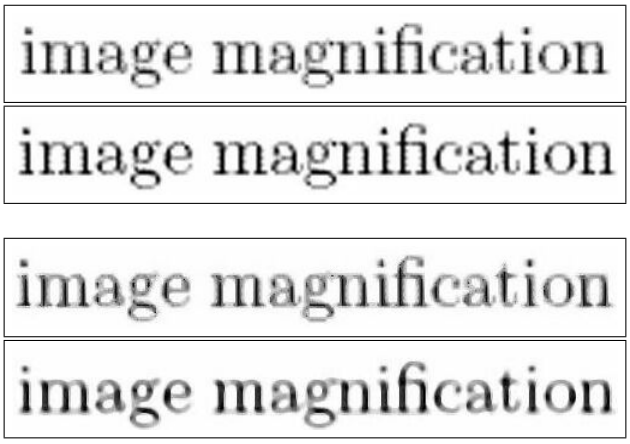


Figure 4.30: Text image comparisons. Top to Bottom: Bilinear Interpolation, Bicubic Interpolation, Level-Set Reconstruction, Constraint-Based Interpolation.

Image	Replication	Bilinear	Level-Set Reconstruction	Constraint-Based
Frymire	0	0	4	36
Lena	-	2	0	38
Monarch	-	0	2	38
Zebra	-	8	0	32

Table 4.1: Human User Study Results. As can be seen from the results, the overall prefered algorithm is Constraint-Based Interpolation.

4.2 Human User Study

A human user study was also performed as an analysis of the Constraint-Based Interpolation algorithm. This study was done by asking 40 random observers which image magnification they preferred. Each human user was shown four types of images that were magnified using several techniques. The presentation of the image sets and the images inside each set was random. The user was then asked which image they prefered from each image set. The results were then recorded, and are tabulated for analysis in Table 4.1. The image sets are included in Appendix A. Only sections of each image was shown in order to accomodate printing the images on a single piece of paper, due to the typical $8\times$ magnification that was performed. Also, each particular section was shown due to the large amount of edge content that was in each section.

As can be easily seen from Table 4.1, Constraint-Based Interpolation was the overall prefered algorithm for image magnification. These findings indicate that the Constraint-Based Interpolation algorithm produces magnifications that are more visually appealing than magnifications produced using common magnification techniques.

4.3 Quantitative Analysis

Five different quantitative analysis measurements were used on four real-world images. In order to obtain these measures, an image was first downsampled by a factor of four. This lower-resolution image was then magnified by a factor of four using a variety of magnification techniques, and then compared with the original image. Three of the measurements refelect the accuracy of the magnification. These measurements are *Mean Squared Error*, *Mean Absolute Error*, and *Cross-correlation Coefficient* [8]. The other two measurements give a raw measurement of the overall curvature in the image. These curvature measurements are calculated by using *Mean Contour Curvature* [2] and *Mean Weighted Contour Curvature*. Results of the quantitative analysis are shown in Table 4.2.

The manner in which the images are initially downsampled is also worth noting since we are downsampling the image, magnifying this lower resolution image and comparing it to the original image. Depending on the way the original image is mapped to the lower resolution image, the comparison of the magnified and original image can be inaccurate, since the image can experience sub-pixel shifts. In order to ensure that the image isn't shifted when comparing the magnification with the original, we used our own downsampling algorithm instead of a third party program such as Photoshop. The downsampling was realized by blurring the original image with a box filter the size of the magnification factor. For instance, the images were downsampled by a magnification factor of three. Thus, the original image was blurred using a 3×3 box filter. Then, every third pixel was simply used as the low resolution image, having (0,0) mapped to (0,0), (3,3) mapped to (1,1), etc. This ensures that no sub-pixel shifting occurs between the magnified image and the original image.

Image	Measure	Repl.	Bilinear	Bicubic	Quad	Level-Set	Con-Based	3x3 Box
Lena	MSE	314.25	151.64	123.48	151.79	170.30	159.56	107.29
	MAE	9.75	7.25	6.39	7.34	7.56	7.42	6.08
	CCC	0.92	0.96	0.97	0.97	0.96	0.96	0.97
	MCC	0.67	0.46	0.46	0.61	0.52	0.31	0.38
	MWCC	0.06	0.04	0.03	0.05	0.05	0.03	0.02
Face	MSE	100.89	38.74	28.37	39.90	53.37	41.36	36.14
	MAE	5.55	3.33	2.86	3.50	3.88	3.51	3.50
	CCC	0.99	0.99	0.99	0.99	0.99	0.99	0.99
	MCC	0.61	0.37	0.35	0.43	0.35	0.23	0.25
	MWCC	0.03	0.02	0.01	0.02	0.02	0.01	0.01
Monarch	MSE	278.44	143.38	109.74	143.08	199.65	170.56	131.56
	MAE	7.21	5.64	4.84	5.75	6.22	5.97	5.53
	CCC	0.91	0.96	0.96	0.96	0.94	0.95	0.96
	MCC	0.67	0.40	0.39	0.46	0.44	0.27	0.29
	MWCC	0.04	0.02	0.02	0.03	0.03	0.02	0.02
Baboon	MSE	604.14	493.67	473.74	495.52	544.05	563.92	480.61
	MAE	17.74	16.25	15.85	16.28	17.02	17.33	16.14
	CCC	0.82	0.85	0.86	0.86	0.83	0.83	0.86
	MCC	0.73	0.72	0.62	0.85	0.53	0.30	0.53
	MWCC	0.15	0.11	0.09	0.13	0.09	0.05	0.05

Table 4.2: Results of performing the quantitative analysis. MSE = Mean Squared Error. MAE = Mean Absolute Error. CCC = Cross-Correlation Coefficient. MCC = Mean Contour Curvature. MWCC = Mean Weighted Contour Curvature. As can be seen from the results, Constraint-Based interpolation produces similar accuracy measurements compared with Bilinear Interpolation and Quad-Based Interpolation, but with greatly decreased contour curvature than any interpolation.

4.4 Accuracy Measurements

The quantitative measurements were gathered by comparing the magnified image with the original image using several metrics which are *Mean Squared Error*, *Mean Absolute Error*, and *Cross-correlation Coefficient*. The equations are given in equations 4.1 through 4.3.

$$\textit{Mean Squared Error} = \frac{\Sigma_{x=1}^{M} \Sigma_{y=1}^{N} (\hat{\mathrm{I}}(x,y) - I(x,y))^2}{MN} \tag{4.1}$$

