Planet Photo-Topography Using Shading and Stereo

by

Charles XiaoJian Yan

B.S., California Institute of Technology, 1990

Submitted to the Department of Physics
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

December 1993

Signature of Author ....
Department of Physics
December 15, 1993

Certified by ....
Prof. Berthold K. P. Horn
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by ....
Prof. Robert P. Redwine
Professor of Physics
Thesis Supervisor

Accepted by ....
Prof. G. F. Koster
Chairman, Departmental Committee on Graduate Students

This page is left blank intently.

Planet Photo-Topography Using Shading and Stereo

by

Charles XiaoJian Yan

Submitted to the Department of Physics
on December 15, 1993, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Two of the most successful methods in Computer Vision developed over the years are Shading and Stereo. Newly proposed DFSS(Depth From Shading and Stereo) method for fusing the two is examined. The most robust z-only method is utilized and explored in analyzing planetary data images, namely the Mars image data from Viking Space Project. Two major extensions to DFSS are to integrate all the image cells, instead of single test cell to increase robustness, and to use realistic Mars images, instead of synthetic test images. Another unique feature presented in this thesis is in dealing with real world images, containing noise, geometry error, calibration errors, and reflectance errors.

Thesis Supervisor: Prof. Berthold K. P. Horn
Title: Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Prof. Robert P. Redwine
Title: Professor of Physics

This page is left blank intently.

Acknowledgments

First and foremost, I would like to thank my thesis advisor, Professor Berthold K. P. Horn, for his invaluable guidance, support, understanding and inspiration. Aside from academic and research help, Professor Horn is a very kind and gentle person. I am also thankful to Professor Robert Redwine, who is not only research supervisor but also academic advisor. Professor Redwine has listen to the problems I encountered during the years, helped to solve them, steered me towards a mature professional.

Also, I am grateful to Dr. Mike Caplinger. Dr. Caplinger has put in much time to select and process the images pair for this research. He has given many helpful suggestions during this research. Without Dr. Caplinger's help, the project will never succeed.

Also thanks go to fellow students at the AI lab for the discussions and suggestions, especially Dr. Clay Thompson.

Last, certainly not the least, I feel deeply in debt to the support, understanding, and help from my wife, Ellen Hui Ding, my parents, Professor WuGuang Yan and Professor WenXiu Gao. Without their continuous caring and encouragement, I can never be wherever I am today.

This research was supported in part by NASA contract NAS5-31352. Additional support came from DARPA contract N00014-91-J-4038 which helps support general activities at the M.I.T. Artificial Intelligence Laboratory.

This page is left blank intently.

Contents

This page is left blank intently.

List of Figures

This page is left blank intently.

List of Tables

This page is left blank intently.

Chapter 1

Introduction

Machine vision is the study of algorithms and techniques for analyzing and processing visual inputs so as to determine one or more properties of the external world. Following Marr's [15] Classification, vision algorithms can be grouped into one of three levels by the processing and analyzing stage: Early vision, Intermediate vision, and High level vision. I will concentrate on Early Vision which seeks to work with raw image data and to produce an estimate of some property or properties of the 3-D world. An example in this field is Shape from Shading algorithm which estimates the shape or relative depth of an object from a gray-level image, as discussed in "Robot Vision" [8].

Machine vision is closely allied with three fields, image processing, pattern classification and scene analysis. Machine vision differs from image processing in that the result is not a better or enhanced image, not a new image, but an estimate of some external properties of the 3-D world. Pattern classification and scene analysis are associated with intermediate vision and high level vision. Unlike computer graphics which tries to produce a realistic image from a stored model of the world, machine vision tries to produce a realistic model of the world from an image. In this way, a vision system (camera+algorithm) can be described as a sensor that converts a large number (N^2 on an N-by-N grid) of measurements into a representation of the external world, for example, height information of a N by N pixel image. Vision system is complex because it process information in 2-D in contrast to most system processing 1-D or 0-D information. Its complexity also stems from input in 2-D and the result in even higher 3-D. In another word, it is a synthesis process rather than a deduction process. We can not at this stage to build a "universal" or " general purpose" vision system. Instead, we address ourselves either to system that perform a particular task in a controlled environment or to a model that could eventually become part of a general purpose system.

In this thesis, I will describe a machine vision algorithm that combines the methods of three successful early vision algorithms, its implementation and the performance on real images, namely Viking Mars Survey images. The algorithm seeks to determine the topology of a planet from two images, which are taken at different time by two cameras at different location with different lighting conditions.

1.1 Background

Over the past two decades many early vision algorithms have been developed. Most notably, Algorithms have been developed for edge finding [16], Binocular Stereo [17, 3], Photometric Stereo [28, 20], Shape from Shading [8, 11], Shape from texture [13], Structure from Motion [25] and Optical Flow [10]. These algorithms, for the most part, are very sensitive to noise in the image(s). They seem to perform well on synthetic images, but perform poorly on real images. In order to make the algorithms more robust, some researchers are moving toward integration or fusion of two or more of these methods. The typical fusion paradigm is to explore physical constraints that are present between the solutions of the candidate methods. These constraints are then used to combine the outputs of each method to produce a fused solution to the vision problem. The hope is that by combining the information available from disparate methods, a more robust vision system can be formed.

Figure 1-1: Example sketch of cameras and light source in a pair of Viking images

The vision algorithms mentioned above fall into one of two camps: those algorithms that are based on variational formulations, such as Horn's Shape from Shading algorithm, and those algorithms that are feature based, such as the Marr and Poggio's Binocular Stereo algorithm. Variational methods usually result in an optimization problem while feature based methods usually employ direct search methods. Both approaches have been successful for certain problems.

The type of algorithm affects how easy it is to integrate or fuse more than one algorithm. In general, the feature based methods are hard to fuse since the algorithms are highly specialized and tuned to each task. On the other hand, the variational methods perhaps can be easily fused by simply combining the cost functions from disparate methods intelligently and forming a combined optimization problem. In the discussion below, I will present a method based on variational approach.

The problem on hand is to fuse the Shape from Shading, Binocular Stereo, and Photometric Stereo algorithms so as to obtain more accurate and robust estimates of the surface topography. Following its originally proposed name [1], I call the resulting algorithm Depth from Shading and Stereo (DFSS). The research here mainly is concerned with the robustness, efficiency, and performance of the *z-only* DFSS algorithm in dealing with real images.

This research is motivated by a problem proposed by NASA. NASA is interested in determining the surface structure (or topography) of the planets in our solar system.

[1]Depth from Shading and Stereo was proposed and developed by Clay Thompson in his 1992 Ph.D. thesis [24]

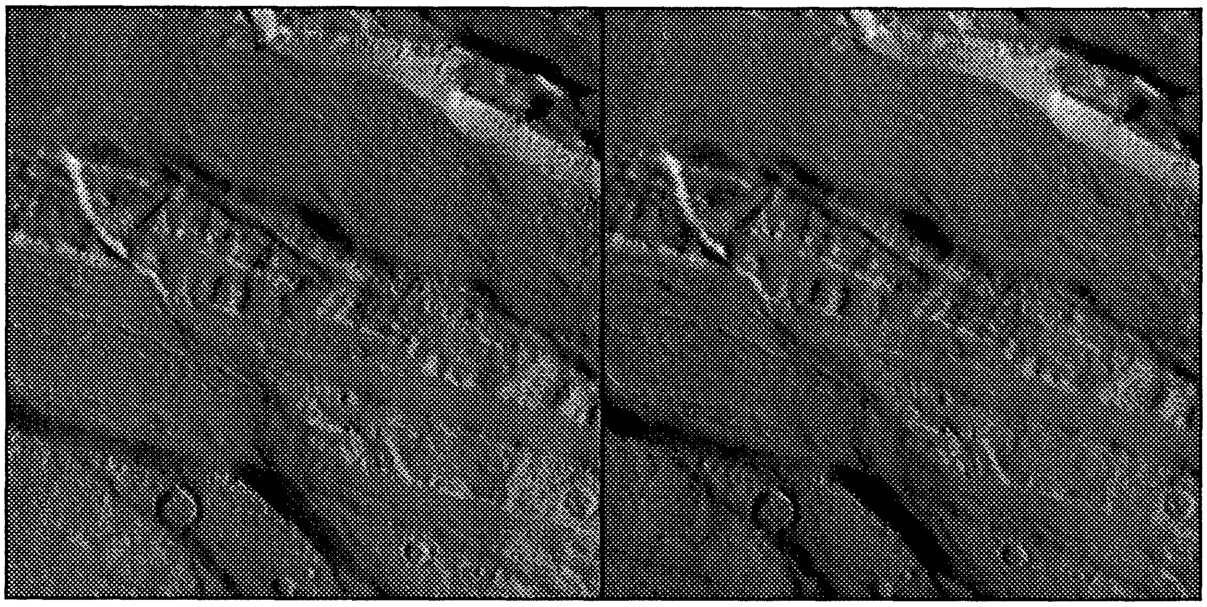


Figure 1-2: Example image pair from Viking Space Project. The difference in shading is clearly shown, due to the different shading conditions

Toward this end, NASA has used its explorer probes (e.g. Viking and Voyager) to obtain images of the same patch of surface on a planet from two different, but known, locations (see Figure 1-1). Since the images are taken at different time, the sun is not in the same position relative to the planet or the camera. This results in the images often being radically different from each other. (see Figure 1-2) This means none of the existing algorithm can handle the problem.

The importance of this property becomes clear when you compare this situation with the assumptions of several Early Vision algorithms. Binocular stereo algorithms usually assume that the two images only differ by an offset (called disparity) that is caused by the projection of a 3-D object. This implies that the images should look very similar. As mentioned above, the NASA Viking images do not meet this assumption, hence the normal Binocular stereo algorithms will fail on these images.

Photometric stereo algorithms, on the other hand, assume that the images were taken from the same camera position but with different light positions. This implies that the corresponding points in each image are the projection of the same point in the scene. Again, the NASA Viking images do not meet this assumption, and the Photometric stereo algorithms will not work.

Only the Shape from Shading algorithms can be used with NASA Viking images, except that the images must be processed one at a time. This results in two different interpretations of the planet's surface topography.

The NASA Viking images are thus a natural choice for the investigation of fusion techniques. The Depth from Shading and Stereo algorithm I have implemented will incorporate the three modules mentioned above into one.

1.2 Planet Photo-Topography

The planet photo-topography problem in this thesis seeks to determine the topology of a planet's surface based on two images of the planet, taken from two different vantage points at two different time, as shown in Figure 1-1 and Figure 1-2. As discussed in previous section, no ready algorithm can be applied to this problem. However, it bears great resemblance with some well understood and developed algorithms, specifically, Binocular Stereo, Photometric Stereo, and Shape from Shading.

Planet photo-topography has certain similarity to Binocular Stereo. Unfortunately, it is more complicated than stereo vision in two major ways. First, The images are typically taken at two different time and position; Second, the distance between the two camera position are usually large and the camera directions are different. This means two different light source and camera positions and directions. As the result, The two images might look quite different from each other, even though they are images of the same surface patch, see Figure 1-2.

Contrast to this, Binocular image pairs are usually taken simultaneously, from positions that are near to each other, and with the same lighting. The images look very similar except for a relative shift (i.e. disparity) of objects due to their distance from cameras. If the disparity for all points in the image and the relative distance from the cameras is known, then the depth of the objects can be computed directly. The images are similar, so most stereo algorithm determine the disparity by trying to match features in one image with features in the other. Unfortunately, this approach won't work for planet photo-topography images.

Planet photo-topography also shares some aspects with Shape from Shading problem. Shape from Shading takes a gray-scale image of a surface and determines the surface topology by exploiting the Shading information in the image. Shape from Shading requires that the surface reflectance properties to be know. Assuming the reflectance properties are known, we could use a Shape from Shading algorithm to estimate the surface topology from each of the planet images. Unfortunately, the surface estimates based on each image will be different. They may not even be similar. Worst of all, the surface estimates from each image may not have the same orientation; one could be concave while the other is convex.

Planet photo-topography also has aspects in common with Photometric Stereo problem. A Photometric Stereo takes two images of the same scene with two different lighting conditions. The cameras is not moved between images. The result is two images that look different but where the correspondence is known explicitly. If the light positions are far apart enough, it is possible to determine the surface orientation directly. For Lambertian reflectance, two images can constraint the surface orientation to two possible values at each point.

The NASA Viking images are based on two different light source positions and directions. Like Photometric Stereo, the two light sources can constrain the surface orientation, if the correspondence is known. But the correspondence is based on Binocular Stereo. Thus planet photo-topography, Photometric Stereo and Binocular

Stereo are closely linked.

1.3 Viking Project, Planet Image Data

Digital image data from the Viking Mission to Mars, NASA's Planetary Data System(PDS), have been available. Through the Geosciences Discipline Node at Washington University, the Image Node at the U. S. Geological Survey, Flagstaff, Arizona and the Jet Propulsion Laboratory, the digital archive of images acquired by the Viking orbital Visual Imaging Subsystem (VIS), including the Experiment Data Record (EDR), are placed on compact read only optical disk media (CD-ROM).

1.3.1 Viking Mission

The Viking Mission consisted of four spacecraft: two identical orbiters and two identical landers. One of the orbiter experiment was the Visual Imaging Subsystem (VIS), which acquired the images that are used in this thesis.

The Viking orbiter spacecraft operated in orbit around Mars from 1976 to 1980. The orbiter imaging systems imaged all of the terrains on Mars, collected some color and stereo images, and made observations of Phobos and Deimos. Some image sequences acquired by the VIS experiment include systematic medium and high resolution coverage of large portions of the surface, stereo images, observations of Phobos and Demios, color images of the equatorial regions, observations of the polar regions, and monitoring dust storm activity.

1.3.2 Viking Orbiter Visual Subsystem

Each Viking Orbiter was equipped with two identical vidicon cameras, called the Visual Imaging Subsystem (VIS) [26], [14], [1]. Each VIS camera consisted of a telescope, a slow scan vidicon, a filter wheel, and associated electronics. A digital image was generated by scanning the vidicon face plate. A full resolution, uncompressed

Viking orbiter image consists of an array of 1056 lines with 1204 samples per line. The images then transmitted back to earth station. The images were radiometrically and geometrically calibrated and stored on tape. Subset of the images are distributed on CD-ROM.

1.4 Related Research

The algorithm discussed here is mostly related to the works of Horn [12], [9], [8], [11], Gennert [5], and Szeliski [21], [22]. Variational (least squares) approach is based significantly on the work of Horn [11], [10], [19]. The Shape from Shading part is build upon the work of Horn [11], Szeliski [22]. The stereo part is based on the gray-scale stereo algorithm of Gennert [5]. Hierarchical basis functions of Szeliski and use conjugate gradient optimization as do Leclerc and Bobick. This work also has close relation to the work of Hartt and Carlotto [6], [7], [27], and McEwen [18].