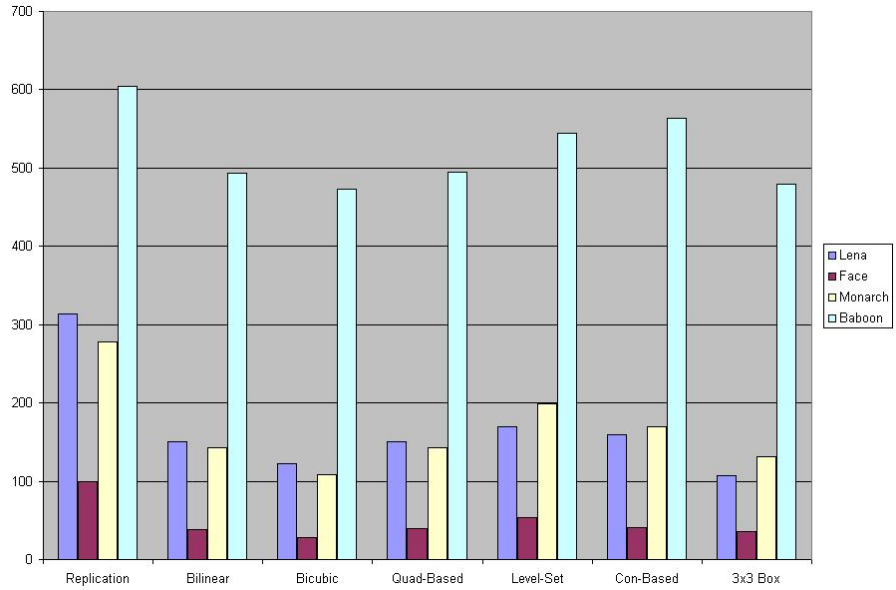


Figure 4.31: Mean Squared Error results

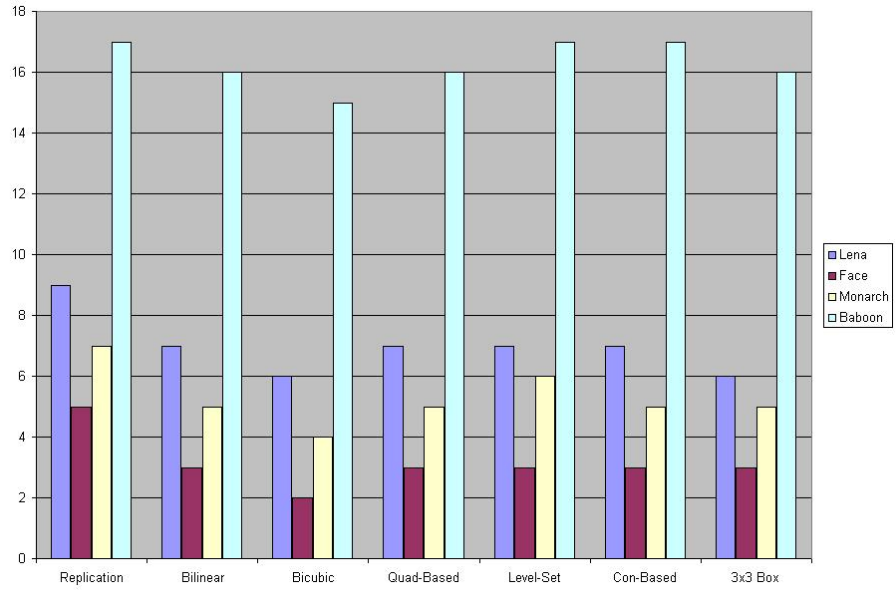


Figure 4.32: Mean Absolute Error results

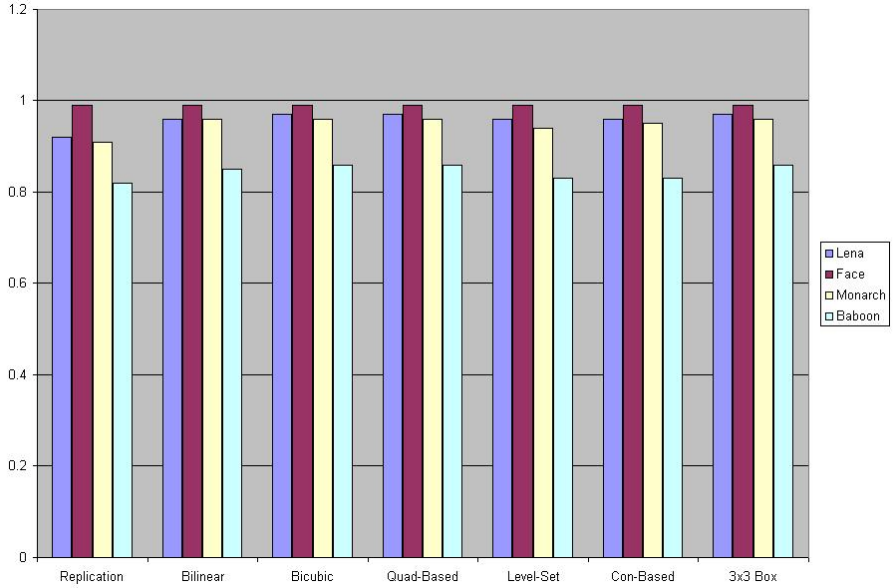


Figure 4.33: Cross-Correlation Coefficient results

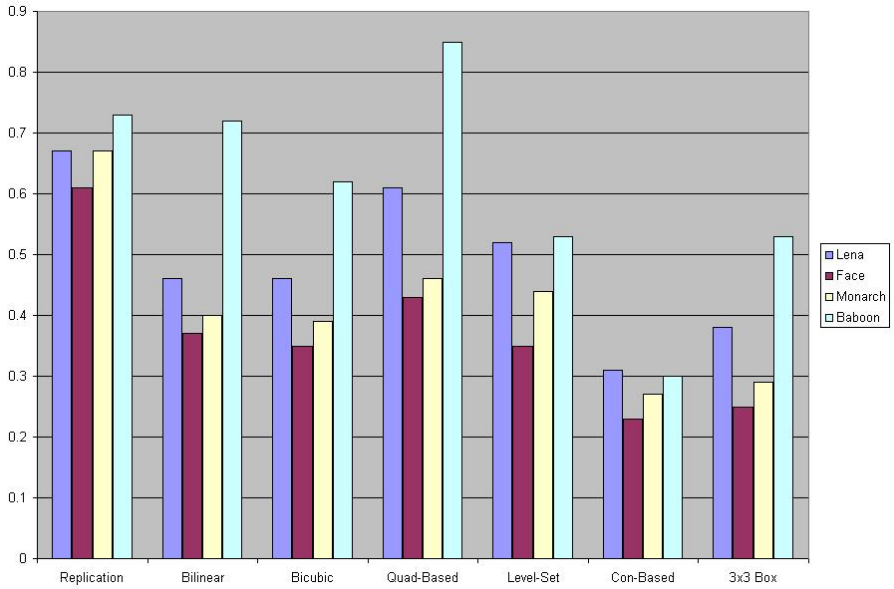


Figure 4.34: Mean Contour Curvature results

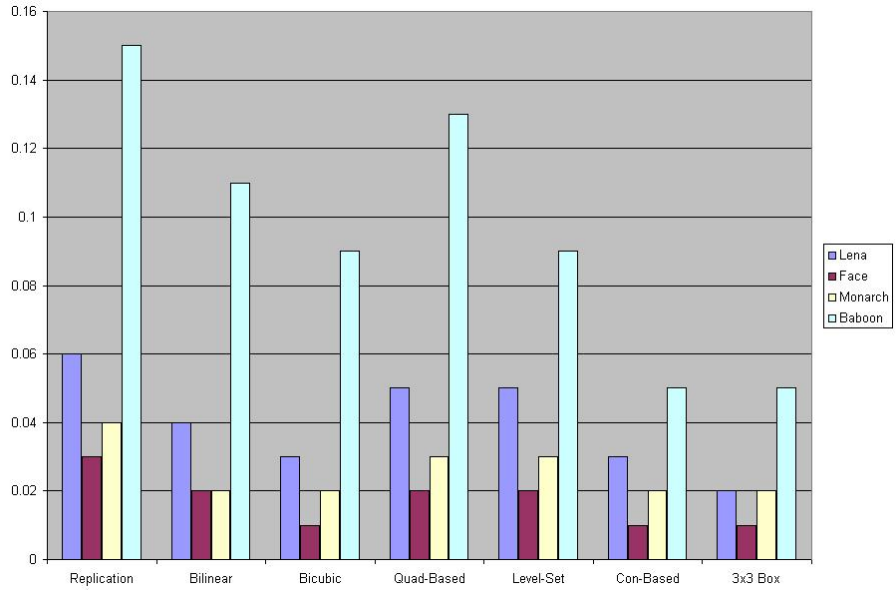


Figure 4.35: Mean Weighted Contour Curvature results

$$Mean\ Absolute\ Error = \frac{\Sigma_{x=1}^{M}\Sigma_{y=1}^{N}|\hat{\mathrm{I}}(x,y) - I(x,y)|}{MN} \tag{4.2}$$

$$Cross\text{-}Correlation\ Coefficient = \left| \frac{(\Sigma_{x=1}^{M}\Sigma_{y=1}^{N}\hat{\mathrm{I}}(x,y)I(x,y) - nab)}{((\Sigma_{x=1}^{M}\Sigma_{y=1}^{N}\hat{\mathrm{I}}^{2}(x,y) - na^{2})(\Sigma_{x=1}^{M}\Sigma_{y=1}^{N}I^{2}(x,y) - nb^{2}))^{\frac{1}{2}}} \right| \tag{4.3}$$

where $\hat{\mathrm{I}}$ is the magnified image, I is the original image, n is the total number of pixels, and a and b are the corresponding average pixel value in each image. The Cross-Correlation Coefficient equation is further explained in "A New Edge-Adaptive Zooming Algorithm for Digital Images" [8]. It should also be noted that lower values of mean squared and absolute error indicate higher accuracy while values closer to one indicate higher accuracy in cross-correlation coefficient values.

As shown by the human user study and the example images, the images used in the quantitative analysis are very different from each other. When these results are analyzed in conjunction with the human user study and the visual inspection, it becomes clear that the accuracy measurements alone are not sufficient to determine image quality. For instance, Bilinear Interpolation consistently attains slightly better accuracy scores than Constraint-Based Interpolation. However, the human user study indicates that Constraint-Based Interpolation is selected nearly 14 times as much as Bilinear Interpolation. Visual inspection also clearly shows obvious artifacts such as blurring and jaggies in images produced with Bilinear Interpolation. Therefore, accuracy measurements are not enough to indicate image quality.

As shown in the analysis, Constraint-Based Interpolation has higher Mean Squared Error than Bicubic Interpolation. Errors can be introducted into the Constraint-Based Interpolation algorithm by the point-spread function that is selected. If this point-spread function is different from the sampling function that was used to create the low-resolution image, then Constraint-Based Interpolation can either over or under sharpen, and thus errors are introduced into the interpolation. Therefore, Constraint-Based Interpolation can have higher Mean Squared Error than Bicubic Interpolation, yet it should be noted that images produced using Constraint-Based Interpolation are selected more often in the human user study than those produced by Bicubic Interpolation.

The accuracy measurements also indicate that Constraint-Based Interpolation creates more accurate magnifications than Level-Set Image Reconstruction. This leads to the conclusion that the Sensor model is more robust than the constraints used in Level-Set Image Reconstruction, such as Image Anchors.

4.5 Curvature Measurements

As stated in "Level-Set Image Reconstruction" [2], a magnification of an image that has jagged edges has more contour curvature than a magnification of the same image that produces smooth contours. This led to the development of a metric called Mean Contour Curvature, which has the equation:

$$Mean\ Contour\ Curvature = \frac{1}{MN}\sum_{x=1}^{M}\sum_{y=1}^{N}|\kappa_{(x,y)}| \tag{4.4}$$

where $\kappa_{(x,y)}$ is the isophote curvature at pixel (x, y). This measurement calculates the mean absolute contour curvature in an image. As was shown in Level-Set Image Reconstruction, human users selected images that had lower mean contour curvature than other images, and hence less jaggies. As an enhancement to this metric, a new metric called *Mean Weighted Contour Curvature* was created to give more weight to edges in an image, since human users tend to notice jaggies along edges in an image more than in other areas. Simply put, Mean Weighted Contour Curvature gives greater weight to areas of the image that have higher gradient magnitude (edges):

$$Mean\ Weighted\ Contour\ Curvature = \frac{1}{MN}\sum_{x=1}^{M}\sum_{y=1}^{N}|\kappa_{(x,y)}|\ ||\bigtriangledown I(x,y)|| \tag{4.5}$$

These two contour measurements imply that an image with a lower metric has less jaggies. As shown by the human user study, the images that have lower contour curvature were picked more frequently than those with higher contour curvatures. A correlation can now be assumed that humans prefer images without jaggies. Constraint-Based Interpolation, as shown by the example images, clearly does not produce jaggies.

The quantitative analysis also indicates that Constraint-Based Interpolation con-

sistently had lower curvature values than Level-Set Image Reconstruction. This is due to constraints such as Image Anchors in Level-Set Image Reconstruction which does not allow certain pixels in the image to be smoothed, thus possibly increasing the contour curvature in the image.

Chapter 5

Summary and Future Work

5.1 Summary

Images are becoming a necessary element of all types of media in today's digital age. Images are often magnified to a greater resolution for a number of reasons, including viewing, printing, or editing. This thesis has introduced a new magnification technique that produces higher quality image magnifications than previous standard techniques. This algorithm is called Contraint-Based Interpolation.

The Constraint-Based Interpolation algorithm attempts to remove many of the artifacts that are predominant in standard magnification techniques. Common artifacts include blurring, ringing, and the jaggies, or jaggedness along edges in an image. These artifacts are removed in the algorithm by using level-set smoothing and sharpening models. Accuracy is maintained by using a sensor model that continually pushes the magnification to be as close as possible to the information contained in the original lower-resolution image. By combining these three models, the Constraint-Based Interpolation algorithm is able to produce real-world magnifications that user studies

show are preferred over magnifications produced by common magnification methods. The algorithm is also robust to high magnification scale values, as demonstrated in the example images.

Level-Set Image Reconstruction [2] showed in its analysis that Level-Set Image Reconstruction was able to create magnifications that had no greater error in accuracy than common methods but lower contour curvature. As shown in the human user study, subjects preferred the images with lower contour curvature. As in Level-Set Image Reconstruction, Constraint-Based Interpolation, an enhancement to Level-Set Image Reconstruction, was able to produce even lower contour curvatures and similar accuracy measurements than even Level-Set Image Reconstruction. It was also noted that human subjects in the user study also greatly preferred images produced by Constraint-Based Interpolation over images produced with Level-Set Image Reconstruction.

In closing, this research and thesis provides a new interpolation technique that can produce quality real-world image magnifications.

5.2 Future Work

5.2.1 Sharpening Curvature Constraint Limit

This thesis used a sharpening model in order to reduce blurring, which is common in many magnification algorithms. In order to constrain this model, a sharpening curvature constraint was implemented. As an enhancement to this constraint, more research should be done to provide an upper limit on how much a particular area should be allowed to be sharpened.

5.2.2 Stopping Criteria

The Constraint-Based Interpolation algorithm is an iterative process, whose number of iterations are defined by the user. Further research should be done to analyze the iterative process and intelligently select the stopping criteria, or alter the algorithm in order to allow the algorithm to converge.

If the algorithm cannot be modified to converge, then further work should be done to determine how many iterations a particular image should need to produce a realistic image. For instance, through experimentation determine the correlation of required iterations and a metric such as Mean Contour Curvature in the low-resolution image. This ratio could then be used to automatically determine the number of iterations that the algorithm should run.

5.2.3 Arbitrary Warps

The Sensor model was created in such a way as to be able to create magnifications of arbitrary x and y magnification scales. This was done by using a supersampled kernel to allow, for example, a magnification of three times in x and five times in y. However, a further enhancement to the Constraint-Based Interpolation algorithm is to allow for any arbitrary warp, such as rotation and skew. The algorithm can then be used to enhance image warps and remove artifacts that are known to occur with such transformations.

5.2.4 Initial Interpolations and Weights

Further research on which initial interpolations produce the best higher-resolution magnifications, based on the metrics defined in chapter 4 as well as visual inspec-

tion, should be performed. This also includes attempting to find the optimal sensor, smoothing, and sharpening weights that create the lowest accuracy errors but while removing the most artifacts from the image.

Bibliography

[1] J. Allebach and P.W. Wong. Edge-directed interpolation. *IEEE International Conference on Image Processing*, 3:707–710, 1996.

[2] B. S. Morse and D. Schwartzwald. Image magnification using level-set reconstruction. *Proceedings Computer Vision and Pattern Recognition*, 333–340, 2001.

[3] R. R. Schultz and R. L. Stevenson. A Bayesian approach to image expansion for improved definition. *IEEE Transactions on Image Processing*, 3(3):233–242, 1994.

[4] J. Sun and N. Zheng and H. Tao and H. Shum. Image hallucination with primal sketch priors. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 729–736, 2003.

[5] E. Meijering. A chronology of interpolation: from ancient astronomy to modern signal and image processing. *Proceedings of the IEEE*, 90(3):319–341, 2002.

[6] K. Jensen and D. Anastassiou. Subpixel edge localization and the interpolation of still images. *IEEE Transactions on Image Processing*, 4(3):285–295, 1995.

[7] X. Li and M. T. Orchard. New edge-directed interpolation. *IEEE Transactions on Image Processing*, 10(10):1521–1527, 2001.

[8] S. Battiato and G. Gallo and F. Stanco. A new edge-adaptive zooming algorithm for digital images. *Proceedings of Signal Processing and Communications SPC*, 144–149, 2000.

[9] G. Wolberg and H. Massalin. A fast algorithm for digital image scaling. *Proceedings of Computer Graphics International*, June, 1993.

[10] D. D. Muresan and T. W. Parks. Optimal recovery approach to image interpolation. *Proceedings of the IEEE International Conference on Image Processing*, 3:848–851, 2001.

[11] D. D. Muresan and T. W. Parks. Optimal face reconstruction using training. *IEEE International Conference on Image Processing*, 3:373–376, 2002.

[12] D. D. Muresan and T. W. Parks. Adaptively quadratic (aqua) image interpolation. To appear in *IEEE Journal on Image Processing*, 2004.

[13] Q. Wang and R. Ward. A New edge-directed image expansion scheme. *IEEE International Conference on Image Processing*, 3:899–902, 2001.

[14] T. E. Boult and G. Wolberg. Local image reconstruction and sub-pixel restoration algorithms. *Computer Graphics and Image Processing: Graphical Models and Image Processing*, 55(1):63–77, 1993.

[15] X. Yu and B. Morse and T. W. Sederberg. Image reconstruction using data-dependent triangulation. *IEEE Computer Graphics and Applications*, 21(3):62–69, 2001.

[16] L. Alvarez and P.L. Lions and J.M. Morel. Image Selective Smoothing and Edge Detection by Nonlinear Diffusion (II). *SIAM Journal of Numerical Analysis*, 29:845–866, 1992.

[17] A. Hertzmann and C.E. Jacobs and N. Oliver and B. Curless and D.H. Salesin. Image Analogies. *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, 327–340, 2001.

[18] S. Osher and J.A. Sethian. Fronts propagating with curvature dependent speed: Algorithms based on Hamilton-Jacobi formulation. *Journal of Computational Physics*, 12–49, 1988.

[19] S. Osher and L. Rudin. Feature-Oriented Image Enhancement using Shock Filters. *SIAM Journal of Numerical Analysis*, 27:919–940, 1990.

[20] C. Duchon. Lanczos filtering in one and two dimensions. *Journal of Applied Meteorology*, 18:1016–1022, 1979.