The Depth from Shading and Stereo was proposed and developed by Clay Thompson in his PH.D. thesis, [24]. In his thesis, Thompson proposed a fusion method with variational technique based on Shape from Shading and Stereo. A cost function combined Shape from Shading and stereo with smoothness regulated term is used. Conjugate gradient optimization is performed to estimate depth information. Test on synthetic images show *z-only* algorithm is the best performed and the most robust method. I will discuss the details of *z-only* algorithm in chapter 2 and chapter 3. Test on synthetic images are presented in chapter 4.

1.5 About This Thesis

In the following Chapters, I will lay the ground for Depth from Shading and Stereo by discussing each underlying fused algorithms, Shape from Shading, Binocular Stereo and Photometric Stereo, and presents *z-only* Depth from Shading and Stereo algorithm's performance on synthetic test image pairs and real Mars Viking Space Project image pairs. In Chapter 2, I will discuss each components of DFSS algorithm, Shape from Shading, Binocular and Photometric Stereo in detail. In Chapter 3, I will show the formulation of DFSS algorithm in a variational approach. In Chapter 4, I will present the result of running DFSS algorithm on synthetic images, to understand its strength and weakness. Chapter 5 shows the performance of DFSS algorithm on Mars image pairs from Viking Space Project. Chapter 6 is devoted to analysis of various error and its effects on the algorithm's performance on real images. Chapter 7 summaries and discuss issues in DFSS' merits, limitation, implementation and its extension.

This thesis is organized for people with various backgrounds in vision field. For people who are familiar with machine vision, they can skip to chapter 3. For those who are familiar with DFSS algorithm, they can skip to chapter 5, and compare the result on Viking images with that on synthetic images.

Chapter 2

Overview

In this chapter we discuss how images are formed and how they are represented by a computer. Understanding image formation is a prerequisite for full understanding of the methods for recovering information from images. The focus will be on coordinate system, image generation process and photo-topography properties. In particular, we will discuss Binocular Stereo, photometric stereo, and shape from shading. Finally, a set of simplified equations aiming at photo-topography problem are summarized.

2.1 Coordinate System

An image is a two dimensional pattern of brightness. In analyzing the process by which a three-dimensional world is projected onto a two-dimensional image plane, we have to first set up the proper coordinate system. The most straight forward description is that the surface represents a height function over some selected 2-D domain., such as:

$$z = z(x, y) \tag{2.1}$$

Image domain can be defined in two ways, the *image-centered domain* and the *object-centered domain.* The *image-centered domain* uses the image coordinates as

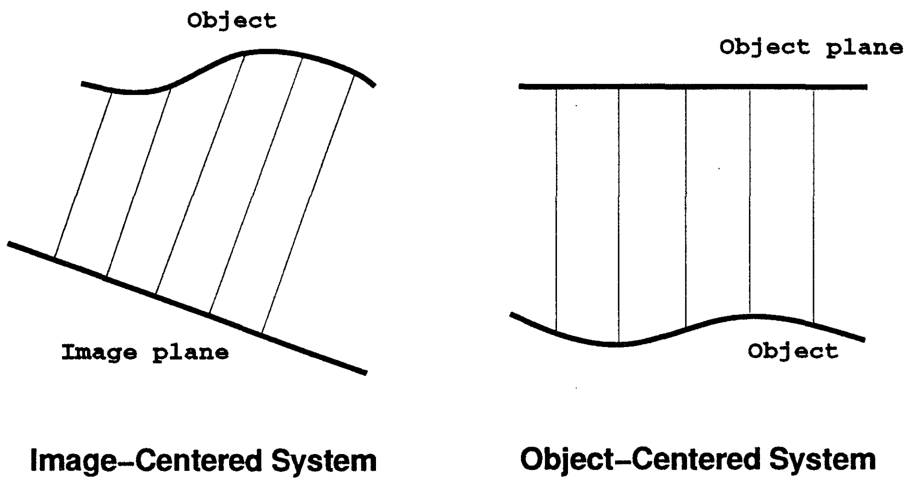


Figure 2-1: Domain System Choices

the fundamental domain and assigns the depth(or height) value to each surface point that projects to each image position. The *object-centered domain* uses the object coordinate system as the fundamental domain and assigns a surface depth(or height) information to each point in the domain. (See Figure 2-1)

Once the domain is selected, we still have the freedom to choose between *perspective* and *othographic* projection.

2.1.1 Perspective Projection

Consider an ideal pinhole sits in between an object and an image plane (See Figure 2-2). Since the light travels in straight lines, each point in the image corresponds to a particular direction defined by a ray from the point through that pinhole. This is perspective projection.

The optical Axis thus defined to be perpendicular from the pinhole to the image plane. A Cartesian coordinate system is setup with the origin at the pinhole and *z-axis* along optical axis pointing to the image. The z component of the coordinates of the image will therefore be negative.

For each point P on some object in front of the camera will appear P' on the

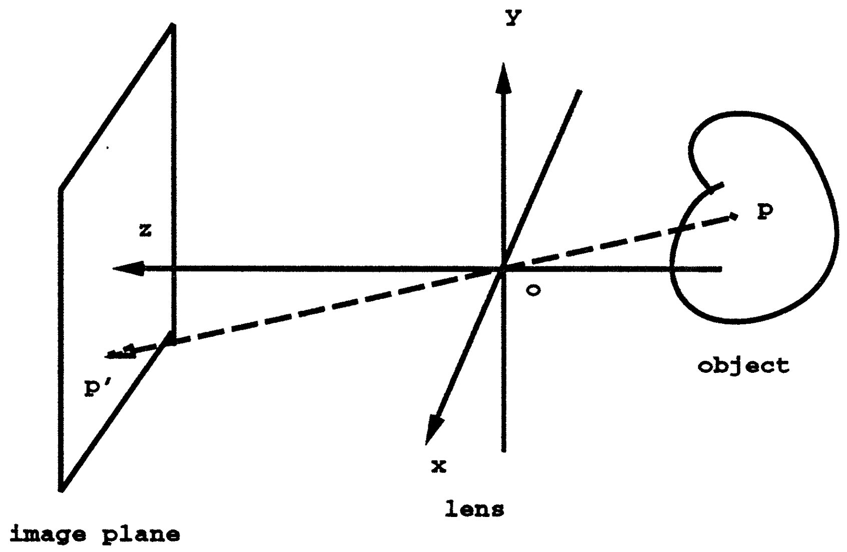


Figure 2-2: Perspective Projection

image. (See Figure 2-2). Let $\mathbf{r} = (x, y, z)^T$ denotes P, and $\mathbf{r}' = (x', y', f)^T$ [1] denotes P'. From geometry optics, we know:

$$r = -z \sec\alpha = -(\mathbf{r} \cdot \hat{\mathbf{z}}) \sec\alpha \tag{2.2}$$

where $\hat{\mathbf{z}}$ is the unit vector along the optical axis. The length of $\mathbf{r}'$ is

$$r' = f \sec\alpha \tag{2.3}$$

$$\frac{1}{f}\mathbf{r}' = \frac{1}{\mathbf{r} \cdot \hat{\mathbf{z}}}\mathbf{r} \tag{2.4}$$

or

[1] $z' = f$, f is the focal length.

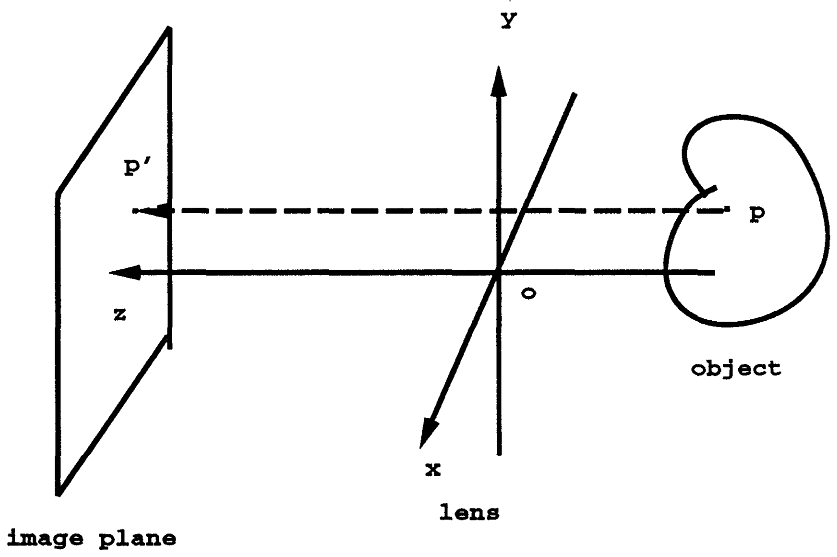


Figure 2-3: Orthographic Projection

$$\frac{x'}{f} = \frac{x}{z} \quad and \quad \frac{y'}{f} = \frac{y}{z} \tag{2.5}$$

2.1.2 Othographic Projection

Consider that if we put the image plane at $z = z_0$, and define *lateral magnification* m as the ratio of the distance between two points measured in the image to the distance between the corresponding points on the plane.

$$m = \frac{f}{-z_0} \tag{2.6}$$

where $-z_0$ is the distance of the plane from the pinhole.

A small object at an average distance $-z_0$ will give rise to an image which is magnified by m. Let the *depth* be the distance from the object to the camera. The magnification is approximately constant when the *depth* range of the scene is relative small to the average distance. Then

$$x' = -mx \quad and \quad y' = -my \tag{2.7}$$

For convenience, we can set $m = -1$, and

$$x' = x \quad and \quad y' = y \tag{2.8}$$

This is the othographic projection. It can be depictured as the ray runs parallel to the optical axis (See Figure 2-3).

If we choose *image-centered* domain, and orthographic projection is used, the projection map is straightforward. However, if perspective projection is used, then this mapping can become quite complex, for example, surface normal calculation. If we choose *object-centered* domain, both perspective and orthographic projection are straightforward. However, the projected points in general won't map to the center of each pixel. Thus interpolation is needed to obtain values at each pixel.

2.2 Shape From Shading

2.2.1 Image Formation Process

Shape from Shading problem is to generate three dimensional topography information from two dimensional image. It is crucial to understand how images are formed. This process can be viewed as two stages, object radiance and image formation. This process also depends on four factors:

1. object irradiance.

2. reflectance map.

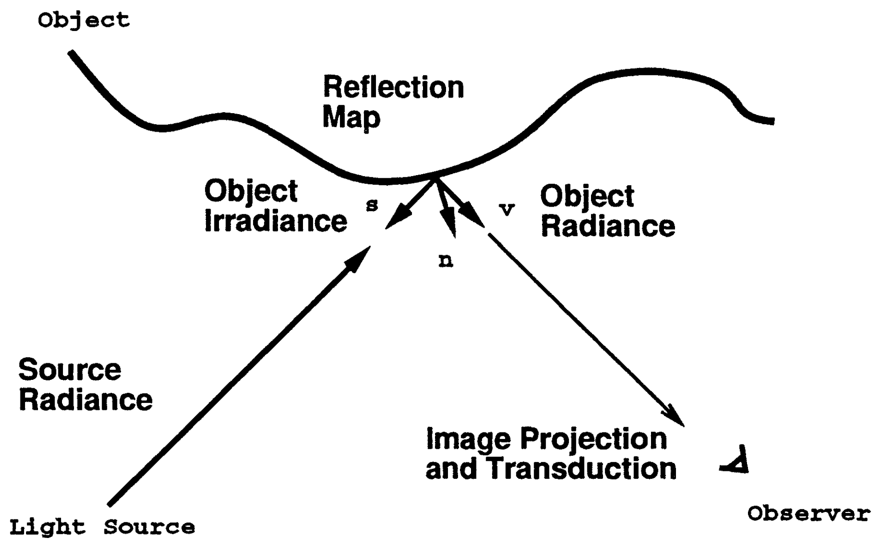


Figure 2-4: Image Formation Process

3. image projection.

4. image transduction.

2.2.2 Object radiance stage

The amount of light falling on a surface is called the *irradiance.* At each point ξ on the surface, and for each direction $\hat{s}$, the irradiance distribution function is $E(\xi, \hat{s})$. The amount of light radiated from a surface is called *radiance.* Obviously, the object radiance depends on the object irradiance and the surface reflectance properties. The surface reflectance properties is the physical character of the surface, independent of the irradiance, and can be described by *Bidirectional Reflectance Distribution Function* (BRDF).[2] At the point ξ, the BRDF $f(\xi, \hat{n}, \hat{v}, \hat{s})$ relates the brightness of the surface patch with normal $\hat{n}$ illuminated from the direction $\hat{s}$ and as seen from the direction $\hat{v}$. Using these two distribution functions, the radiance function can be written as:

[2]For more information on BRDF, see[8, p. 209]

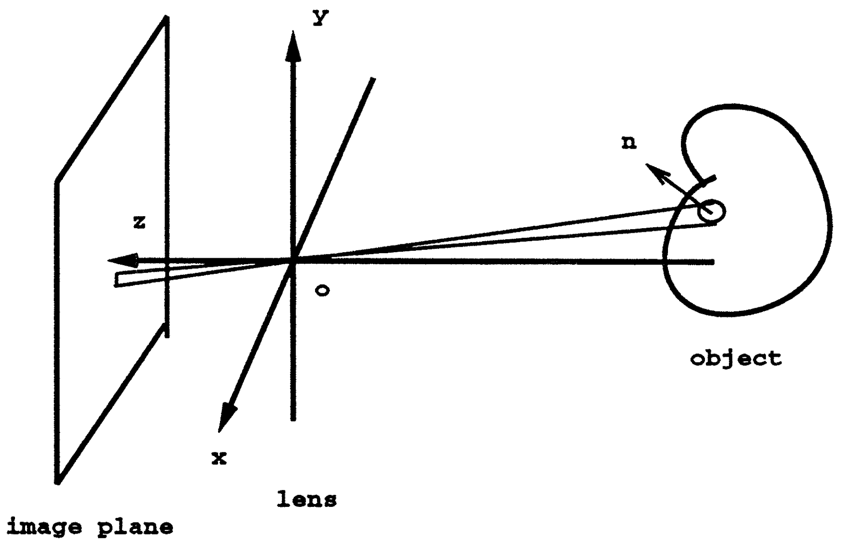


Figure 2-5: Image Irradiance Process

$$L(\xi, \hat{n}, \hat{v}) = \int\int_H f(\xi, \hat{n}, \hat{v}, \hat{s}) E(\xi, \hat{s})(\hat{s} \cdot \hat{n}) d\omega(\hat{s}) \tag{2.9}$$

where H is the hemisphere of possible light sources directions for the patch at surface point ξ, and $d\omega(\hat{s})$ is the solid angle subtended in the direction $\hat{s}$. Representing $\hat{s}$ in spherical coordinates, above equation becomes

$$L(\xi, \hat{n}, \hat{v})1z = \int_{-\pi}^{\pi} \int_0^{\pi/2} f(\xi, \hat{n}, \hat{v}, \hat{s}) E(\xi, \hat{s}) \sin\theta \cos\theta d\theta d\phi \tag{2.10}$$