Appendix A

User Study Images

A.1 Lena
A.2 Monarch
A.3 Frymire
A.4 Zebra

Figure A.1: Original Lena image showing section that was magnified

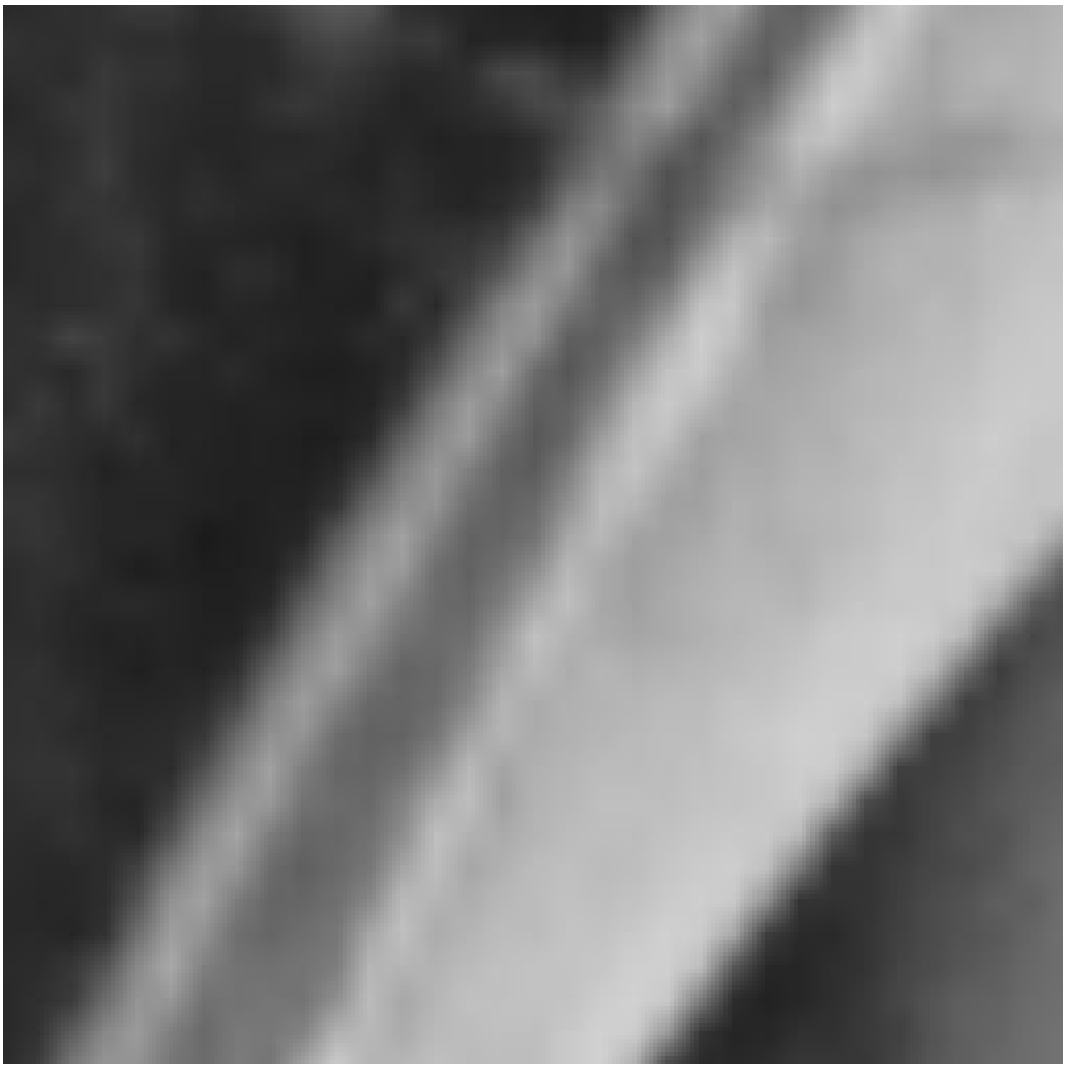


Figure A.2: Lena image, 8x Bilinear Interpolation

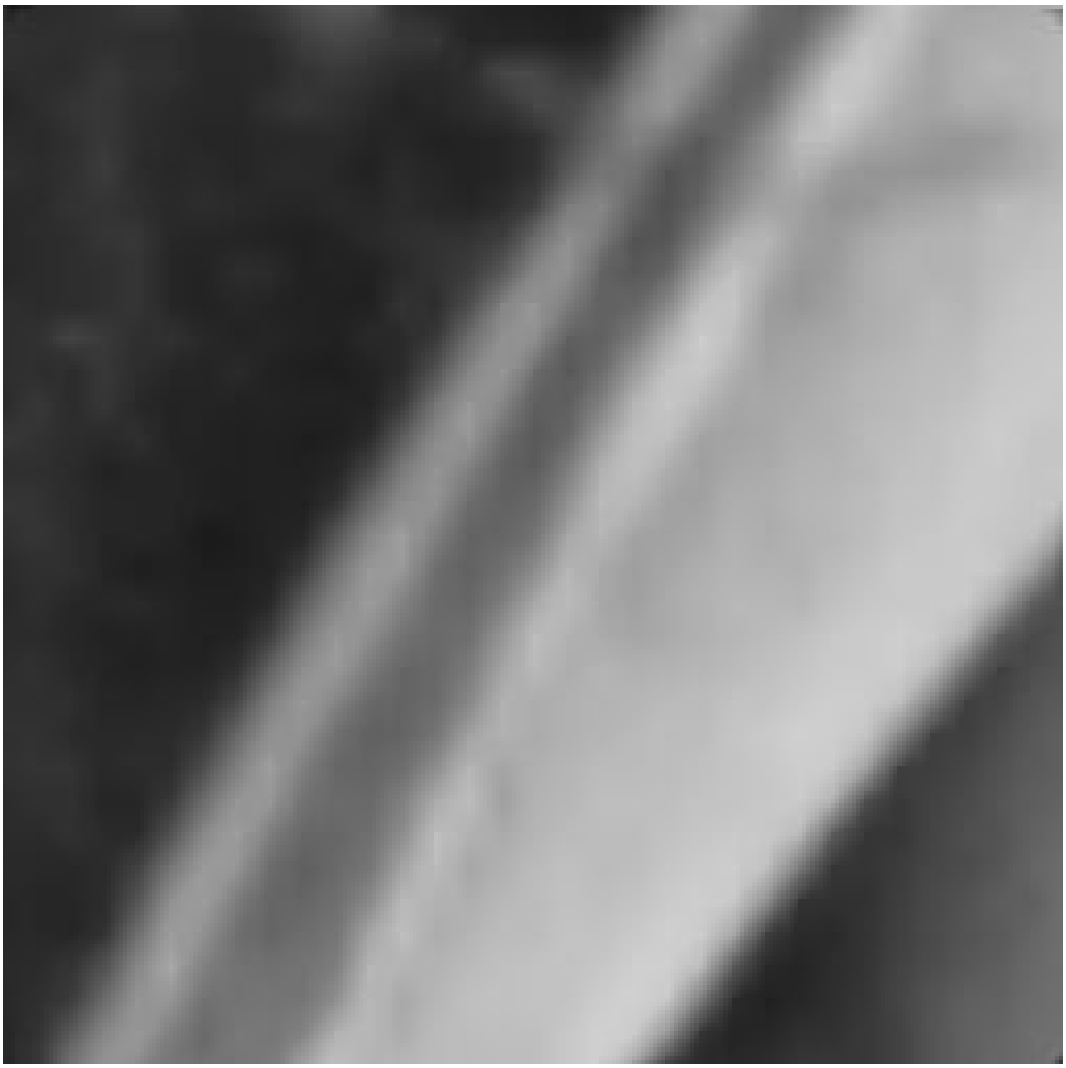


Figure A.3: Lena image, 8x Level-set Reconstruction

Figure A.4: Lena image, 8x Constraint-Based Interpolation

Figure A.5: Original Monarch image showing section that was magnified

Figure A.6: Monarch image, 4x Bilinear Interpolation

Figure A.7: Monarch image, 4x Level-set Reconstruction

Figure A.8: Monarch image, 4x Constraint-Based Interpolation

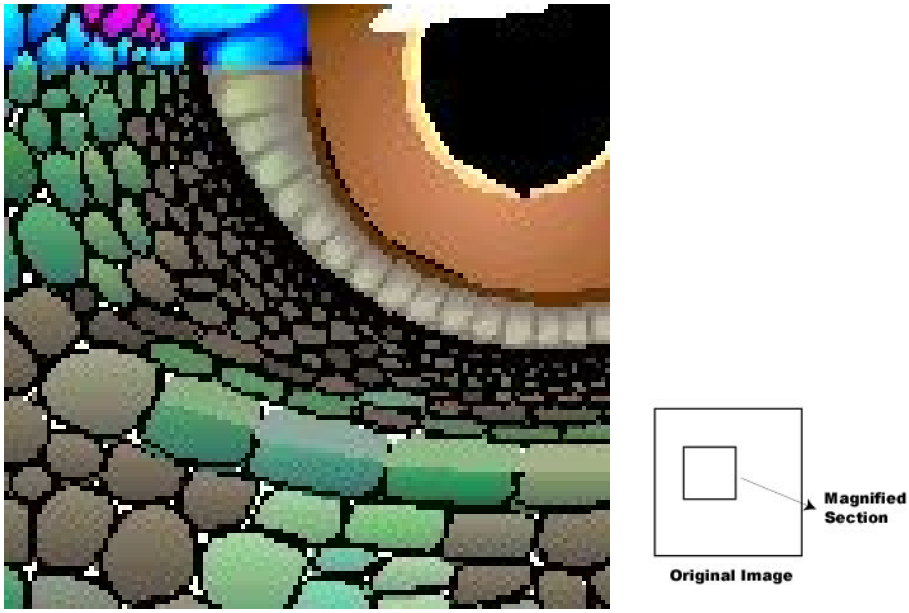


Figure A.9: Original Frymire image showing section that was magnified

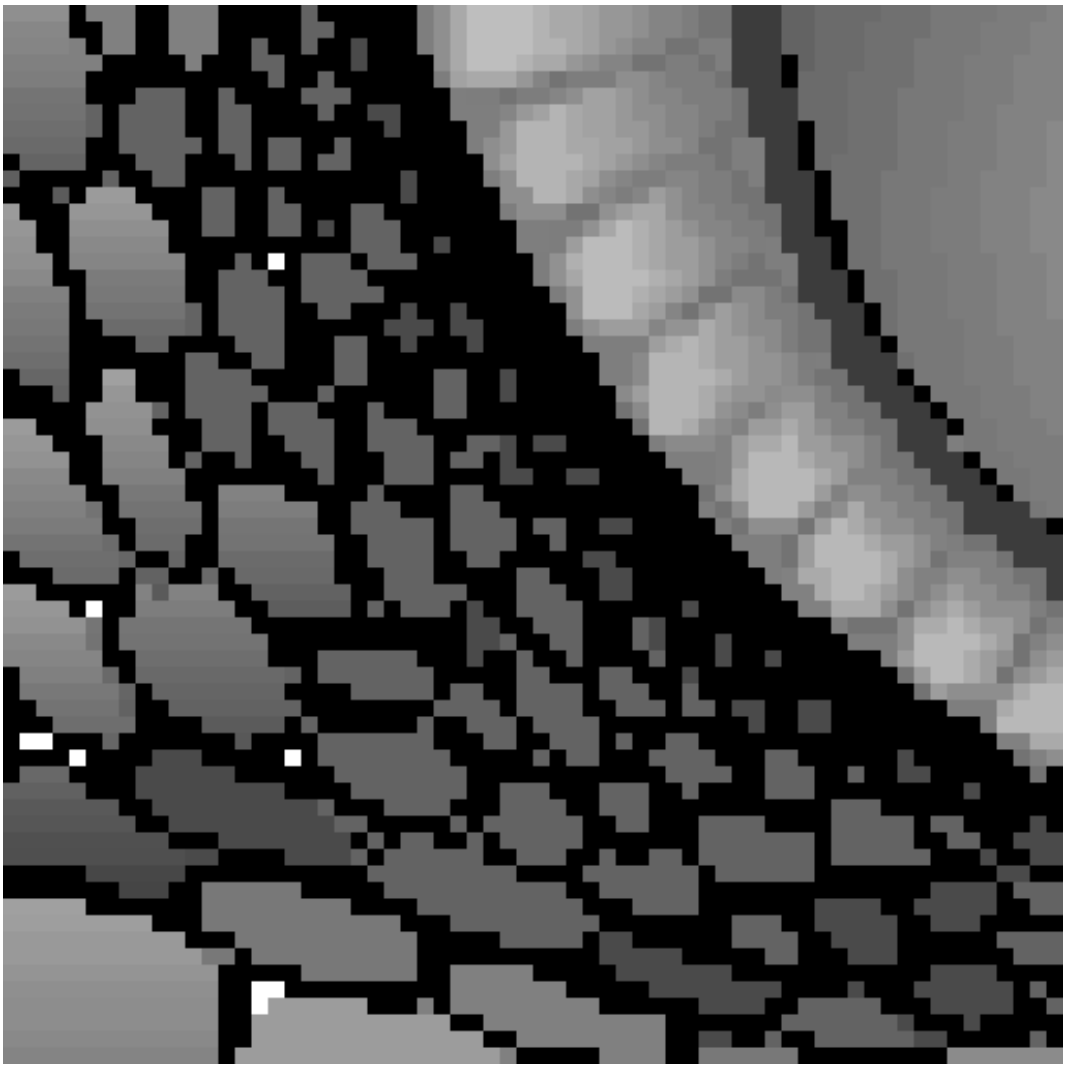


Figure A.10: Frymire image, 8x Pixel Replication

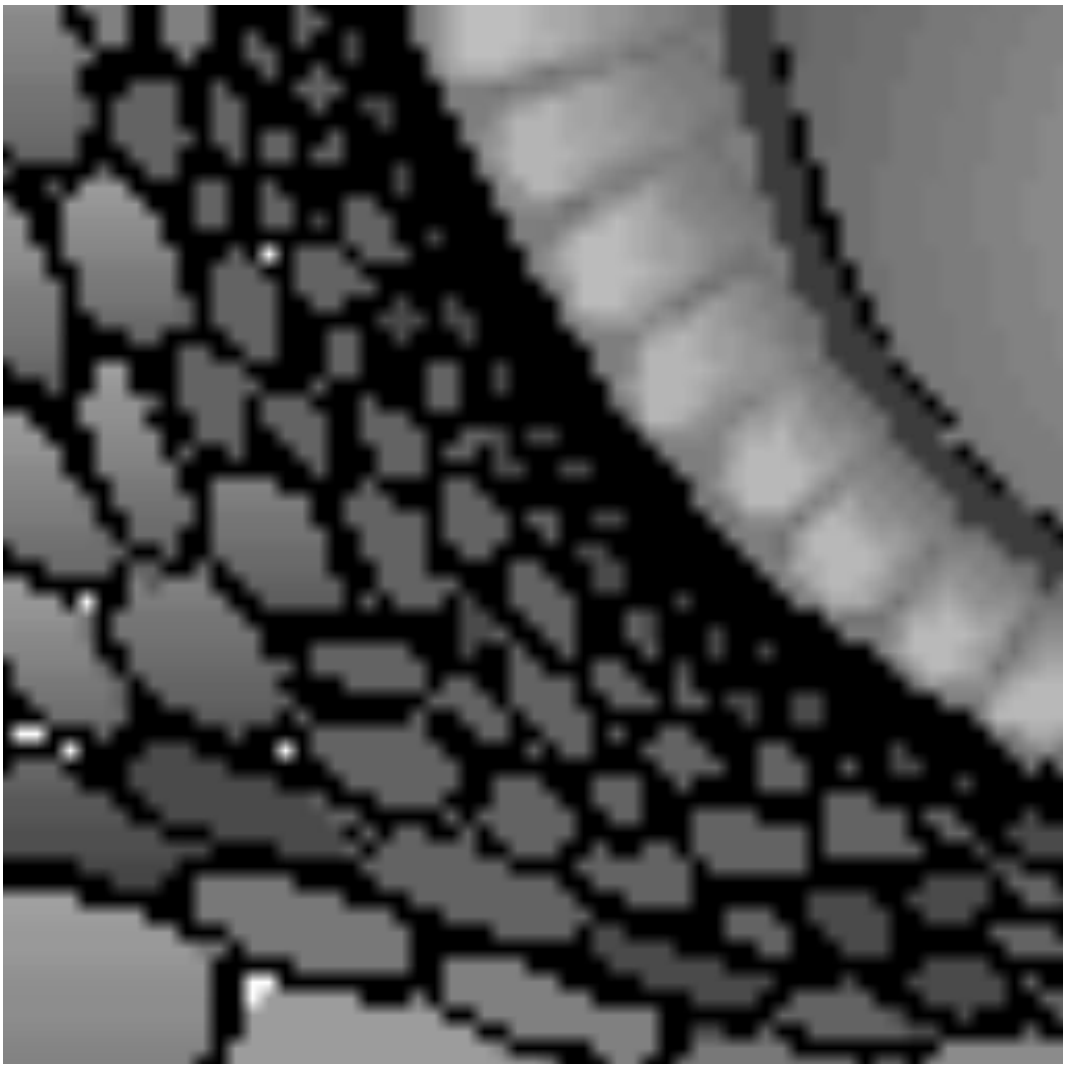


Figure A.11: Frymire image, 8x Bilinear Interpolation

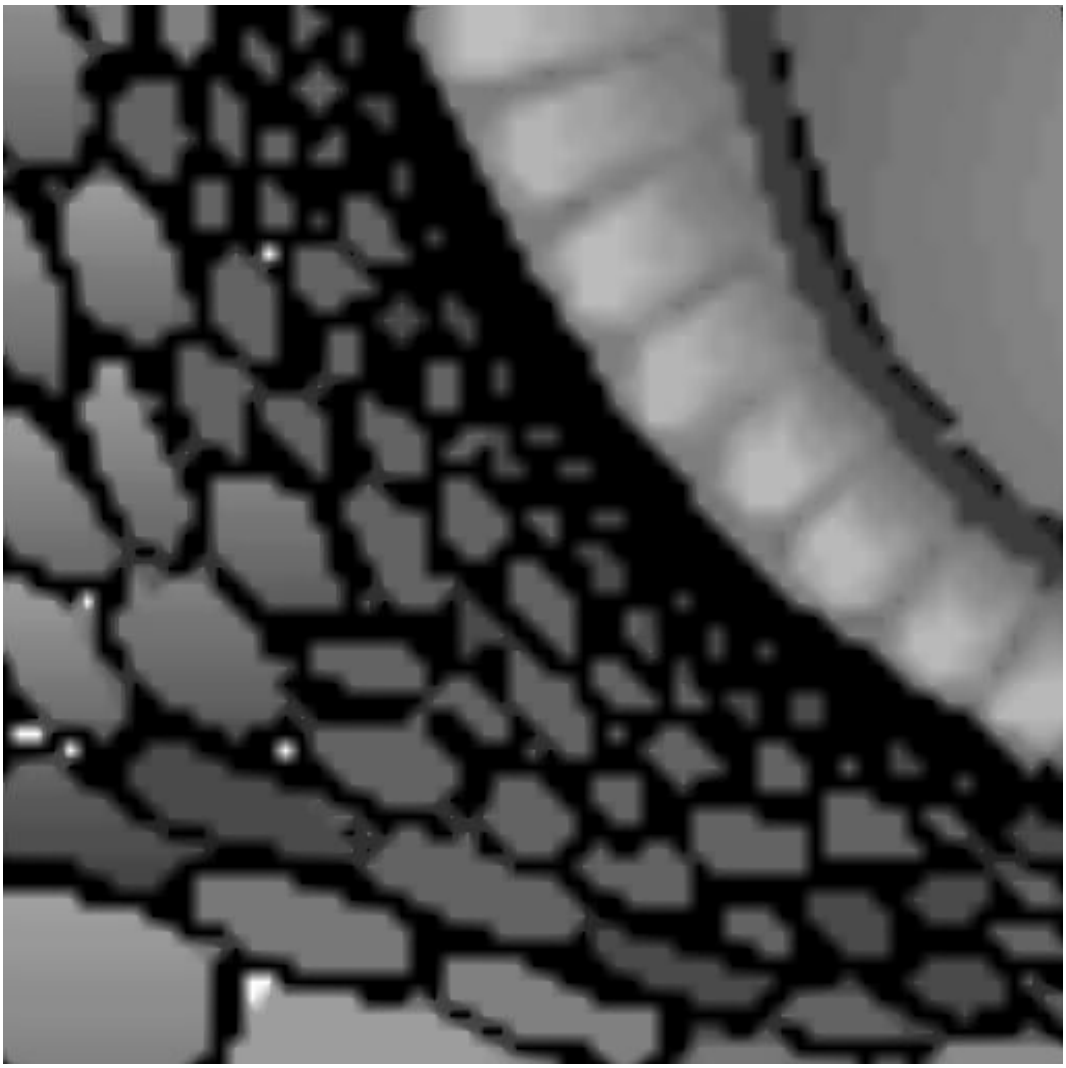


Figure A.12: Frymire image, 8x Level-set Reconstruction

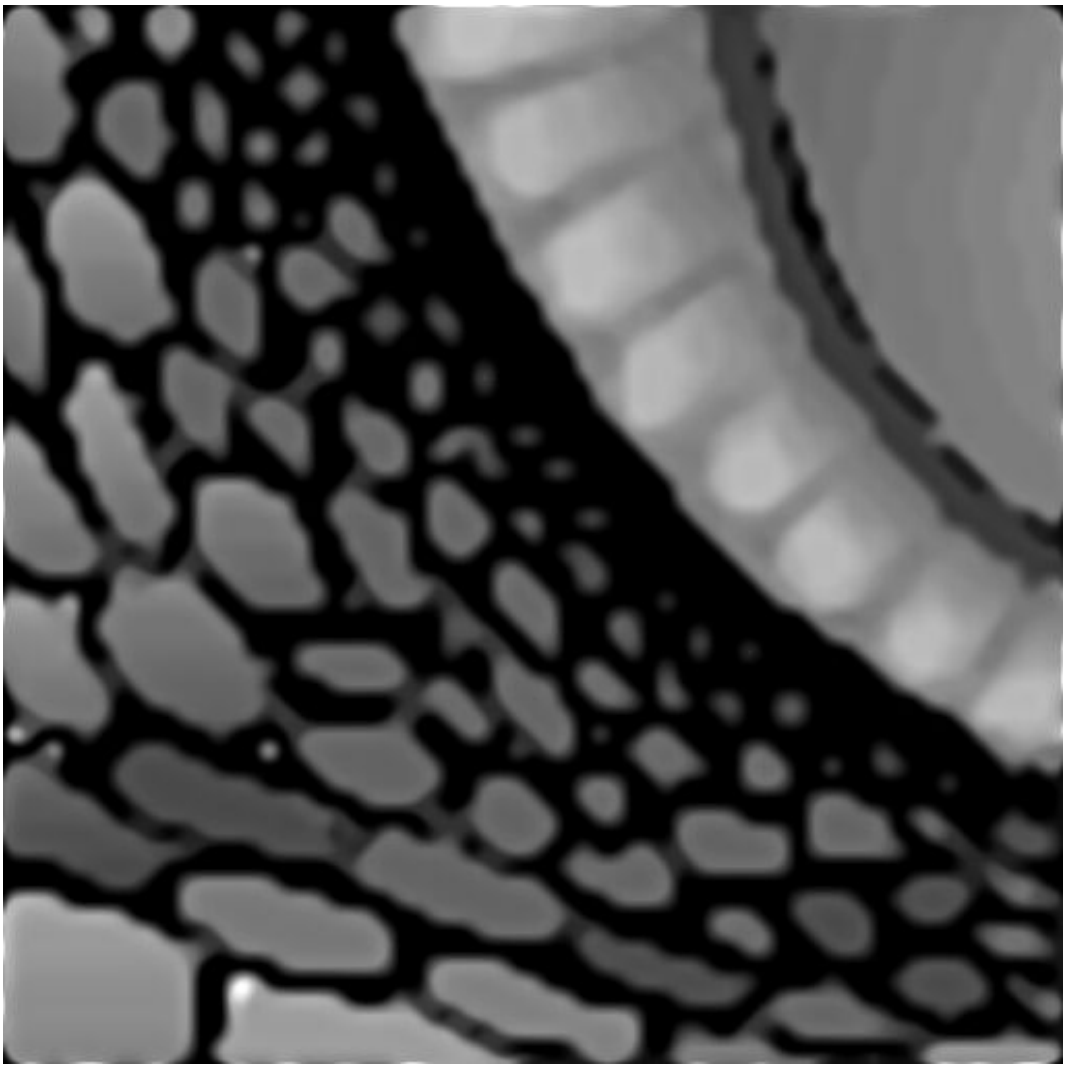


Figure A.13: Frymire image, 8x Constraint-Based Interpolation

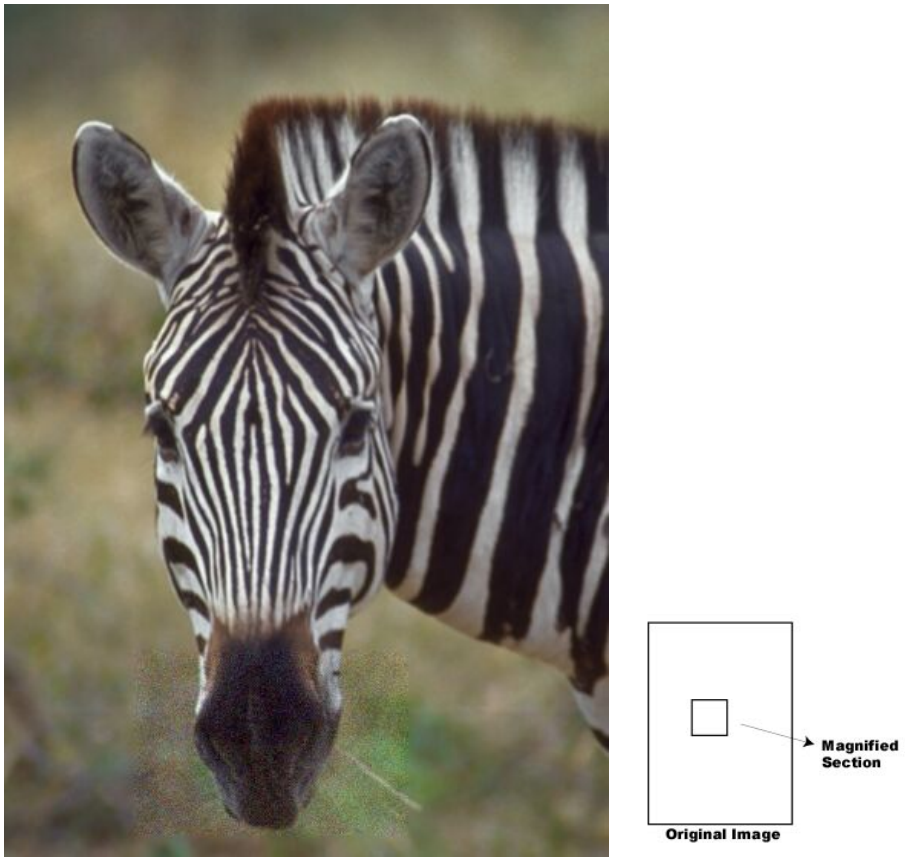


Figure A.14: Original Zebra image showing section that was magnified

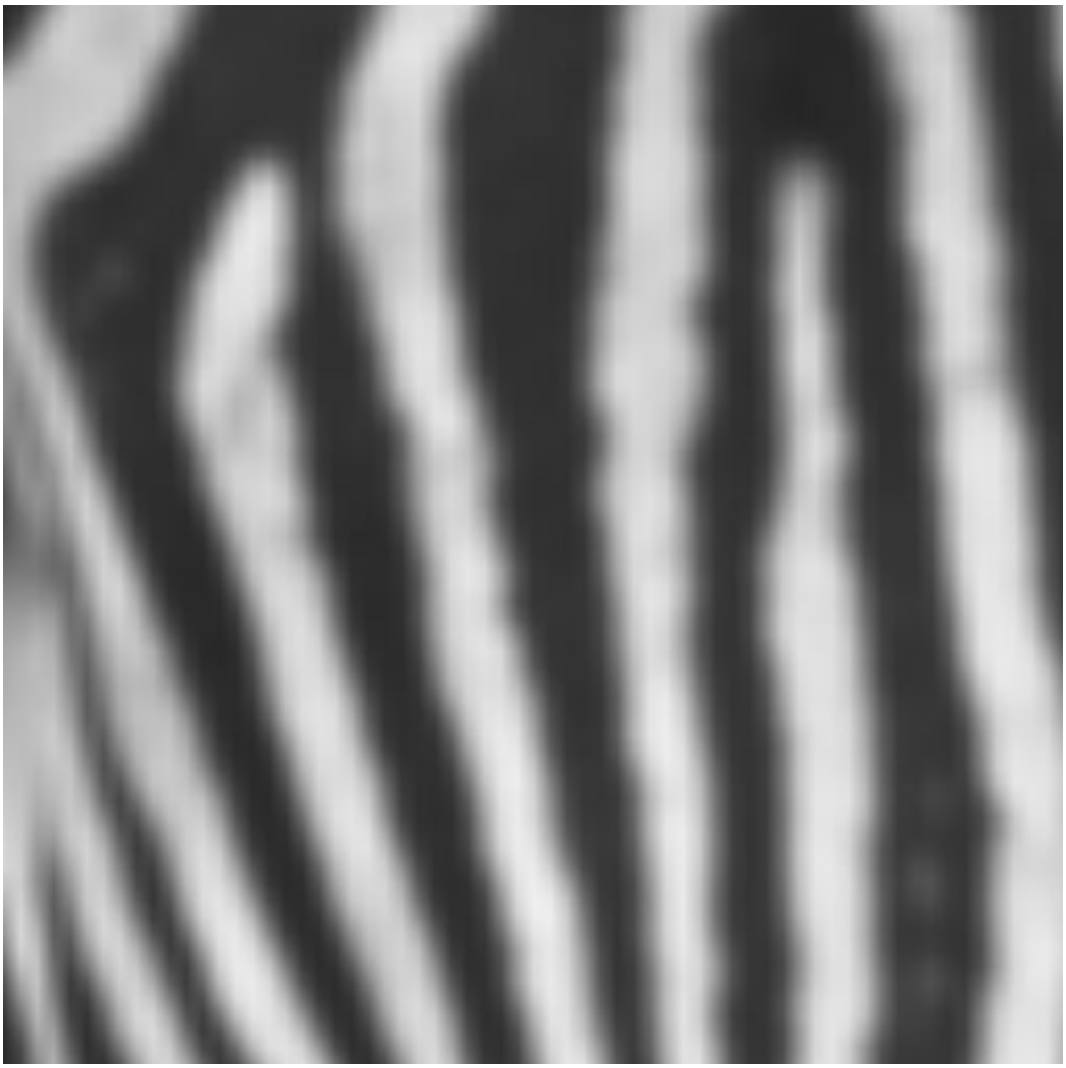


Figure A.15: Zebra image, 8x Bilinear Interpolation

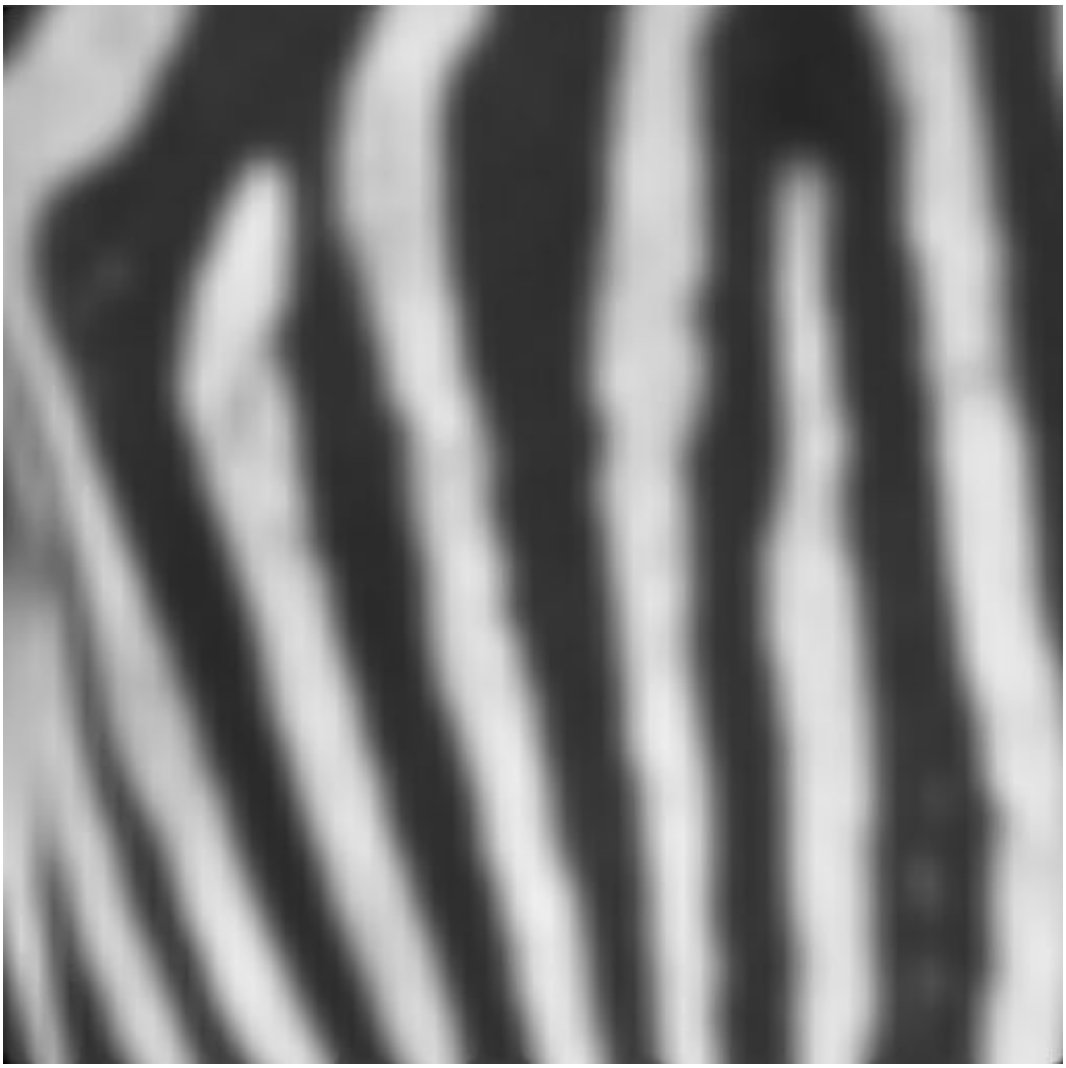


Figure A.16: Zebra image, 8x Level-set Reconstruction

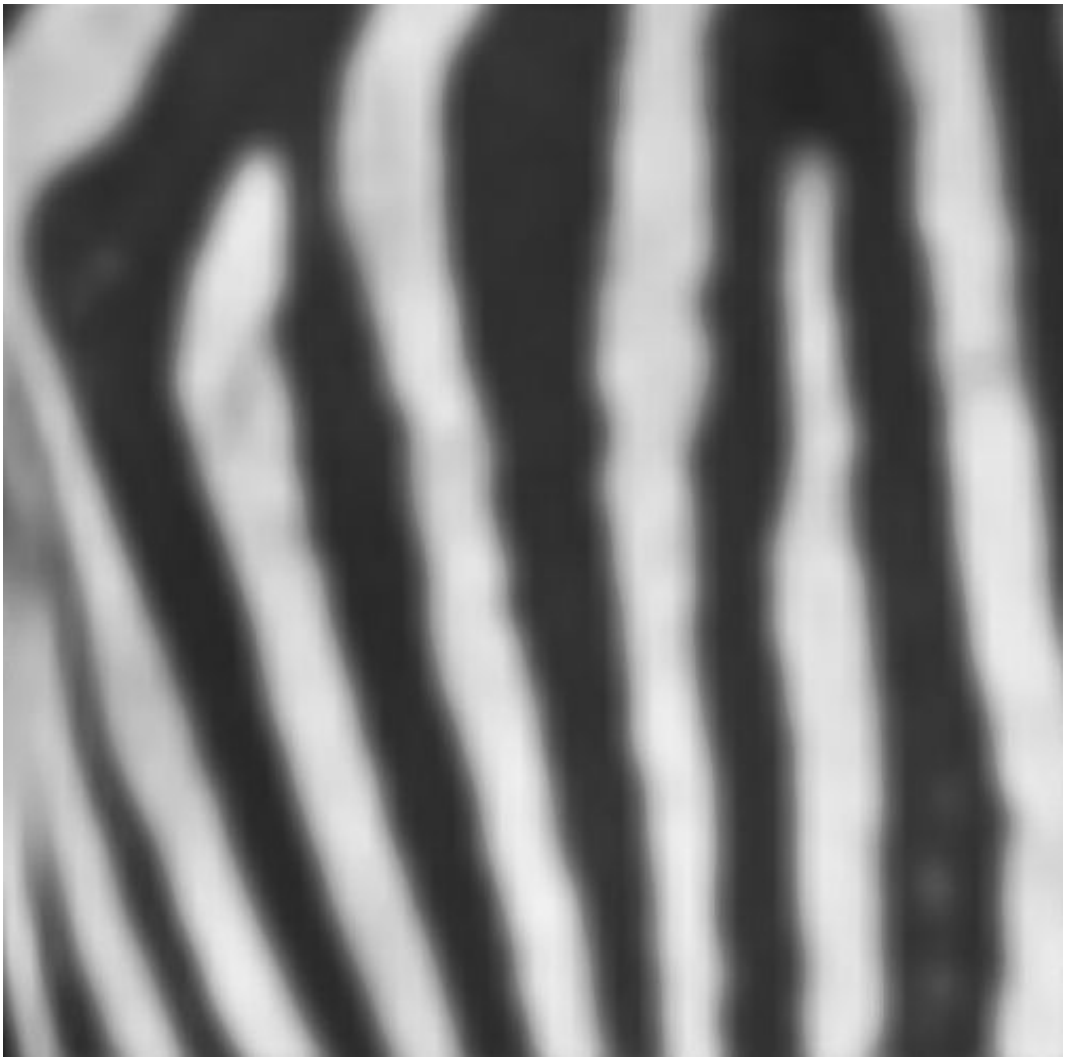


Figure A.17: Zebra image, 8x Constraint-Based Interpolation

Appendix B

Source Code Fragments

B.1 Sensor Model

```
//iterate over our low resolution pixels
double highx, highy, sum, sumweights, lowx, lowy;
int topleftx, toplefty, bottomrightx, bottomrighty, kernelx, kernely;
for( i = 0; i < width; i++ ) {
  for( j = 0; j < height; j++ ) {
    //save the low res value
    value = gray->get( i, j );

    //calc topleft and bottomright x, y values for the high res bounding box
    highx = HighResMappingX->get( i, j );
    highy = HighResMappingY->get( i, j );

    //these values are automatically truncated to ints
    topleftx = highx - ((int)(kernelwidth/2.0))*magfactor;
    toplefty = highy - ((int)(kernelwidth/2.0))*magfactory;
    bottomrightx = highx + ((int)(kernelwidth/2.0))*magfactor;
    bottomrighty = highy + ((int)(kernelwidth/2.0))*magfactory;

    //iterate over the high resolution bounding box and perform the convolution
    sum = 0;
    sumweights = 0;
    for( x = topleftx; x <= bottomrightx; x++ ) {
      for( y = toplefty; y <= bottomrighty; y++ ) {
        //check for bounds
        if( x >= 0 && y >= 0 && x < magimagewidth && y < magimageheight ) {
```

```
          //find out where this high res pixel maps into low res
          lowx = LowResMappingX->get( x, y );
          lowy = LowResMappingY->get( x, y );

          //calculate the nearest kernel subdivision
          kernelx = (lowx - (double)i +
    (((double)kernelwidth - 1.0)/2.0))*subdivsize;
          kernely = (lowy - (double)j +
    (((double)kernelwidth - 1.0)/2.0))*subdivsize;

          sum += magnification->get( x, y )*kernel1->get( kernelx, kernely );
          sumweights += kernel1->get( kernelx, kernely );
        }
      }
    }

    //calculate the difference between high res average and low res pixel
    diff = value - sum/sumweights;

    //finally, we want to spread and save the difference values
    //to do this, we take our diff variable, multiply by the nearest
//kernel subdivision weight, and then divide by the sumoftheweights
    for( x = topleftx; x <= bottomrightx; x++ ) {
      for( y = toplefty; y <= bottomrighty; y++ ) {
        //check for bounds
        if( x >= 0 && y >= 0 && x < magimagewidth && y < magimageheight ) {
          //find out where this high res pixel maps into low res
          lowx = backwardx->get( x, y );
          lowy = backwardy->get( x, y );

          //calculate the nearest kernel subdivision
          kernelx = (lowx - (double)i +
    (((double)kernelwidth - 1.0)/2.0))*subdivsize;
          kernely = (lowy - (double)j +
    (((double)kernelwidth - 1.0)/2.0))*subdivsize;

          tomove = diff*kernel1->get( kernelx, kernely )/sumweights;
          sensorflow->set(x, y, sensorflow->get(x, y) + tomove);
        }
      }
```

```
    }
  }
}
```

B.2 Sharpening Model - Calculate Nudge Flow

```
int i, x, y, width, height;
float nudgeval, gradmag, xval, yval, xxval, yyval, xyval, orig, iwwval;
float dxpos, dxneg, dypos, dyneg;

width = magnification->getWidth();
height = magnification->getHeight();

RealGrayImage *Ix = new RealGrayImage(width, height);
RealGrayImage *Iy = new RealGrayImage(width, height);