Two common reflectance models are the *Lambertian* model and the specular model. The *Lambertian* model deals matter surface. An ideal *Lambertian* surface is one that appears equally bright from all viewing direction and reflects all incident light, which means the BRDF is independent of source direction $\hat{s}$, surface normal direction $\hat{n}$ and viewing direction $\hat{v}$. Follow this definition, we get

$$f_{Lambertian}(\xi, \hat{n}, \hat{v}, \hat{s}) = \frac{1}{\pi}\rho(\xi) \tag{2.11}$$

where $\rho(\xi)$ is the surface albedo, or the fraction of light re-emitted by the surface. Evaluate 2.10 for a *Lambertian* surface illuminated by a point source at infinity, the surface radiance is

$$L_{Lambertian}(\xi, \hat{\mathbf{n}}, \hat{\mathbf{v}}) = \frac{\lambda\rho(\xi)}{\pi}(\hat{\mathbf{s}}_0 \cdot \hat{\mathbf{n}}) \tag{2.12}$$

Specular model deals with metallic surfaces. All light from the direction $\hat{\mathbf{s}}_0$ is reflected into the direction $2(\hat{\mathbf{n}} \cdot \hat{\mathbf{s}}_0)\hat{\mathbf{n}} - \hat{\mathbf{s}}_0$, so that the BRDF is

$$f_{Specular}(\xi, \hat{\mathbf{n}}, \hat{\mathbf{v}}, \hat{\mathbf{s}}) = \rho(\xi)\hat{\delta}(\hat{\mathbf{v}}, 2(\hat{\mathbf{n}} \cdot \hat{\mathbf{s}}_0)\hat{\mathbf{n}} - \hat{\mathbf{s}}_0) \tag{2.13}$$

For a point source at infinity, the surface radiance for a specular surface is found to be

$$L_{Specular}(\mathbf{R}, \hat{\mathbf{n}}, \hat{\mathbf{v}}) = \lambda\rho(\xi)\hat{\delta}(\hat{\mathbf{v}}, 2(\hat{\mathbf{n}} \cdot \hat{\mathbf{s}}_0)\hat{\mathbf{n}} - \hat{\mathbf{s}}_0) \tag{2.14}$$

A radiance function representing most surface is a linear combination of these two plus a constant ambient term, which models haze and atmospheric reflection, as well as the effects of a uniformly distributed light source.

$$L = \alpha L_{Lambertian} + \beta L_{specular} + \gamma \tag{2.15}$$

where α, β and γ are scalars so that $\alpha + \beta + \gamma = 1$.

Many vision researchers prefer to use a representation of the surface radiance based on a global coordinate system. This is called the Reflectance function. Given Known surface properties and a known light source, the reflectance function, $R(\xi, \hat{\mathbf{n}}, \hat{\mathbf{v}})$ is defined using the corresponding surface radiance via a change in coordinates,

$$R(\xi_G, \hat{\mathbf{n}}_G, \hat{\mathbf{v}}_G) = L(\xi, \hat{\mathbf{n}}, \hat{\mathbf{v}}) \tag{2.16}$$

where ξ_G, $\hat{\mathbf{n}}_G$, and $\hat{\mathbf{v}}_G$ are the surface position, normal direction vector, and viewing direction vector.

2.2.3 Image Generation Stage

In earlier section, we introduced the perspective and orthographic projection. We also need to know how the brightness of the image is affected by the projection. Assuming perfect lens, the irradiance from a patch on object is

$$E(\mathbf{r}) = L(\xi(\mathbf{r}), \hat{\mathbf{n}}, \hat{\mathbf{v}}(r))\frac{\pi}{4}(\frac{d}{f})^2 \cos\alpha^4 \tag{2.17}$$

where $L(\xi(\mathbf{r}), \hat{\mathbf{n}}, \mathbf{v}\hat{(}\mathbf{r}))$ is the radiance of the corresponding object patch, d is the diameter of the lens, and α is the off-axis angle of the projecting ray. Here, $\hat{\mathbf{v}}$ and ξ are functions of $\mathbf{r}$ via the projection from image points $\mathbf{r}$ to object points. Equivalently, this equation can be restated using reflectance function instead,

$$E(\mathbf{r}) = R(\xi(\mathbf{r}), \hat{\mathbf{n}}, \hat{\mathbf{v}}(r))\frac{\pi}{4}(\frac{d}{f})^2 \cos\alpha^4 \tag{2.18}$$

There is one more factor in image generation stage. Light will be converted into digital signal, during the conversion, there is distortion of the image. The effect can be removed by calibration so that the measured image irradiance can be related to object radiance in a straightforward way. With a perfect calibration, the image irradiance equation can be put into a very simple form,

$$E(\mathbf{r}) = R(\xi(\mathbf{r}), \hat{\mathbf{n}}, \mathbf{v}\hat{(}\mathbf{r})) \tag{2.19}$$

where the constant term and $\cos\alpha^4$ can be set to 1 during calibration.

2.3 Stereo

For normal stereo situations, the camera are close together and both pictures are taken simultaneously or nearly so. The stereo images that result look very similar,

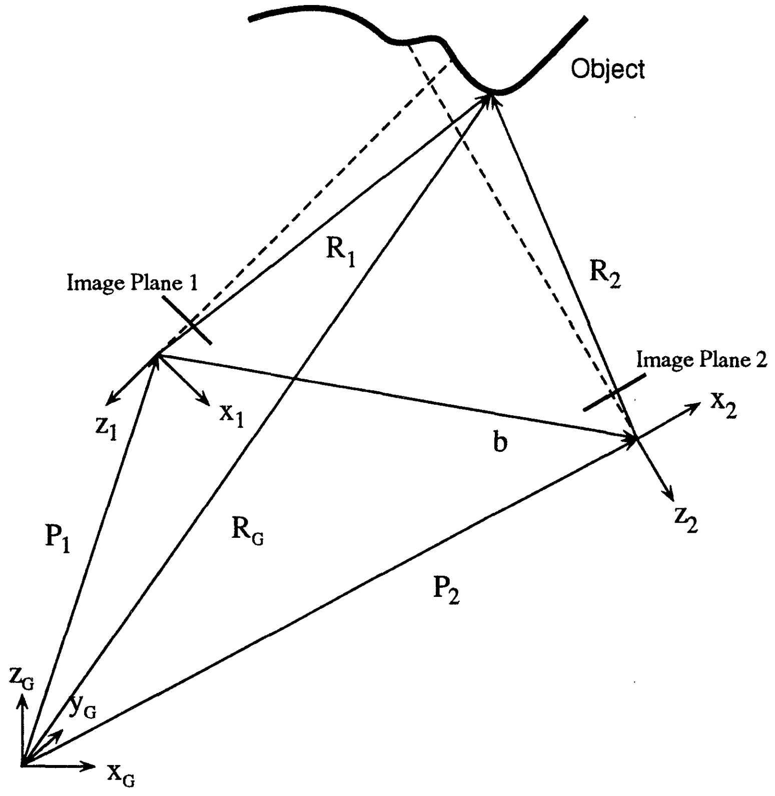


Figure 2-6: Stereo Geometry

mostly differing in a shift of objects in each image caused by perspective projection. the difference in the shift of an object point in the left image and the right image is called the *disparity.* If the relative position of the cameras is known, and it is known the correspondence relation of each pixel in the right and left image, it is possible to determine the depth directly for the surface points that projects to those pixels.[8]

To determine the depth directly from the disparity, See Figure 2-6. If the corresponding points in each image map to rays intersect, we can use geometry to determine the depth of the point that is halfway between the rays at their closest approach.

First find the relationship between the two camera coordinate system, Suppose, we know the position of the principal point of each camera in some global coordinate system, $\mathbf{P}_1$ and $\mathbf{P}_2$, and we also know the rotational transformation matrices from each local camera coordinate system to the global coordinate system, $\mathbf{T}_1$ and $\mathbf{T}_2$, the coordinates of the point ξ in the two camera coordinate system is

$$\begin{aligned}\mathbf{R}_1 &= T_1^{-1}(\xi - \mathbf{P}_1)\\ \mathbf{R}_2 &= T_2^{-1}(\xi - \mathbf{P}_2)\end{aligned} \tag{2.20}$$

By moving ξ from the above equations and defining $\mathbf{b} = \mathbf{P}_2 - \mathbf{P}_1$, the relationship between a point in Camera Coordinate System 1 and a point in Camera Coordinate System 2 is

$$\mathbf{R}_1 = T_1^{-1}\mathbf{b} + T_1^{-1}T_2\mathbf{R}_2 \tag{2.21}$$

Now, determine the relationship between disparity and depth. Suppose we are given image points, $\mathbf{r}_1 = (x_1, y_1, f)^T$ and $\mathbf{r}_2 = (x_2, y_2, f)^T$ (one in each image), that correspond to the same surface point, then the best estimate for the surface position can be found by finding the point on each ray (along $\mathbf{r}_1$ and $\mathbf{r}_2$) where the distance between the ray is minimized. That is the problem

$$\min_{s,t} \|\mathbf{b} + t\mathbf{r}_2 - s\mathbf{r}_1\|^2 \tag{2.22}$$

must be solved where s and t are scalar parameters. For now assume all the vectors are given on the same basis.

By differentiating the above equation with respect to s and t, setting the resulting equation to zero, and solving, it is found that the minimum occurs when

$$\begin{aligned} s &= \frac{(\mathbf{r}_1 \cdot \mathbf{r}_2)(\mathbf{b} \cdot \mathbf{r}_1) - (\mathbf{r}_1 \cdot \mathbf{r}_2)(\mathbf{b} \cdot \mathbf{r}_2)}{(\mathbf{r}_1 \cdot \mathbf{r}_1)(\mathbf{r}_2 \cdot \mathbf{r}_2) - (\mathbf{r}_1 \cdot \mathbf{r}_2)^2} = \frac{(\mathbf{r}_2 \times \mathbf{b}) \cdot (\mathbf{r}_2 \times \mathbf{r}_1)}{\|\mathbf{r}_2 \times \mathbf{r}_1\|}\\ t &= \frac{(\mathbf{r}_1 \cdot \mathbf{r}_2)(\mathbf{b} \cdot \mathbf{r}_1) - (\mathbf{r}_1 \cdot \mathbf{r}_2)(\mathbf{b} \cdot \mathbf{r}_2)}{(\mathbf{r}_1 \cdot \mathbf{r}_1)(\mathbf{r}_2 \cdot \mathbf{r}_2) - (\mathbf{r}_1 \cdot \mathbf{r}_2)^2} = \frac{(\mathbf{r}_1 \times \mathbf{b}) \cdot (\mathbf{r}_2 \times \mathbf{r}_1)}{\|\mathbf{r}_2 \times \mathbf{r}_1\|} \end{aligned} \tag{2.23}$$

if $\mathbf{r}_1$, $\mathbf{r}_2$ and $\mathbf{b}$ are coplanar then the above set of equations is just a fancy representation of the sines.

The global position of the point halfway between the two rays at thee closest approach is

$$\begin{aligned} \xi &\approx \mathbf{P}_1 + s\mathbf{r}_1 + \frac{1}{2}(\mathbf{b} + t\mathbf{r}_2 - s\mathbf{r}_1), \\ &= \mathbf{P}_1 + \frac{1}{2}\mathbf{b} + \frac{1}{2}\frac{(T_2\mathbf{r}_2 \times T_1\mathbf{r}_1)}{\|T_1\mathbf{r}_2 \times T_2\mathbf{r}_1\|^2}[(T_2\mathbf{r}_2 \times \mathbf{b})T_1\mathbf{r}_1 + (T_1\mathbf{r}_1 \times \mathbf{b})T_2\mathbf{r}_2] \end{aligned} \tag{2.24}$$

where $\mathbf{P}_1$, $\mathbf{P}_2$ and $\mathbf{b}$ are given in global coordinates and $\mathbf{r}_1$, $\mathbf{r}_2$ are given in the appropriate local camera coordinate system.

The equations simplify greatly if $\mathbf{r}_1$, $\mathbf{r}_2$ and $\mathbf{b}$ are coplanar, the cameras are aligned so that the optical axes are in the direction $-\hat{\mathbf{z}}$ and $\hat{\mathbf{x}}$ is along $\mathbf{b}$. The stereo geometry is commonly assumed to exist for most binocular stereo algorithms. With these restrictions, the above equation becomes

$$\xi = \mathbf{P}_1 + \frac{b}{(x_2 - x_1)}\mathbf{r}_1 \tag{2.25}$$

where $\mathbf{r}_1$ and $\mathbf{r}_2$ are defined as earlier, and $\mathbf{b} = (b, 0, 0)^T$. the quantity $(x_2 - x_1)$ in the above equation is the disparity mentioned earlier. Note that the disparity can be mapped directly into depth only in the special case. For more general case, 2.24 must be used.

2.4 Photo-Topography

Now we have described image formation process and stereo system, we can formulate the photo-topography problem.

What we know from the introduction of Viking Project in previous chapter, there are two images of an area on the planet surface, taken at two different times from two different positions and angles. The objective is to determine the topography of the planet surface.

The images in a photo-topography problem are taken from two different vantage points (See Figure 1-1). The cameras are often far apart and has pictures that are taken at different time. In this case, it is very likely for the two images to look very different. this makes the correspondence problem very hard. As opposing the small baseline disparity and the same position and time of two images.

The solution is constrained by geometry and the image generation process. Specifically each image is constrained by the perspective projection equation, 2.4 and the image irradiance equation, 2.19 and the stereo constrained equations 2.20.

Combining these equations we find that the photo-topography problem is constrained such that

$$\begin{aligned} E^{(1)}(\mathbf{r}_1) &= R^{(1)}(\xi, \hat{\mathbf{n}}, -T_1\mathbf{r}_1) \\ E^{(2)}(\mathbf{r}_2) &= R^{(2)}(\xi, \hat{\mathbf{n}}, -T_2\mathbf{r}_2) \end{aligned} \tag{2.26}$$

where

$$\begin{aligned} \mathbf{r}_1 &= \frac{fT_1^{-1}(\xi - \mathbf{P}_1)}{T_1^{-1}(\xi - \mathbf{P}_1) \cdot T_1\hat{\mathbf{z}}_1} \\ \mathbf{r}_2 &= \frac{fT_2^{-1}(\xi - \mathbf{P}_2)}{T_2^{-1}(\xi - \mathbf{P}_2) \cdot T_2\hat{\mathbf{z}}_2} \end{aligned} \tag{2.27}$$

$E^{(1)}$ and $E^{(2)}$ are the image brightness measured in the first and the second cameras respectively, and $R^{(1)}$ and $R^{(2)}$ are the reflectance maps based on the first and second light source positions. As before, ξ is the surface position in global coordinates.