RealGrayImage *upIx = new RealGrayImage(width, height);
RealGrayImage *upIy = new RealGrayImage(width, height);
RealGrayImage *Ixx = new RealGrayImage(width, height);
RealGrayImage *Iyy = new RealGrayImage(width, height);
RealGrayImage *Ixy = new RealGrayImage(width, height);
RealGrayImage *upgradmag = new RealGrayImage( width, height );
RealGrayImage *dvalues = new RealGrayImage( width, height );

//calculate our derivative images
ldg->calcIxAccurate(magnification, Ix, 0, 0, width, height );
ldg->calcIyAccurate(magnification, Iy, 0, 0, width, height );
ldg->calcIxx(magnification, Ixx, 0, 0, width, height );
ldg->calcIyy(magnification, Iyy, 0, 0, width, height );
ldg->calcIxy(magnification, Ixy, 0, 0, width, height );

//calculate our DValues
calcConstraintValues( magnification, dvalues, IwwLowRes, gray, startx, starty );

for(y = 0; y < height; y++) {
  for(x = 0; x < width; x++) {
    xval = Ix->get(x, y);
    yval = Iy->get(x, y);
    xxval = Ixx->get(x, y);
    yyval = Iyy->get(x, y);
    xyval = Ixy->get(x, y);
    orig = magnification->get(x, y);

    //calculate Iww using local derivatives
    iwwval = xval*xval*xxval + 2.0*xval*yval*xyval + yval*yval*yyval;
```

```
//calculate possible upwind derivatives
dxpos = magnification->get( x + 1, y ) - magnification->get( x, y );
dxneg = magnification->get( x, y ) - magnification->get( x - 1, y );
dypos = magnification->get( x, y + 1 ) - magnification->get( x, y );
dyneg = magnification->get( x, y ) - magnification->get( x, y - 1 );

//do x
if( dxpos * dxneg <= 0 ) {
 //don't move
 xval = 0;
} else if( iwwval * dxpos >= 0 ) {
 xval = dxpos;
} else {
 xval = dxneg;
}

//do y
if( dypos * dyneg <= 0 ) {
 //don't move
 yval = 0;
} else if( iwwval * dypos >= 0 ) {
 yval = dypos;
} else {
 yval = dyneg;
}

//calculate the gradient magnitude from the upwind derivatives
gradmag = (float)sqrt( (xval * xval) + (yval * yval) );

//multiply iww by -1
iwwval *= -1.0;

//now, multiply gradmag by the sign of iwwval
if( iwwval < 0 ) {
 gradmag *= -1.0;
} else if( iwwval == 0 ) {
 gradmag = 0.0;
}
```

```
    //now, save how much we want to nudge into nudgeflow
    gradmag = gradmag * dvalues->get( x, y );
    nudgeflow->set(x, y, gradmag);
   }//for
}//for

delete Ix;
delete Iy;
delete Ixx;
delete Iyy;
delete Ixy;
delete upIx;
delete upIy;
delete upgradmag;
delete dvalues;
```

B.3 Sharpening Model - Sharpening Constraint

```
int x, y, i, j, width, height, origwidth, origheight;
double temp, dx, dy, xval, yval, xxval, xyval, yyval, orig;
double iwwval, degree, magfactorx, magfactory, v1, v2, v3, v4, backx, backy;

width = magnification->getWidth();
height = magnification->getHeight();
origwidth = IwwLowRes->getWidth();
origheight = IwwLowRes->getHeight();

//figure out our magfactor
magfactorx = (double)width/(double)origwidth;
magfactory = (double)height/(double)origheight;

//calculate the backward mappings
RealGrayImage* backwardx = new RealGrayImage( width, height );
RealGrayImage* backwardy = new RealGrayImage( width, height );
for( i = 0; i < width; i++ ) {
  for( j = 0; j < height; j++ ) {
    backwardx->set( i, j, ((double)i)/magfactorx + ((double)startx) );
    backwardy->set( i, j, ((double)j)/magfactory + ((double)starty) );
  }
}


RealGrayImage* graddir = new RealGrayImage( width, height );

//initialize dvalues to be all zeros
for( i = 0; i < width; i++ ) {
  for( j = 0; j < height; j++ ) {
    dvalues->set( i, j, 0 );
  }
}

//calculate the gradient magnitude using the sobel kernel for each point
for( i = 1; i < width-1; i++ ) {
  for( j = 1; j < height-1; j++ ) {
    //sobel dx
```

```
    dx = 0;
    dx += magnification->get( i - 1, j - 1 )  * -1.0;
    dx += magnification->get( i - 1, j     ) * -2.0;
    dx += magnification->get( i - 1, j + 1 ) * -1.0;
    dx += magnification->get( i + 1, j - 1 );
    dx += magnification->get( i + 1, j  ) * 2.0;
    dx += magnification->get( i + 1, j + 1 );

    //sobel dy
    dy = 0;
    dy += magnification->get( i - 1, j - 1 ) * -1.0;
    dy += magnification->get( i    , j - 1 ) * -2.0;
    dy += magnification->get( i + 1, j - 1 ) * -1.0;
    dy += magnification->get( i - 1, j + 1 );
    dy += magnification->get( i    , j + 1 ) * 2.0;
    dy += magnification->get( i + 1, j + 1 );

    //calculate gradient orientation
    temp = atan2( dy, dx ) * 57.29578;
    if( temp < 0 )
      temp = 180.0 + (180.0+temp);

    graddir->set( i, j, temp );
  }
}

//now, for each point we need to calculate its d value
float zero, fourtyfive, ninety, onethirtyfive;

//figure out which two points in the low res image match up most
//closely with the gradient orientation
double iww1, iww2, origv1, origv2, origv3, origv4, orig1, orig2, d;
for( i = 1; i < width-1; i++ ) {
  for( j = 1; j < height-1; j++ ) {
    degree = graddir->get(i, j);

    //get this pixels four neighbors
    backx = backwardx->get( i, j );
    backy = backwardy->get( i, j );
```

```
//this is where v1 is topleft and v4 is bottom left, so clockwise
v1 = IwwLowRes->get( (int)backx, (int)backy );
v2 = IwwLowRes->get( (int)backx + 1, (int)backy );
v3 = IwwLowRes->get( (int)backx + 1, (int)backy + 1);
v4 = IwwLowRes->get( (int)backx, (int)backy + 1);
origv1 = gray->get( (int)backx, (int)backy );
origv2 = gray->get( (int)backx + 1, (int)backy );
origv3 = gray->get( (int)backx + 1, (int)backy + 1);
origv4 = gray->get( (int)backx, (int)backy + 1);

if( degree > 180.0 ) degree -= 180.0;

if( abs(0.0 - degree) < abs(180.0-degree) ) {
  zero = abs(0.0 - degree);
} else {
  zero = abs(180.0-degree);
}
fourtyfive = abs(45.0 - degree);
ninety = abs(90.0 - degree);
onethirtyfive = abs(135.0 - degree);

//whichever variable has the lowest value is the angle that is closest
if( zero < fourtyfive && zero < ninety && zero < onethirtyfive ) {
  //ok, now pick either v1 and v2 or v3 and v4
  if( (backy - (double)((int)backy)) <= .5 ) {
    iww1 = v1;
    iww2 = v2;
    orig1 = origv1;
    orig2 = origv2;
  } else {
    iww1 = v4;
    iww2 = v3;
    orig1 = origv4;
    orig2 = origv3;
  }
} else if( fourtyfive < zero && fourtyfive < ninety
       && fourtyfive < onethirtyfive ) {
  //here we pick v2 and v4
  iww1 = v4;
  iww2 = v2;
```

```
      orig1 = origv4;
      orig2 = origv2;
    } else if( ninety < zero && ninety < fourtyfive
          && ninety < onethirtyfive ) {
      //ok, now pick either v1 and v4 or v3 and v2
      if( (backx - (double)((int)backx)) <= .5 ) {
        iww1 = v1;
        iww2 = v4;
        orig1 = origv1;
        orig2 = origv4;
      } else {
        iww1 = v2;
        iww2 = v3;
        orig1 = origv2;
        orig2 = origv3;
      }
    } else {
      //here we pick v1 and v3
      iww1 = v1;
      iww2 = v3;
      orig1 = origv1;
      orig2 = origv3;
    }

    //now we analzye iww1 and iww2
    //if iww1 and iww2 are the same sign, then dont do sharpening
    if( iww1 >= 0 && iww2 >= 0 || iww1 < 0 && iww2 < 0 ) {
      dvalues->set( i, j, 0 );
    } else {
      //now, calcualte the difference D
      d = iww2 - iww1;

      //if the sign of d is the same as the sign of the difference
  //between the original pixels, dont sharpen
      if( d >= 0.0 && orig2 - orig1 >= 0.0 || d < 0.0 && orig2 - orig1 < 0.0 ) {
        dvalues->set( i, j, 0 );
      } else {
        //really set the d value
        //first, put d through a function
        d = abs(d);
```

```
        d = d/(1.0 + d);


        //explicitly clamp values if something screwy happened
        if( d > 1.0 ) {
          printf("here d= %lf\n", d);
          d = 1.0;
        }
        if( d < 0 ) d = 0;
        dvalues->set( i, j, d );
      }
    }

  }
}


delete( graddir );
delete( backwardx );
delete( backwardy );
```

B.4 Quad-Based Interpolation

```
//now perform the interpolation on the orthogonal quad
switch( minquad ) {
case 0: //square
  //square is just bilinear interpolation
  a = gray->get( x + startx, y + starty );
  b = gray->get( x + 1 + startx, y + starty);
  c = gray->get( x + 1 + startx, y + 1 + starty);
  d = gray->get( x + startx, y + 1 + starty );

  //xper and yper stay the same with the square

  newval = a + (b - a)*xper + (d - a)*yper + (a - b + c - d)*xper*yper;
  break;
case 1: //upperleft quad
  a = gray->get( x + startx, y - 1 + starty );
  b = gray->get( x + 1 + startx, y + starty);
  c = gray->get( x + 1 + startx, y + 1 + starty);
  d = gray->get( x + startx, y + starty );

  //just change the yper
  yper = yper - xper + 1.0;

  newval = a + (b - a)*xper + (d - a)*yper + (a - b + c - d)*xper*yper;
  break;
case 2: //upperright quad
  a = gray->get( x + startx, y + starty );
  b = gray->get( x + 1 + startx, y - 1 + starty);
  c = gray->get( x + 1 + startx, y + starty);
  d = gray->get( x + startx, y + 1 + starty );

  //just change the yper
  yper = yper + xper;

  newval = a + (b - a)*xper + (d - a)*yper + (a - b + c - d)*xper*yper;
  break;
case 3: //rightupper quad
  a = gray->get( x + 1+ startx, y + starty );
  b = gray->get( x + 2 + startx, y + starty);
```

```
    c = gray->get( x + 1 + startx, y + 1 + starty);
    d = gray->get( x + startx, y + 1 + starty );

    //just change the xper
    xper = xper + yper - 1.0;

    newval = a + (b - a)*xper + (d - a)*yper + (a - b + c - d)*xper*yper;
    break;
  case 4: //rightlower quad
    a = gray->get( x + startx, y + starty );
    b = gray->get( x + 1 + startx, y + starty);
    c = gray->get( x + 2 + startx, y + 1 + starty);
    d = gray->get( x + 1 + startx, y + 1 + starty );

    //just change the xper
    xper = xper - yper;

    newval = a + (b - a)*xper + (d - a)*yper + (a - b + c - d)*xper*yper;
    break;
  case 5: //lowerright quad
    a = gray->get( x + startx, y + starty );
    b = gray->get( x + 1 + startx, y + 1 + starty);
    c = gray->get( x + 1 + startx, y + 2 + starty);
    d = gray->get( x + startx, y + 1 + starty );

    //just change the yper
    yper = yper - xper;

    newval = a + (b - a)*xper + (d - a)*yper + (a - b + c - d)*xper*yper;
    break;
  case 6: //lowerleft quad
    a = gray->get( x + startx, y + 1 + starty );
    b = gray->get( x + 1 + startx, y + starty);
    c = gray->get( x + 1 + startx, y + 1 + starty);
    d = gray->get( x + startx, y + 2 + starty );

    //just change the yper
    yper = yper + xper - 1.0;

    newval = a + (b - a)*xper + (d - a)*yper + (a - b + c - d)*xper*yper;
```

```
    break;
  case 7: //leftlower quad
    a = gray->get( x + startx, y + starty );
    b = gray->get( x + 1 + startx, y + starty);
    c = gray->get( x + startx, y + 1 + starty);
    d = gray->get( x - 1 + startx, y + 1 + starty );

    //just change the xper
    xper = xper + yper;

    newval = a + (b - a)*xper + (d - a)*yper + (a - b + c - d)*xper*yper;
    break;
  case 8: //leftupper quad
    a = gray->get( x - 1 + startx, y + starty );
    b = gray->get( x + startx, y + starty);
    c = gray->get( x + 1 + startx, y + 1 + starty);
    d = gray->get( x + startx, y + 1 + starty );

    //just change the xper
    xper = xper - yper + 1.0;

    newval = a + (b - a)*xper + (d - a)*yper + (a - b + c - d)*xper*yper;
    break;
  }
```