It is important to review the assumptions behind these equations. The perspective projection equations assume perfect lenses and perfect knowledge of the camera principal points and optical axes. the surface radiance equation assumes we have perfect knowledge of the surface reflectance properties, light source directions, and all surface points as visible from both cameras(i.e., there is no self occlusions). The simplified form of the image irradiance equation assumes we either have a perfect sensor or we can perfectly calibrate the sensor to remove any abnormalities from the sensor & lens combination. The stereo equations assume we know the relative position and orientation of the two camera perfectly. the only assumption that are truly artificial are the assumptions of perfect knowledge of the reflectance maps and the the assumption of no self-illumination. With more careful measurements and more expensive equipment, it is possible to approach perfect knowledge of other assumptions. The assumption that all surface points be visible merely restricts the roughness of the surface that this research is applicable to.

2.5 Simplification

There are several simplifications that can be made to the equation in the previous section to make them easier to solve.

2.5.1 Special Global Coordinate System

So far all the equations have been written for any global coordinate system. I would like to restrict the equations to a particular global coordinate system, as shown in Figure 2-7. This coordinate system is defined as below:

1. Place the origin of the global coordinate system half way between the principal points of the two cameras.

2. Choose the $\hat{\mathbf{x}}_G$ direction along the line connecting the two cameras. $\hat{\mathbf{x}}_G = \mathbf{b}/||\mathbf{b}||$.

3. Choose the $\hat{\mathbf{z}}_G$ in the direction of the average of the optical axis directions of the two cameras projected into the plane perpendicular to $\mathbf{x}_G$, then

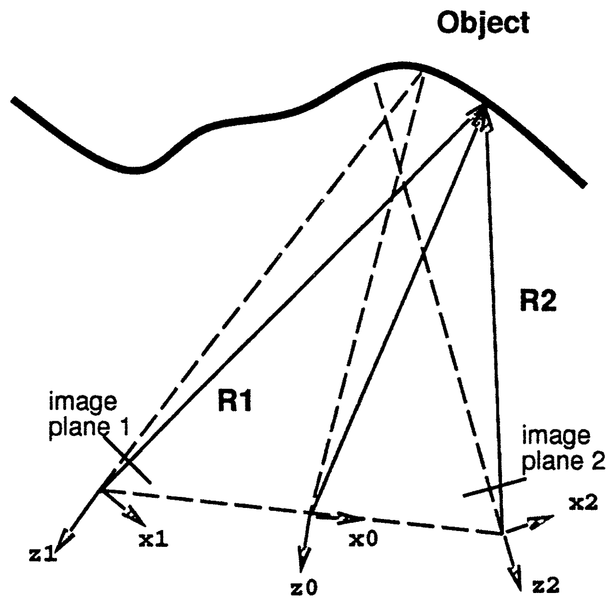


Figure 2-7: Global Coordinate System

$$\hat{\mathbf{z}}_G = \frac{(\hat{\mathbf{z}}_1 - \hat{\mathbf{z}}_1 \cdot \hat{\mathbf{x}}_G) + (\hat{\mathbf{z}}_2 - \hat{\mathbf{z}}_2 \cdot \hat{\mathbf{x}}_G)}{\|(\hat{\mathbf{z}}_1 - \hat{\mathbf{z}}_1 \cdot \hat{\mathbf{x}}_G) + (\hat{\mathbf{z}}_2 - \hat{\mathbf{z}}_2 \cdot \hat{\mathbf{x}}_G)\|}. \tag{2.28}$$

4. Choose $\hat{\mathbf{y}}_G$ in the direction of $\hat{\mathbf{z}}_G \times \hat{\mathbf{z}}_G$ in order to create a right handed coordinate system.

5. Also set up a virtual image plane with f = 1.

Then $\mathbf{P}_1 = -\mathbf{b}/2, \mathbf{P}_2 = \mathbf{b}/2$, and $\mathbf{b} = (b, 0, 0)^T$.

2.5.2 Removing the View Direction Dependence

A common simplification for computer vision is the assumption of a *Lambertian* reflectance map. Since a *Lambertian* surface reflects light equally in all directions, we see from Equation 2.12 that the radiance function does not depend on viewing direction. Thus the dependence on $\hat{\mathbf{v}}$ can be removed from all equations.[3]

[3]This is true for any reflectance function which is viewing independent, not just *Lambertian reflectance.*

Note that the viewing direction could also have been removed from the equations by using orthographic projection for the reflectance function while retaining perspective projection for everything else. In this case the viewing direction would be constant for each camera and its effect could be subsumed into the reflectance map. Doing this would, of course, introduce an error into the calculations since orthographic and perspective projection do not produce the same viewing direction in general. This error would be small for planet photo-topography since the cameras are so far away from the surface. The large viewing distance results a very small relative depth change $\Delta Z/Z_0$. This is especially true when the field of view is small.

2.5.3 Constant Albedo

So far the equations have terms that denote position on the surface ξ. The main reason for this dependence is to take into account varying albedo, varying reflectance properties or both. To simplify, we can assume that the reflectance properties, albedo, or both are constant across the surface. Assuming the reflectance but not albedo, are constant across the surface results in a reflectance function that is separated,

$$R(\xi, \hat{\mathbf{n}}, \hat{\mathbf{v}}) = \rho(\xi)\bar{R}(\hat{\mathbf{n}}, \hat{\mathbf{v}}) \tag{2.29}$$

where $\bar{R}(\hat{\mathbf{n}}, \hat{\mathbf{v}})$ is the reflectance function for a surface with uniform albedo, no interflection, and no mutual occlusion. As for R, any light source effects are included in $\bar{R}$. When both the reflectance and albedo are constant, the dependence of the reflectance map on surface position can be removed,

$$R(\xi, \hat{\mathbf{n}}, \hat{\mathbf{v}}) = \bar{R}(\hat{\mathbf{n}}, \hat{\mathbf{v}}) \tag{2.30}$$

Combined with either *Lambertian reflectance* [4] or orthographic projection, the dependence on $\hat{\mathbf{v}}$ can be removed,

[4]Or any viewing independent reflectance function

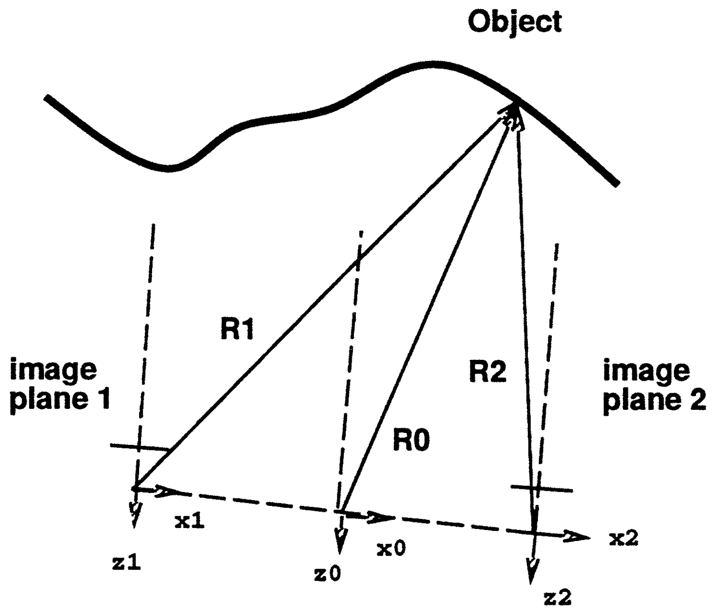


Figure 2-8: Aligned Coordinate System

$$R(\xi, \hat{n}, \hat{v}) = \bar{R}(\hat{n}) \tag{2.31}$$

The simplification restricts the applicability of the algorithm presented in next chapter so that only uniformly colored surface patches have the possibility of being estimated correctly. When algorithms based on this simplification are applied to images that violate these simplifications, we would expect errors at the transition between different colored parts of the surface, within differently colored areas or both.

2.5.4 Aligned Cameras

The Final simplification that can be made is to align the cameras so that their optical axes are parallel, which will also be parallel to the global coordinate system, As shown in Figure 2-8. When the cameras are aligned, the rotational transformations T_1 and T_2 are identity transformations, which simplifies the stereo equations to

$$\mathbf{R}_1 = \xi - \mathbf{P}_1,$$
$$\mathbf{R}_2 = \xi - \mathbf{P}_2. \tag{2.32}$$

While this situation is very unrealistic for the photo-topography problem, any set of images can be re-projected into this coordinate system.

2.5.5 Summary of Simplified equations

The rest of the thesis is based on equations that take into account all the above simplifications. in summary, they are:

1. A special global coordinate system that is halfway between the two camera position.

2. All surface points are assumed to be visible from the two cameras.

3. The reflectance properties of the surface are assumed to be constant and *Lambertian* allowing the viewing direction dependence to be dropped from the reflectance equations. In addition, it is assumed that there is no interflection between different parts of the surface.

4. The surface is assumed to have constant albedo allowing the position dependent term to be dropped.

5. The cameras optical axis are assumed to be aligned with each other allowing the rotational transformation to be set to identity.

Apply all the above, the set of equations are:

$$E^{(1)}(\mathbf{r}_1) = R^{(1)}(\hat{\mathbf{n}})$$

$$E^{(2)}(\mathbf{r}_2) = R^{(2)}(\hat{\mathbf{n}}) \tag{2.33}$$

where

$$\begin{aligned}\mathbf{r}_1 &= \frac{f(\xi + \mathbf{b}/2)}{(\xi + \mathbf{b}/2)\cdot\hat{\mathbf{z}}_1},\\ \mathbf{r}_2 &= \frac{f(\xi - \mathbf{b}/2)}{(\xi - \mathbf{b}/2)\cdot\hat{\mathbf{z}}_2},\end{aligned} \tag{2.34}$$

If we define $z = \xi \cdot \hat{\mathbf{z}}_G$ and $\mathbf{r} = f\xi/z$, then the constraint function can be written as

$$\begin{aligned}E^{(1)}(\mathbf{r} + \frac{f\mathbf{b}}{2z}) &= R^{(1)}(\hat{\mathbf{n}})\\ E^{(2)}(\mathbf{r} - \frac{f\mathbf{b}}{2z}) &= R^{(2)}(\hat{\mathbf{n}})\end{aligned} \tag{2.35}$$

It is found more convenient in subsequent chapters to use a slightly different notation. Use explicitly $\mathbf{r} = (x, y, f)$ and the normal vector $\hat{\mathbf{n}}$ is parameterized using gradient component p and q,

$$\hat{\mathbf{n}} = \frac{(-p, -q, 1)}{\sqrt{p^2 + q^2 + 1}} \tag{2.36}$$

where

$$\begin{aligned}p &= \frac{fz_x}{xz_x + z}\\ q &= \frac{fz_y}{yz_y + z}\end{aligned} \tag{2.37}$$

The photo-topography equations then become

$$E^{(1)}(x + \frac{fb}{2z}, y) = R^{(1)}(p, q)$$
$$E^{(2)}(x - \frac{fb}{2z}, y) = R^{(2)}(p, q) \tag{2.38}$$

2.6 Camera Calibration

In order to relate positions in the image direction vectors in 3-D space, the origin of the camera coordinate system must be known. Finding this origin is the classical *interior orientation* problem. As assumed in section 2.2.3, this origin is the projection of the principal point in the image plane. In the special coordinate system, see Figure 2-8, the position of each camera coordinate system origin can be specified by a vector $\mathbf{v} = (v_x, v_y, f^T)$. these vector specify the offset of the camera coordinate origin for each image. Suppose $\mathbf{v}_0$ is the offset to an object point in the global coordinate system, then the offset to this same point in the camera images are

$$\mathbf{v}_1 = \mathbf{v}_0 + \frac{fb}{2z_0}, \tag{2.39}$$
$$\mathbf{v}_2 = \mathbf{v}_0 - \frac{fb}{2z_0}. \tag{2.40}$$

the values of $\mathbf{v}_1$ and $\mathbf{v}_2$ can be quite large in the aligned coordinate system indicating that the images must be shifted far away from the camera coordinate system origin. While this is not possible physically, it is a consequence of re-projecting real images into the aligned coordinate system.

Chapter 3

Fusion of Shape from Shading and Stereo

In this chapter, I will discuss the Fusion strategy proposed by Clay Thompson, in his Ph.D. Thesis(See [24]). I will focus on the most efficient and robust *z-only* algorithm in solving photo-topography problem. The basic idea is to closely couple the solution of Shape-from-Shading and stereo in a variational approach. The important point is to use each algorithms strength to compensate each one's weakness. (see table 3.1)

Naturally, one way of fusing two or more algorithms is to run each fused algorithms separately on the image, and then combine the output to generate a single solution. (See Figure 3-1) This approach is easy to understand and implement, since existing algorithm can be put together in a *ad-hoc* manner, but it ignores the interconnection embedded in the algorithms and the information coupling each algorithm.

The variational approach, on the other hand, closely couples the algorithms together. (See Figure 3-2) This can be achieved by formulating a combined cost function based on the cost function of each fused algorithms. The result is a combined optimization problem which takes into account both the explicit and implicit constraints between the methods. Variational methods, by their nature, can exploit any orthogonalities in the methods. It then has the potential to create robust, well performing

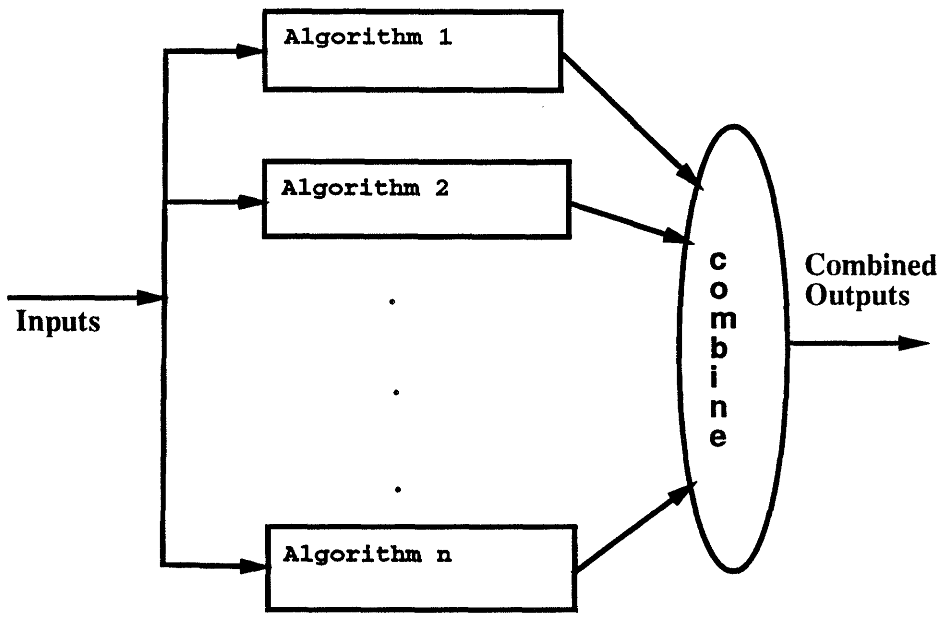


Figure 3-1: Module Based Fusion

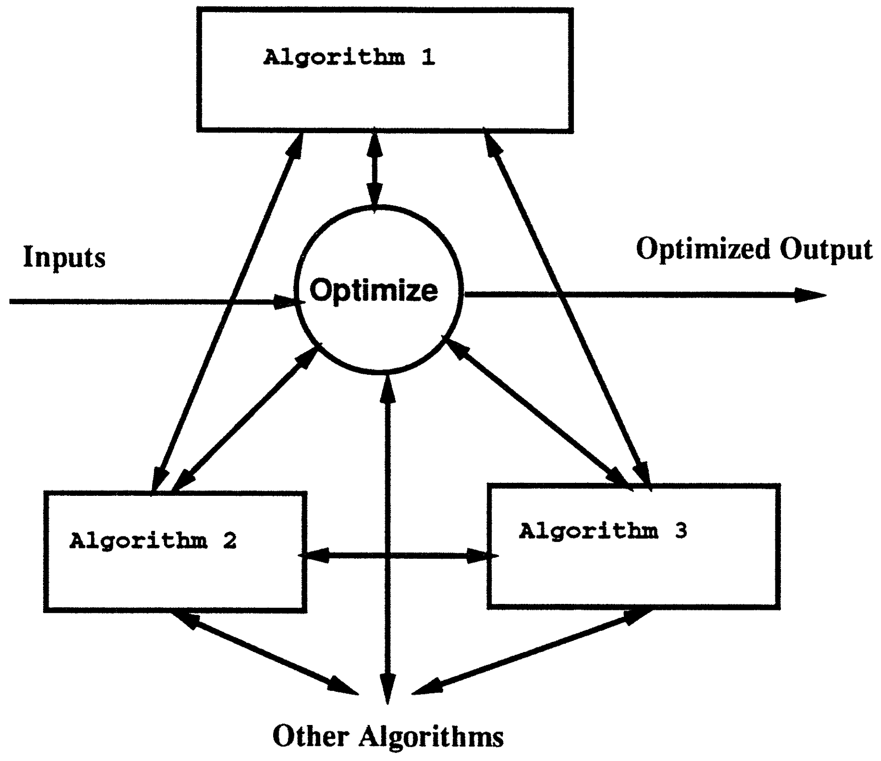


Figure 3-2: Variational methods Based Fusion

combinations of algorithms which can be applied to a wide range of input images.

3.1 Fusion Strategy

The planetary images, particularly Viking images provide us two sources of information.

1. Shading Information: the gray levels in each image are an indication of the surface orientation with respect to the light source.

2. Stereo Information: assuming corresponding pixel in each image can be matched up, the stereo information can be used to recover the shape.

	Shading	Stereo	Lighting
Shape From Shading	Shape From Shading	Surface Constraint	Surface Orientation Constraint
Binocular Stereo	Correspondence Constraint	Binocular Stereo	Correspondence Constraint
Photometric Stereo	Correspondence Constraint	Correspondence Constraint	Photometric Stereo

Table 3.1: Photo-Topography problem source and constraint matrix

3. Photometric Information: the gray levels of corresponding pixel constrain the set of possible surface orientation, since the images are taken at with two different light source position.

Another important point is that the shading and stereo information are independent and mutual compensating. independent means we can differentiate the contribution from each source of information. For example, The shading information is the strongest when shading is smooth, while stereo information is the strongest near surface discontinuities, where feature dominates, and when cameras are widely apart. The photometric information is strongest when the light source positions are widely separated.

3.2 *z-only* DFSS Algorithm

z-only algorithm estimates everything in a single global coordinate system which is defined to be half way between the two camera positions. Figure 3-3 shows the tree diagram of the flow of this algorithm. The current estimate of the surface height z is used to project points in the global coordinate system to points in each image using perspective projection. These points won't in general land on a pixel center so some type of interpolation is used to determined the value of the image at the projected points. This interpolated image F is then compared to a computed image based on

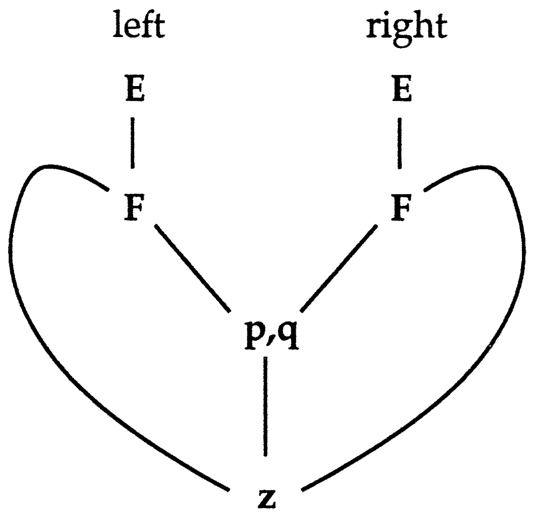


Figure 3-3: Centralize Algorithm Tree Based on Disparity

the current estimate of the surface. The error is used to update p, q, and ultimately z.

z-only algorithm uses hard integrability constraints. With hard integrablity we are guaranteed that any solution obtained will be feasible. The trade-off is that *z-only* algorithm will have less degree of freedom and more susceptible to local minima.

3.2.1 Variables

In *z-only* algorithm, the depth map z is the only optimization variable, thus its name *z-only* algorithm. The surface gradient components p and q are computed directly from the depth map. The photo-topography images, camera geometry and surface reflectance function are inputs to the cost function.

3.2.2 Cost Function

The cost function is formed by integrating the squared photo-topography error introduced by the current estimate for z, together with a penalty function for departure from smoothness.

The penalty function is mainly used to guide the solution towards the minimum. In practice, the smoothness weighting parameter λ is slowly reduced towards zero as the algorithm converges.

$$\begin{aligned}\min_z J = \ & \tfrac{1}{2}\int\int[(E^{(1)}(x+\tfrac{fb}{2z},y)-R^{(1)}(p,q))^2+(E^{(2)}(x-\tfrac{fb}{2z},y)-R^{(2)}(p,q))^2 \\ & +\lambda(z_{xx}^2+2z_{xy}^2+z_{yy}^2)]dxdy. \qquad (3.1)\end{aligned}$$

The smoothness term is based on the second variation. It is equivalent to $p_x^2 + p_y^2 + q_x^2 + q_y^2$ when $z_x \approx fp/z_0$ and $z_y \approx fp/z_0$ where z_0 is the nominal depth[1].

The cost function above is continuous and must be discretized before it can be optimized. The process is an approximation one. using finite difference methods, since the images are in digital form, each pixel represents the average of the brightness falling within the sensitive area of the corresponding photosensor. It then makes sense to approximate the values for p, q and z as arrays of gradient components or surface depth.

Using an array of z, the cost function can be written as:

$$\begin{aligned}\min_z J = \ & \tfrac{1}{2NM\epsilon^2}\textstyle\sum_{x,y\in D}[(F^{(1)}(x,y)-R^{(1)}(p,q))^2+(F^{(2)}(x,y)-R^{(2)}(p,q))^2 \\ & +\lambda(z_{xx}^2+2z_{xy}^2+z_{yy}^2)]dxdy. \qquad (3.2)\end{aligned}$$

[1]The approximations are valid when the field of view is small, the image is centered around the camera's principal point, and the depth of field relative to the nominal depth is small.

where D is the discrete image domain of the underlying variables in the global coordinate system, N and M are the row and column dimensions of the discrete domain, and ϵ is the grid spacing(assume equal spacing in both x and y directions). $F^{(i)}(x,y)$ are interpolated from the input image $E^{(i)}(x,y)$

$$F^{(i)}(x,y) = E^{(i)}(\bar{x},y) + (x \pm \frac{fb}{2z} - \bar{x})[E^{(i)}(\bar{x}+1,y) - E^{(i)}(\bar{x},y)] \tag{3.3}$$

where

$$\bar{x} = floor(x \pm \frac{fb}{2z}). \tag{3.4}$$

The $floor(x)$ function returns the greatest integer towards minus infinity.

Matched grid is used to implement the cost function. In this implementation, p, q and z are chosen all to be the same size as the image arrays. In such a representation, all the functions are sampled on the same grid, thus the name *matched-grid representation.* This approach uses vertex-centered surface derivatives that are valid at each vertex of the z grid. Since p and q are the same size as z, some type of approximation must be made at the array edges. Bicubic interpolation is used to extrapolate the estimates.

3.2.3 Optimization

This algorithms uses direct optimization via the conjugate gradient method. This methods has two advantages, the first is the guarantee reduction of the cost function at each step; the second is that no Hessian needs to be computed or stored.

3.2.4 Solution Techniques

The cost function is of the form

$$\min_z J = \int\int L(u, u', u'', ...)dxdy \tag{3.5}$$

where u are the optimization variable, and L is a possible non-linear function of the optimization variables and its derivatives. The integral is taken over the domain of the variables. The solution to this problem can be found by solving the associated *Euler-Lagrange* equations(See a variational calculus book, such as [8]).

The Euler-Lagrange equations for a problem such as the one above are typically coupled non-linear equations. Such equations are usually very difficult to solve analytically but can sometimes be solved numerically by converting them into discrete equations. The conversion process involves substituting discrete approximations for any derivatives of the optimization variables. The optimization variables may have approximated by a discrete vector as well. The equations are then re-arranged to create iterative update of the form

$$u(k+1) = f(u(k), u(k-1), ...) \tag{3.6}$$

where $u(k)$ is the value of the optimization variables for the k-th iteration.

When u has many components and when the components are updated in sequence based on the best current estimate u, the resulting update scheme is called a *Gauss-Seidel* optimization. When all of the components of u are update simultaneously based on a previous estimate for u, the resulting scheme is called a *Gauss-Jordan* optimization. *Gauss-Seidel* optimization schemes have higher convergence rates and are more robust, and are best implemented on a serial computer. *Gauss-Jordan* schemes, while they have lower convergence rates and are not as robust, can be implemented on parallel computers.

Another way of solving the optimization problem posed is by using direct optimization techniques. In this case the cost function, instead of *Euler-Lagrange* equations are discretized. Any integrations are approximated by sums and any derivatives are approximated by differences. The resulting cost function is of the form

$$\min_u J = \sum_x \sum_y f(u) \tag{3.7}$$

where $f(u)$ is a discrete approximation of $L(u, u', u'', ...)$. We chose *conjugate gradient* optimization. The conjugate gradient scheme doesn't require the formation of of the problem Hessian[2], which for an optimization problem with N variable is a N-by-N matrix. The conjugate gradient scheme is important for vision problem since for a typical 256-by-256 image, the Shape-from-Shading problem would have 256^2 or 65536 optimization variables. The hessian for this problem would have 256^4 or over 4 billion elements.

3.2.5 Speedup Techniques

For vision problems there are two promising speed up techniques: the use of *hierarchical basis functions*, and *multi-grid methods.* Both try to speed up the optimization problems by increasing the information transfer spatially. The methods are based on the properties of many vision algorithms where an optimization variables within a grid of optimization variables may only be affected by its nearest neighbors. Due to the local connectness, many vision algorithms have diffusion like properties, the solution must diffuse throughout the grid. Schemes that transfer information over longer distances thus may speed up an algorithm.

Using hierarchical basis functions transform the optimization space as seen by the optimization algorithm but not as seen by the vision algorithm. Basically it is like change of basis. Figure 3-4 shows the presentation of of a 9-by-9 domain in hierarchical basis. In particular, note that the nodes of the hierarchical basis propagate information over a much larger range than the nodes in nodal basis. It shows linear interpolation between nodes, but any interpolation scheme can be used to build a hierarchical basis. The variables can be transformed to the nodal basis when

[2]A linear approximation to the second order properties of the solution space at a given point

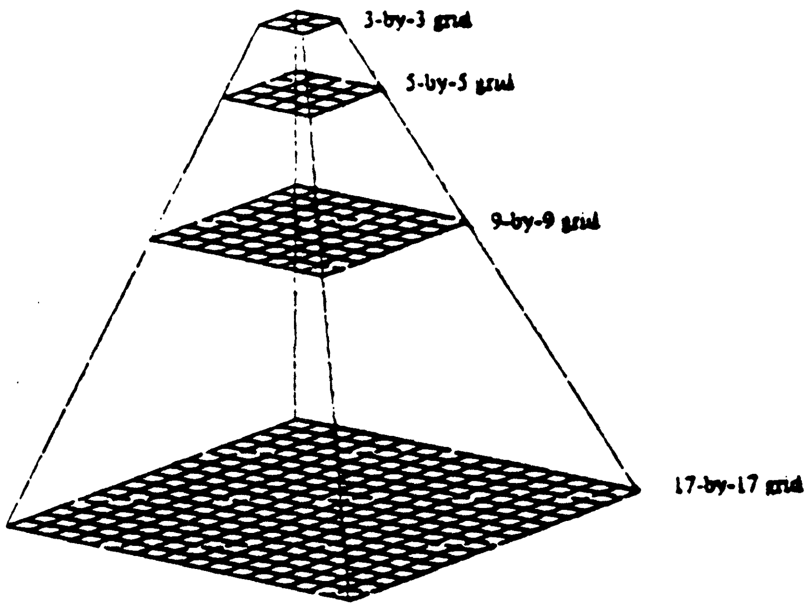


Figure 3-4: Hierarchical Basis Functions

computing the cost function or gradient. Unfortunately, all these transformation can introduce round-off errors may adversely affect sensitive algorithms.

The hierarchical basis functions have the most effect on the convergence and are the easiest to implement when the grid size is $2^n + 1$, where n is any positive integer. for such a grid it is possible to use $n + 1$ hierarchical basis levels. Using hierarchical basis functions increases the communication between nodes in the image array. so to speed up the diffusion process considerably.