B.5 Quad-Based Interpolation Cost Function

```
/*
The Quad-Based Interpolation Cost Function is as follows. For each of the nine
possible quadrilateral values, two costs are calculated.
One cost is given by (|A-B|+|C-D|)/(|A-D|+|B-C|), where A, B, C, D are the
clockwise corners of the quadrilateral, starting from the
uppermost left corner. This first value calculates how well the actual values
of the points align with the quadrilateral, essentially testing edge strengths.
The second cost is added to this first cost. The second cost uses the same
four points, but analyzing the gradient direction at each point.
The gradient direction is then subtracted from the closest angle that the
quadrilateral is aligned with.
The quadrilateral that has the lowest summation of these two values is used as
the quadrilateral to perform the interpolation in.
*/
void Magnify::calcQuadValues( Quads** quadimage, double** graddirs, RealGrayImage*
 gray, int startx, int starty, int magwidth, int magheight ) {
  int x, y;
  double a, b, c, d, val, temp, adir, bdir, cdir, ddir;

  //now I need to calculate the quadrilateral values that help determine which
  //quadrilateral best approximates the gradient direction
  //the value is (|A-B|+|C-D|)/(|A-D|+|B-C|)

  for( x = startx; x < startx + magwidth; x++ ) {
    for( y = starty; y < starty + magheight; y++ ) {

      //Quad 0: square
      a = gray->get( x, y );
      adir = graddirs[x][y];
      b = gray->get( x + 1, y );
      bdir = graddirs[x+1][y];
      c = gray->get( x + 1, y + 1 );
      cdir = graddirs[x+1][y+1];
      d = gray->get( x, y + 1 );
      ddir = graddirs[x][y+1];

      //you need to check both combinations of two points to see if
  //found gradient for the square
```

```
      val = fabs( a - d ) + fabs( b - c ); //denominator
      if( val == 0 ) {  //check for zero
        temp = 100000;
      } else {
        temp = (fabs( a - b ) + fabs( c - d ))/val;
      }
      val = fabs( a - b ) + fabs( c - d ); //denominator
      if( val == 0 ) {  //check for zero
        val = 1000000;
      } else {
        val = (fabs( a - d ) + fabs( b - c ))/val;
        val *= 100.0; //attempt to normalize with the angle cost too
      }
      if( temp < val ) {
        val = temp;
        //add in a value for how far the fourcorners are off from this
//quads corner angles
        //this is for an edge line that is 0 degrees so check for 90/270
//degree gradient directions
        if( fabs( 90 - adir ) < fabs( 270 - adir ) ) {
val += fabs( 90 - adir ); } else { val += fabs( 270 - adir ); }
        if( fabs( 90 - bdir ) < fabs( 270 - bdir ) ) {
val += fabs( 90 - bdir ); } else { val += fabs( 270 - bdir ); }
        if( fabs( 90 - cdir ) < fabs( 270 - cdir ) ) {
val += fabs( 90 - cdir ); } else { val += fabs( 270 - cdir ); }
        if( fabs( 90 - ddir ) < fabs( 270 - ddir ) ) {
val += fabs( 90 - ddir ); } else { val += fabs( 270 - ddir ); }
      } else {
        //add in a value for how far the fourcorners are off from this quads
//corner angles
        //this is for an edge line that is 90 degrees so check for 0/180
/degree gradient directions
        if( fabs( 0 - adir ) < fabs( 180 - adir ) ) {
val += fabs( 0 - adir ); } else { val += fabs( 180 - adir ); }
        if( fabs( 0 - bdir ) < fabs( 180 - bdir ) ) {
val += fabs( 0 - bdir ); } else { val += fabs( 180 - bdir ); }
        if( fabs( 0 - cdir ) < fabs( 180 - cdir ) ) {
val += fabs( 0 - cdir ); } else { val += fabs( 180 - cdir ); }
        if( fabs( 0 - ddir ) < fabs( 180 - ddir ) ) {
val += fabs( 0 - ddir ); } else { val += fabs( 180 - ddir ); }
```

```
      }
      quadimage[x-startx][y-starty].values[0] = val;

      //Quad 1: upperleft quad
      a = gray->get( x, y - 1 );
      adir = graddirs[x][y-1];
      b = gray->get( x + 1, y );
      bdir = graddirs[x+1][y];
      c = gray->get( x + 1, y + 1 );
      cdir = graddirs[x+1][y+1];
      d = gray->get( x, y );
      ddir = graddirs[x][y];

      val = fabs( a - d ) + fabs( b - c ); //denominator
      if( val == 0 ) {  //check for zero
        val = 100000;
      } else {
        val = (fabs( a - b ) + fabs( c - d ))/val;
        val *= 100.0; //attempt to normalize with the angle cost too
      }
      //add in a value for how far the fourcorners are off from this
  //quads corner angles
      if( fabs( 135 - adir ) < fabs( 315 - adir ) ) {
val += fabs( 135 - adir ); } else { val += fabs( 315 - adir ); }
      if( fabs( 135 - bdir ) < fabs( 315 - bdir ) ) {
val += fabs( 135 - bdir ); } else { val += fabs( 315 - bdir ); }
      if( fabs( 135 - cdir ) < fabs( 315 - cdir ) ) {
val += fabs( 135 - cdir ); } else { val += fabs( 315 - cdir ); }
      if( fabs( 135 - ddir ) < fabs( 315 - ddir ) ) {
val += fabs( 135 - ddir ); } else { val += fabs( 315 - ddir ); }
      quadimage[x-startx][y-starty].values[1] = val;

      //Quad 2: upperright quad
      a = gray->get( x, y );
      adir = graddirs[x][y];
      b = gray->get( x + 1, y - 1 );
      bdir = graddirs[x+1][y-1];
      c = gray->get( x + 1, y );
      cdir = graddirs[x+1][y];
      d = gray->get( x, y + 1 );
```

```
      ddir = graddirs[x][y+1];

      val = fabs( a - d ) + fabs( b - c ); //denominator
      if( val == 0 ) {  //check for zero
        val = 100000;
      } else {
        val = (fabs( a - b ) + fabs( c - d ))/val;
        val *= 100.0; //attempt to normalize with the angle cost too
      }
      //add in a value for how far the fourcorners are off from
  //this quads corner angles
      if( fabs( 225 - adir ) < fabs( 45 - adir ) ) {
val += fabs( 225 - adir ); } else { val += fabs( 45 - adir ); }
      if( fabs( 225 - bdir ) < fabs( 45 - bdir ) ) {
val += fabs( 225 - bdir ); } else { val += fabs( 45 - bdir ); }
      if( fabs( 225 - cdir ) < fabs( 45 - cdir ) ) {
val += fabs( 225 - cdir ); } else { val += fabs( 45 - cdir ); }
      if( fabs( 225 - ddir ) < fabs( 45 - ddir ) ) {
val += fabs( 225 - ddir ); } else { val += fabs( 45 - ddir ); }
      quadimage[x-startx][y-starty].values[2] = val;

      //Quad 3: rightupper quad
      b = gray->get( x + 1, y );
      bdir = graddirs[x+1][y];
      c = gray->get( x + 2, y );
      cdir = graddirs[x+2][y];
      d = gray->get( x + 1, y + 1 );
      ddir = graddirs[x+1][y+1];
      a = gray->get( x, y + 1 );
      adir = graddirs[x][y+1];

      val = fabs( a - d ) + fabs( b - c ); //denominator
      if( val == 0 ) {  //check for zero
        val = 100000;
      } else {
        val = (fabs( a - b ) + fabs( c - d ))/val;
        val *= 100.0; //attempt to normalize with the angle cost too
      }
      if( fabs( 225 - adir ) < fabs( 45 - adir ) ) {
val += fabs( 225 - adir ); } else { val += fabs( 45 - adir ); }
```

```
      if( fabs( 225 - bdir ) < fabs( 45 - bdir ) ) {
val += fabs( 225 - bdir ); } else { val += fabs( 45 - bdir ); }
      if( fabs( 225 - cdir ) < fabs( 45 - cdir ) ) {
val += fabs( 225 - cdir ); } else { val += fabs( 45 - cdir ); }
      if( fabs( 225 - ddir ) < fabs( 45 - ddir ) ) {
val += fabs( 225 - ddir ); } else { val += fabs( 45 - ddir ); }
      quadimage[x-startx][y-starty].values[3] = val;

      //Quad 4: rightlower quad
      b = gray->get( x, y );
      bdir = graddirs[x][y];
      c = gray->get( x + 1, y );
      cdir = graddirs[x+1][y];
      d = gray->get( x + 2, y + 1 );
      ddir = graddirs[x+2][y+1];
      a = gray->get( x + 1, y + 1 );
      adir = graddirs[x+1][y+1];

      val = fabs( a - d ) + fabs( b - c ); //denominator
      if( val == 0 ) {  //check for zero
        val = 100000;
      } else {
        val = (fabs( a - b ) + fabs( c - d ))/val;
        val *= 100.0; //attempt to normalize with the angle cost too
      }
      if( fabs( 135 - adir ) < fabs( 315 - adir ) ) {
val += fabs( 135 - adir ); } else { val += fabs( 315 - adir ); }
      if( fabs( 135 - bdir ) < fabs( 315 - bdir ) ) {
val += fabs( 135 - bdir ); } else { val += fabs( 315 - bdir ); }
      if( fabs( 135 - cdir ) < fabs( 315 - cdir ) ) {
val += fabs( 135 - cdir ); } else { val += fabs( 315 - cdir ); }
      if( fabs( 135 - ddir ) < fabs( 315 - ddir ) ) {
val += fabs( 135 - ddir ); } else { val += fabs( 315 - ddir ); }
      quadimage[x-startx][y-starty].values[4] = val;

      //Quad 5: lowerright quad
      a = gray->get( x, y );
      adir = graddirs[x][y];
      b = gray->get( x + 1, y + 1 );
      bdir = graddirs[x+1][y+1];
```

```
      c = gray->get( x + 1, y + 2 );
      cdir = graddirs[x+1][y+2];
      d = gray->get( x, y + 1 );
      ddir = graddirs[x][y+1];

      val = fabs( a - d ) + fabs( b - c ); //denominator
      if( val == 0 ) {  //check for zero
        val = 100000;
      } else {
        val = (fabs( a - b ) + fabs( c - d ))/val;
        val *= 100.0; //attempt to normalize with the angle cost too
      }
      //add in a value for how far the fourcorners are off
  //from this quads corner angles
      if( fabs( 135 - adir ) < fabs( 315 - adir ) ) {
val += fabs( 135 - adir ); } else { val += fabs( 315 - adir ); }
      if( fabs( 135 - bdir ) < fabs( 315 - bdir ) ) {
val += fabs( 135 - bdir ); } else { val += fabs( 315 - bdir ); }
      if( fabs( 135 - cdir ) < fabs( 315 - cdir ) ) {
val += fabs( 135 - cdir ); } else { val += fabs( 315 - cdir ); }
      if( fabs( 135 - ddir ) < fabs( 315 - ddir ) ) {
val += fabs( 135 - ddir ); } else { val += fabs( 315 - ddir ); }
      quadimage[x-startx][y-starty].values[5] = val;

      //Quad 6: lowerleft quad
      a = gray->get( x, y + 1 );
      adir = graddirs[x][y+1];
      b = gray->get( x + 1, y );
      bdir = graddirs[x+1][y];
      c = gray->get( x + 1, y + 1 );
      cdir = graddirs[x+1][y+1];
      d = gray->get( x, y + 2 );
      ddir = graddirs[x][y+2];

      val = fabs( a - d ) + fabs( b - c ); //denominator
      if( val == 0 ) {  //check for zero
        val = 100000;
      } else {
        val = (fabs( a - b ) + fabs( c - d ))/val;
        val *= 100.0; //attempt to normalize with the angle cost too
```

```
      }
      if( fabs( 225 - adir ) < fabs( 45 - adir ) ) {
val += fabs( 225 - adir ); } else { val += fabs( 45 - adir ); }
      if( fabs( 225 - bdir ) < fabs( 45 - bdir ) ) {
val += fabs( 225 - bdir ); } else { val += fabs( 45 - bdir ); }
      if( fabs( 225 - cdir ) < fabs( 45 - cdir ) ) {
val += fabs( 225 - cdir ); } else { val += fabs( 45 - cdir ); }
      if( fabs( 225 - ddir ) < fabs( 45 - ddir ) ) {
val += fabs( 225 - ddir ); } else { val += fabs( 45 - ddir ); }
      quadimage[x-startx][y-starty].values[6] = val;

      //Quad 7: leftlower quad
      b = gray->get( x, y );
      bdir = graddirs[x][y];
      c = gray->get( x + 1, y );
      cdir = graddirs[x+1][y];
      d = gray->get( x, y + 1 );
      ddir = graddirs[x][y+1];
      a = gray->get( x - 1, y + 1 );
      adir = graddirs[x-1][y+1];

      val = fabs( a - d ) + fabs( b - c ); //denominator
      if( val == 0 ) {  //check for zero
        val = 100000;
      } else {
        val = (fabs( a - b ) + fabs( c - d ))/val;
        val *= 100.0; //attempt to normalize with the angle cost too
      }
      if( fabs( 225 - adir ) < fabs( 45 - adir ) ) {
val += fabs( 225 - adir ); } else { val += fabs( 45 - adir ); }
      if( fabs( 225 - bdir ) < fabs( 45 - bdir ) ) {
val += fabs( 225 - bdir ); } else { val += fabs( 45 - bdir ); }
      if( fabs( 225 - cdir ) < fabs( 45 - cdir ) ) {
val += fabs( 225 - cdir ); } else { val += fabs( 45 - cdir ); }
      if( fabs( 225 - ddir ) < fabs( 45 - ddir ) ) {
val += fabs( 225 - ddir ); } else { val += fabs( 45 - ddir ); }
      quadimage[x-startx][y-starty].values[7] = val;

      //Quad 8: leftupper quad
      b = gray->get( x - 1, y );
```

```
      bdir = graddirs[x-1][y];
      c = gray->get( x, y );
      cdir = graddirs[x][y];
      d = gray->get( x + 1, y + 1 );
      ddir = graddirs[x+1][y+1];
      a = gray->get( x, y + 1 );
      adir = graddirs[x][y+1];

      val = fabs( a - d ) + fabs( b - c ); //denominator
      if( val == 0 ) {  //check for zero
        val = 100000;
      } else {
        val = (fabs( a - b ) + fabs( c - d ))/val;
        val *= 100.0; //attempt to normalize with the angle cost too
      }
      if( fabs( 135 - adir ) < fabs( 315 - adir ) ) {
val += fabs( 135 - adir ); } else { val += fabs( 315 - adir ); }
      if( fabs( 135 - bdir ) < fabs( 315 - bdir ) ) {
val += fabs( 135 - bdir ); } else { val += fabs( 315 - bdir ); }
      if( fabs( 135 - cdir ) < fabs( 315 - cdir ) ) {
val += fabs( 135 - cdir ); } else { val += fabs( 315 - cdir ); }
      if( fabs( 135 - ddir ) < fabs( 315 - ddir ) ) {
val += fabs( 135 - ddir ); } else { val += fabs( 315 - ddir ); }
      quadimage[x-startx][y-starty].values[8] = val;
    }
  }

}
```