The multigrid methods seek to propagate information over a large range by solving a series of problems of different size. Usually the original problem is formulated on grids that decrease in size by a factor of two when going from one layer to the next. The solutions on one layer are related to solutions on the layers above and below via interpolation or prolongation. The solution are kept consistent with each other via both intra-layer and inter-layer process. (See [23] and [2]).

Multigrid methods have the potential to be much faster than the hierarchical basis functions since most of the computation is done on the smaller layers. Multigrid methods are well suited to linear problems but may not work for non-linear problems. Since

$$\sum f(u) \neq f(\sum u) \tag{3.8}$$

for non-linear function f, and the multigrid methods rely on equality of the previous equation to constrain the solutions on the smaller grids so that they don't bias the solution on the larger grid.

One type of multi-grid method that can be used for non-linear problems is the *coarse-to-fine* method. In this method, the problem is solved on coarse layers first and the solution to each layer provides the initial condition to the next finer layer below. This method is significantly faster that just optimizing on the finest grid but does not produce as much convergence speed up as the full multigrid method.

Like the hierarchical methods, the multigrid methods work best when the grid size is $2^n + 1$. However, since the multigrid methods define a series of problems rather than just choosing a new set of basis functions, the multigrid methods can be implemented easily for all grid size. There is some evidence, that the grid size reduction should be near 2 for best convergence rate (See [23]).

3.2.6 Discussion

The question of existence and uniqueness come up when working with optimization algorithms. For the cost function presented earlier, it is clear that a solution exist. The solution are bounded from below by zero. That is, the best possible value for the cost function is zero and can be achieved only when the estimated surface images and the actual images match exactly and when the regularization term is set to zero.

The uniqueness of a solution depends a great deal on the surface to be estimated. In general, both global and local minima will exist. The optimization techniques discussed above only guarantee convergence to a local minima. The global minimum may only be achieved if the initial conditions for the optimization algorithm are close to the true solution.

Chapter 4

Synthetic Image Test Results

It is crucial to test an new algorithm's performance, to understand its strength and weakness before applying it to the real images. To test, it must be possible to compare the estimated surface with the actual surface. The way to do it is to create synthetic images from a known surface topology, estimate the surface with *z-only* Depth from Shading and Stereo algorithm, compare the estimated surface with actual surface. Only after that, we can be confident in the correctness of the new algorithm.

4.1 Synthetic Images

Four synthetic images are used with various difficulties to the Depth from Shading and Stereo, from three typical topology.

- **Easy Crater Images:** The first pair of images is based on a crater on a flat plane. The light sources are oblique, this makes it relative easy for DFSS to estimate.

- **Hard Crater Images:** The second pair of images is based on a crater on a flat plane. The light sources are almost behind the camera, this makes it difficult for DFSS to estimate.

	b	f	z_0	$\Delta z/z_0$	$p_s^{(1)}$	$q_s^{(1)}$	$p_s^{(2)}$	$q_s^{(2)}$
Easy Crater	500	-2750	-997	0.0038	0.2	-0.5	-0.3	0.1
Hard Crater	500	-2750	-997	0.0038	0.1	0.1	-0.1	0.1
Hills	500	-12222	1000	0.0011	1.0	1.0	0.3	0.1
Mountain	100	-2292	-996	0.0153	0.5	0.5	-0.5	0.0

Table 4.1: Camera Geometry

- **Hill Images:** The third is based on a fractally-generated set of rolling hills. The light sources are oblique.

- **Mountain Images:** The third is based on a fractally-generated set of mountain terrain. The light sources are oblique and the baseline is smaller. This set of images poses a challenge and it most related to planet images.

The calibration parameters for the test images are summarized in Table 4.1. The table lists value for the baseline distance b, camera focal length f, nominal depth z_0 and light source vector $p_s^{(1)}$, $q_s^{(1)}$, $p_s^{(2)}$, $q_s^{(2)}$. Notice the focal length and nominal depth are negative so that it is consistent with a right hand system in perspective projection. Baseline b, depth z and light source components are in units of miles. The camera focal length and pixel spacing are based on camera units, millimeters.

All the test images are generate noise-free to test the best performance of the *z-only* DFSS algorithm. All the results are presented with history of smoothness weighting term λ, history of cost function, history of estimated error, and the mesh plot of depth z.

4.2 Algorithm Performance

In this section, I will show the performance of *z-only* of the above four set of images. All images are 65-by-65 pixel and using 6 levels of hierarchical basis, which is the largest number of levels that can be used.

4.2.1 Easy Crater Images

This set are based on a crater on a plat surface. It is very simple, and thus serves a good test bed. The very different light source positions give very different shadings to each image. As shown in Figure 4-1 The very strong shading information makes it easy for DFSS to estimate. Contour plots of the reflectance map clearly shows the effect of lighting. The larger separation of reflectance contours gradient space makes it easy to constrain the possible set of feasible gradient directions from brightness. Apparent *z-only* DFSS algorithm correctly estimated the easy crater surface. It is interesting to notice the small anomaly in the lower left corner of the surface. With the lighting condition, this anomaly has little effect on the estimated surface and the programs' convergence. *z-only* DFSS algorithm has very good performance, the correct surface topology is obtained in less than 150 iterations. The DFSS results is shown in Figure 4-4.

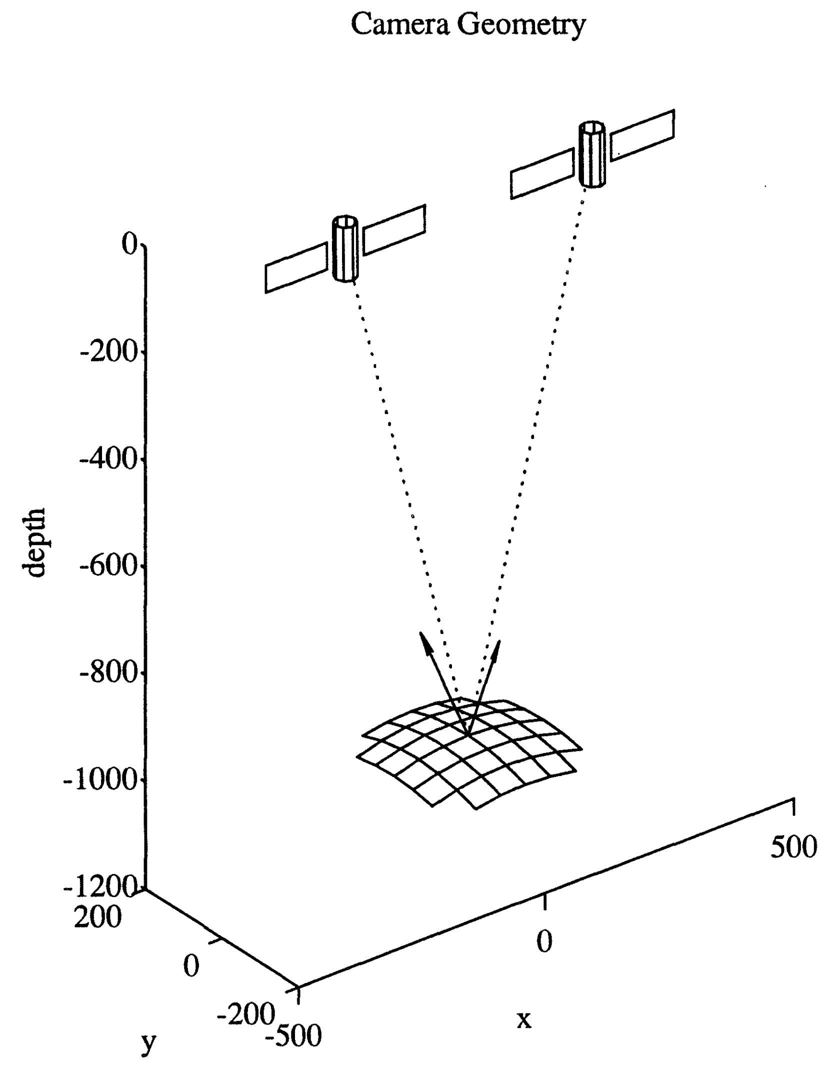
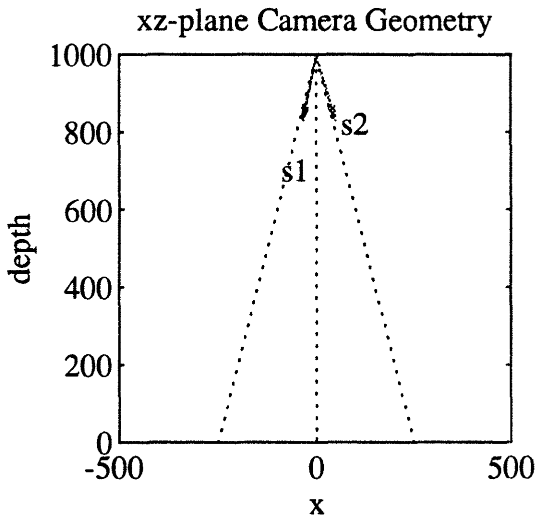


Figure 4-1: Easy Crater Synthetic Images Camera Geometry. Graph shows the camera position and direction(dotted lines) and light source direction(solid lines)



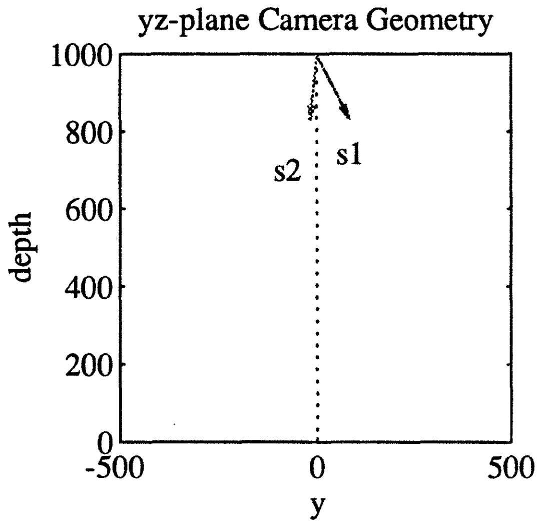
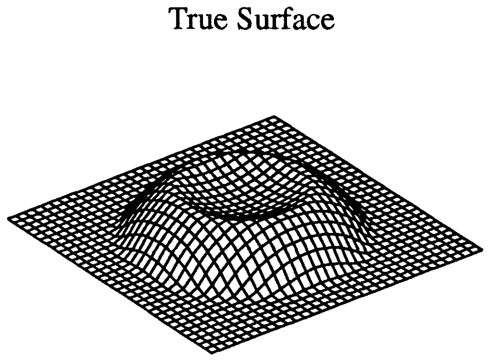
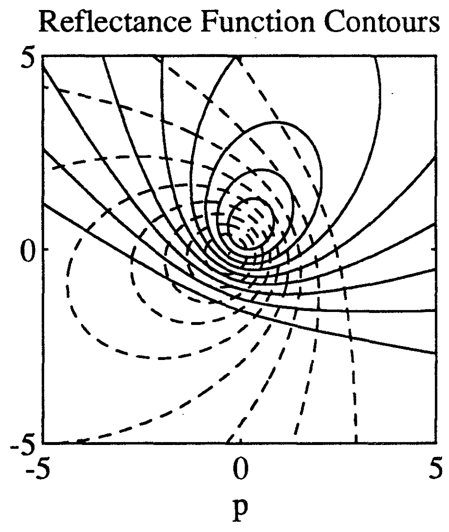


Figure 4-2: Easy Crater Synthetic Images Camera Geometry projected in xz and yz plane





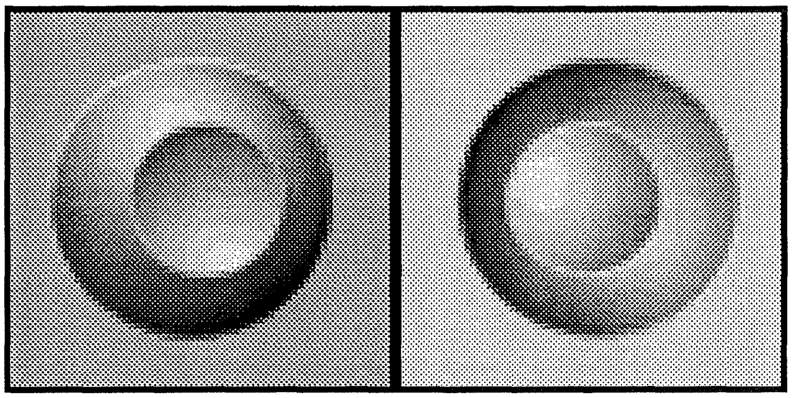
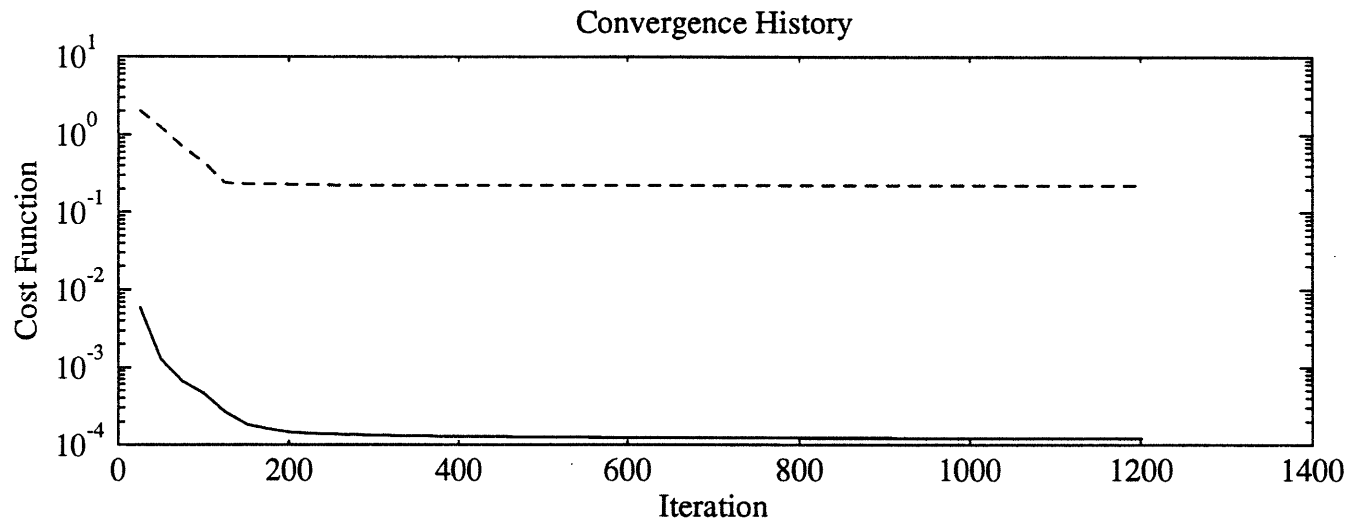


Figure 4-3: Easy Crater Synthetic Images



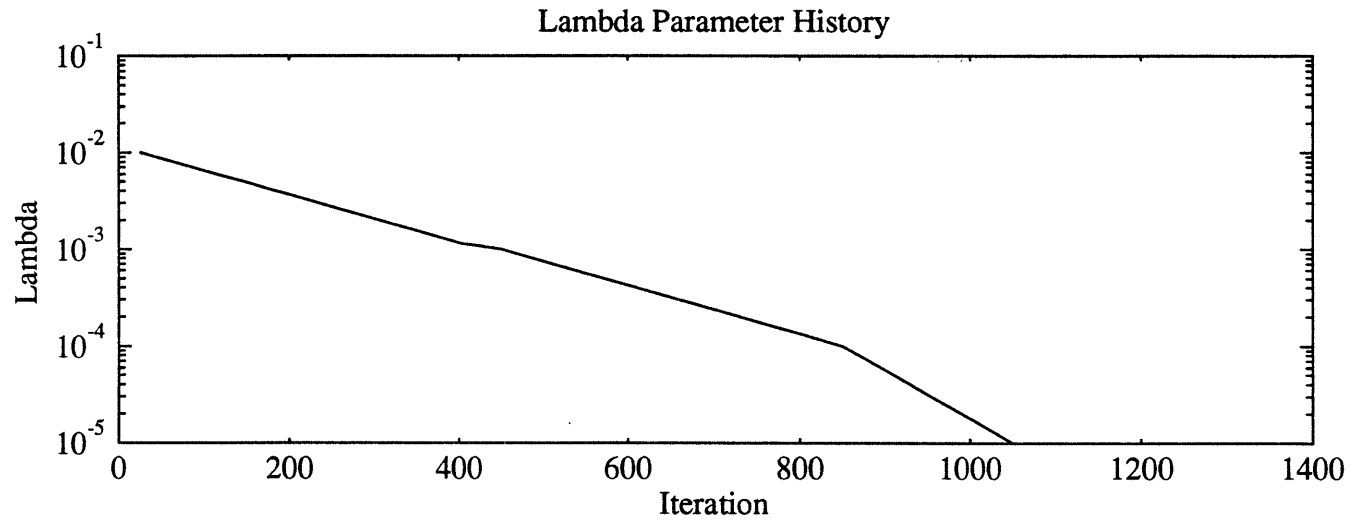
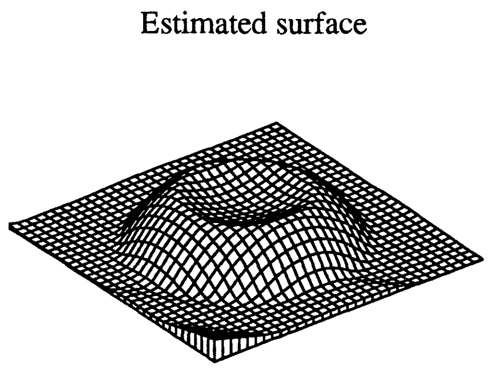


Figure 4-4: *z-only* DFSS iteration history on Easy Crater Images. Graph shows the *z-only* DFSS algorithm performance on the easy crater image pair. The upper graph shows the cost function(solid line) and RMS error(dashed line) of the estimated surface. The lower graph shows history of the smoothness weighting term λ



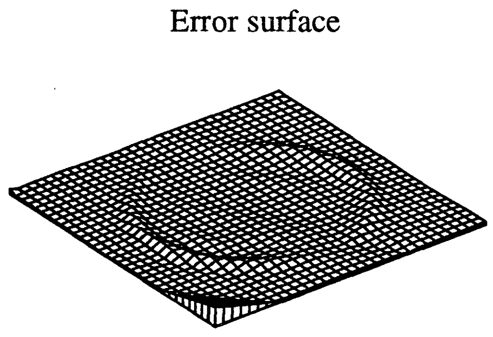


Figure 4-5: *z-only* DFSS Estimated Surface and Error on Easy Crater Images at the end of iterations

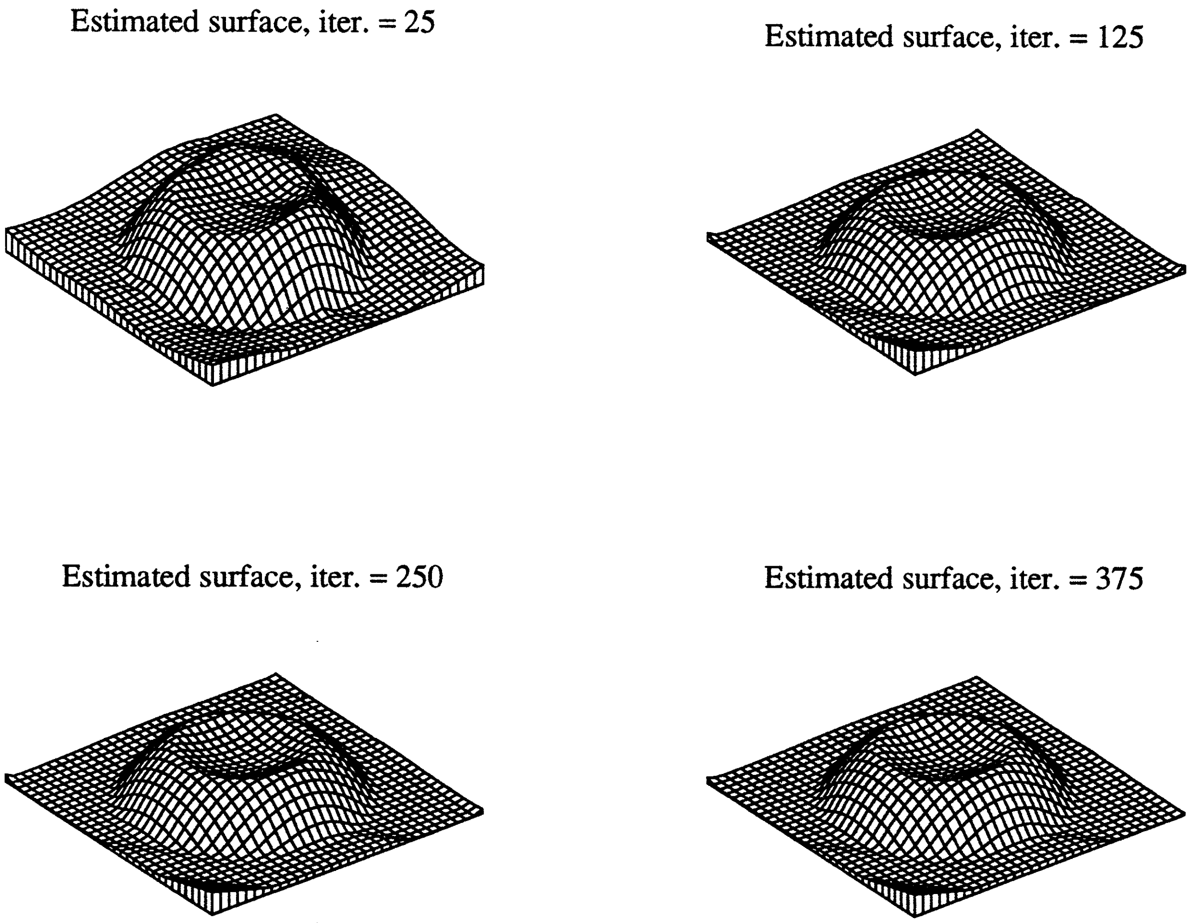

Estimated surface, iter. = 25
Estimated surface, iter. = 125
Estimated surface, iter. = 250
Estimated surface, iter. = 375

Figure 4-6: *z-only* DFSS estimation at various iteration steps on Easy Crater Images

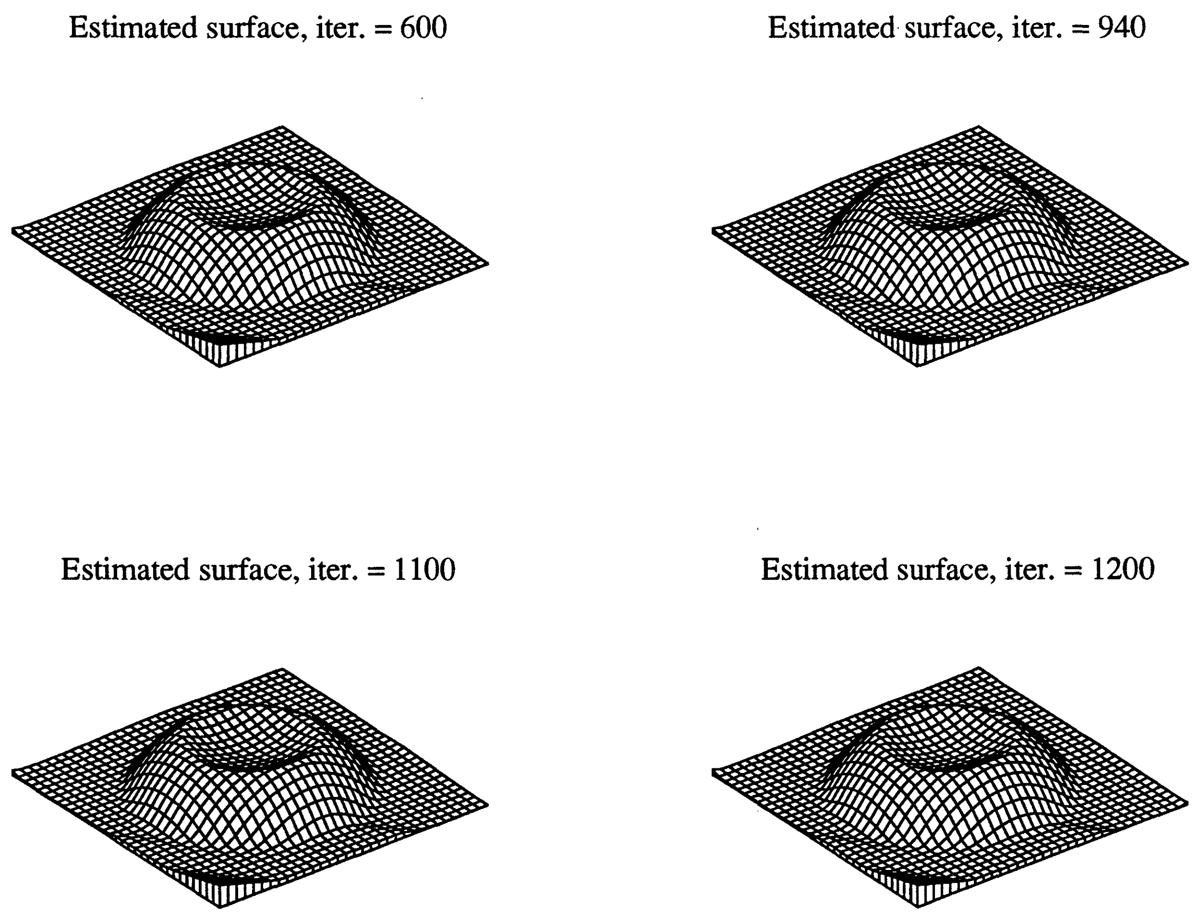

Estimated surface, iter. = 600
Estimated surface, iter. = 940
Estimated surface, iter. = 1100
Estimated surface, iter. = 1200

Figure 4-7: *z-only* DFSS estimation at various iteration steps on Easy Crater Images

4.2.2 Hard Crater Images

In this set of images, the baseline distance between the cameras is about half the distance to the surface and the light source positions for the two images are nearly the same and almost directly behind the cameras. As shown in Figure 4-8, from the light contour plot and the true images, the two images are very bright and look very much the same except for slight difference in the shading.

This images represent the worst case for DFSS algorithm since the shading information is weak and the range of brightness is small. However, they do have strong stereo correspondence information which is unfortunately not heavily utilized by *z-only* DFSS algorithm.

Figure 4-11 to Figure 4-14, shows the results of applying the *z-only* DFSS algorithm to this test case. It shows the estimated surface resembles the true image in figure 4-8, even though they don't match. Clearly the *z-only* DFSS algorithm gets stuck in a local minimum. The problem stems from that the algorithms incorrectly interpreted the surface to be concave when it is actually convex. Even though stereo information is presented and can be used to correctly determine the orientation of the surface, the *z-only* DFSS algorithm is biased towards shading information. The surface orientation ambiguity is also a result of having both light sources directly behind the cameras.

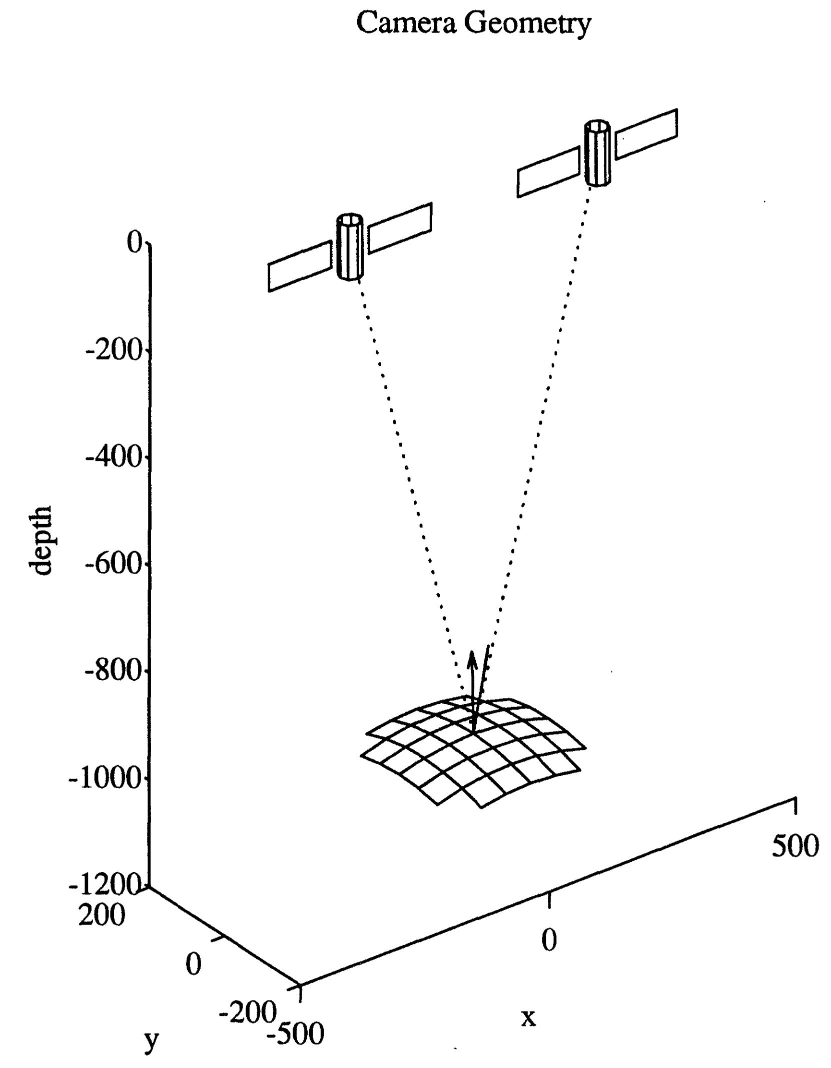
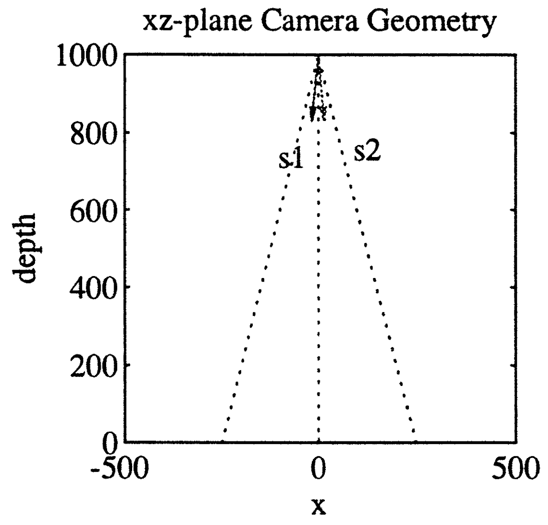


Figure 4-8: Hard Crater Synthetic Images Camera Geometry. Graph shows the camera position and direction(dotted lines) and light source direction(solid lines).



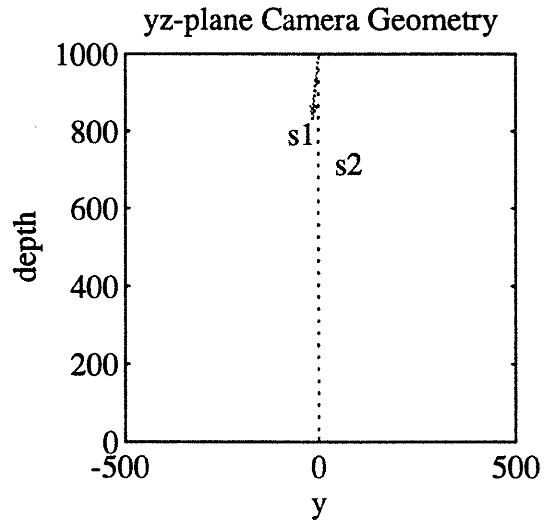
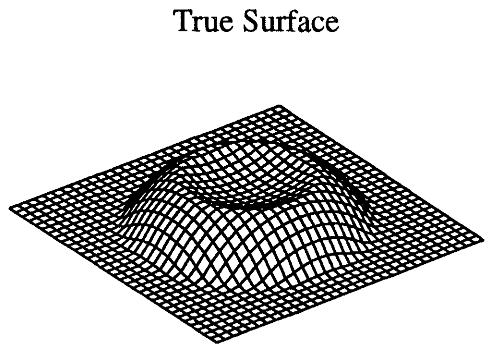
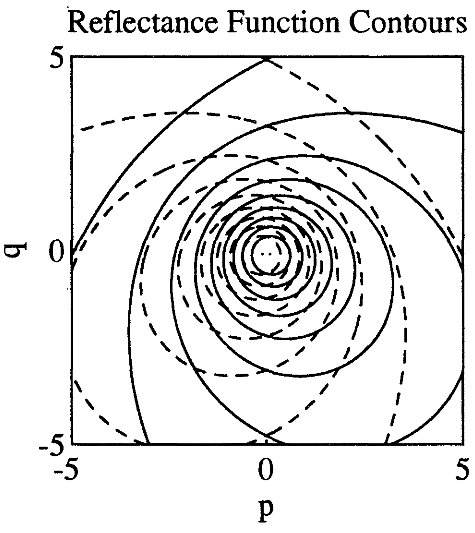


Figure 4-9: Hard Crater Synthetic Images Camera Geometry projected in xz and yz plane





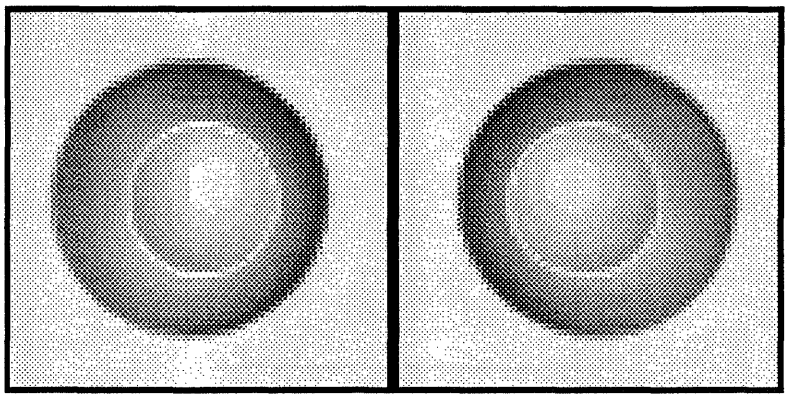


Figure 4-10: Hard Crater Synthetic Images

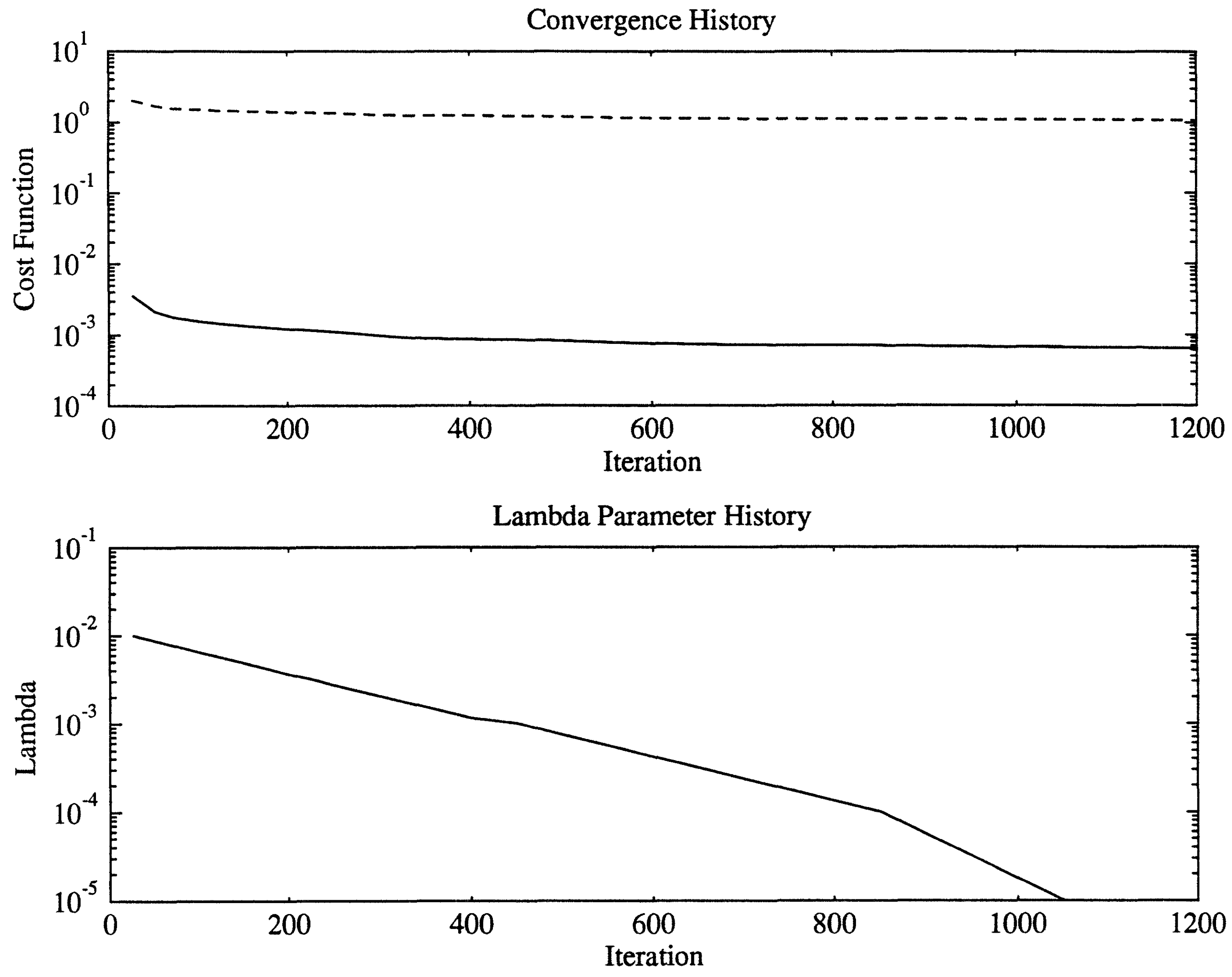


Figure 4-11: *z-only* DFSS iteration history on Hard Crater Images. Graph shows the *z-only* DFSS algorithm performance on the hard crater image pair. The upper graph shows the cost function(solid line) and RMS error(dashed line) of the estimated surface. The lower graph shows history of the smoothness weighting term λ

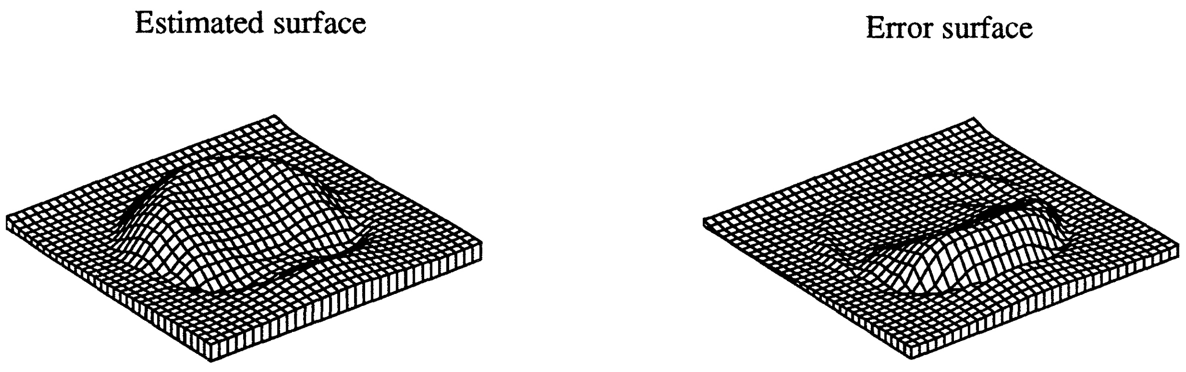


Figure 4-12: *z-only* DFSS Estimated Surface and Error on Hard Crater Images at the end of iterations

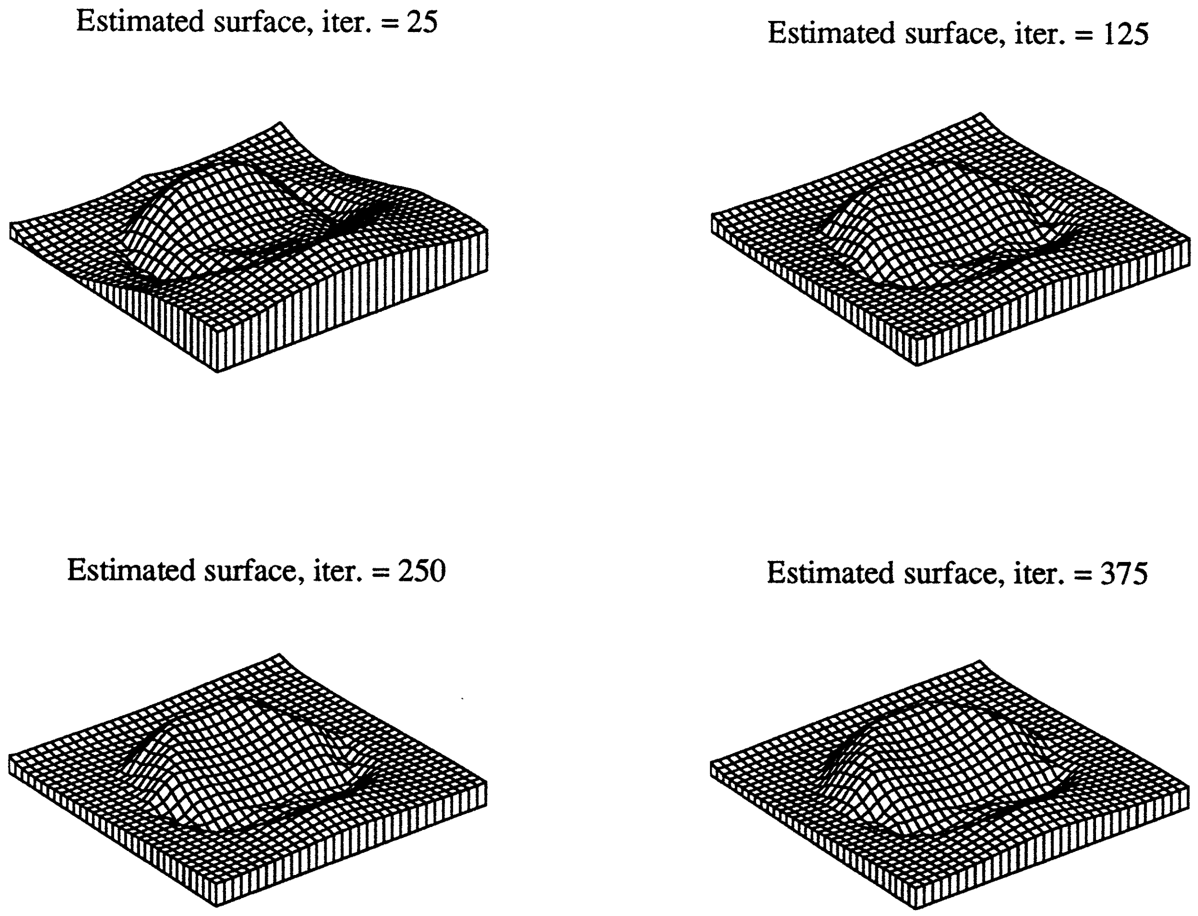

Estimated surface, iter. = 25
Estimated surface, iter. = 125
Estimated surface, iter. = 250
Estimated surface, iter. = 375

Figure 4-13: *z-only* DFSS estimation at various iteration steps on Hard Crater Images

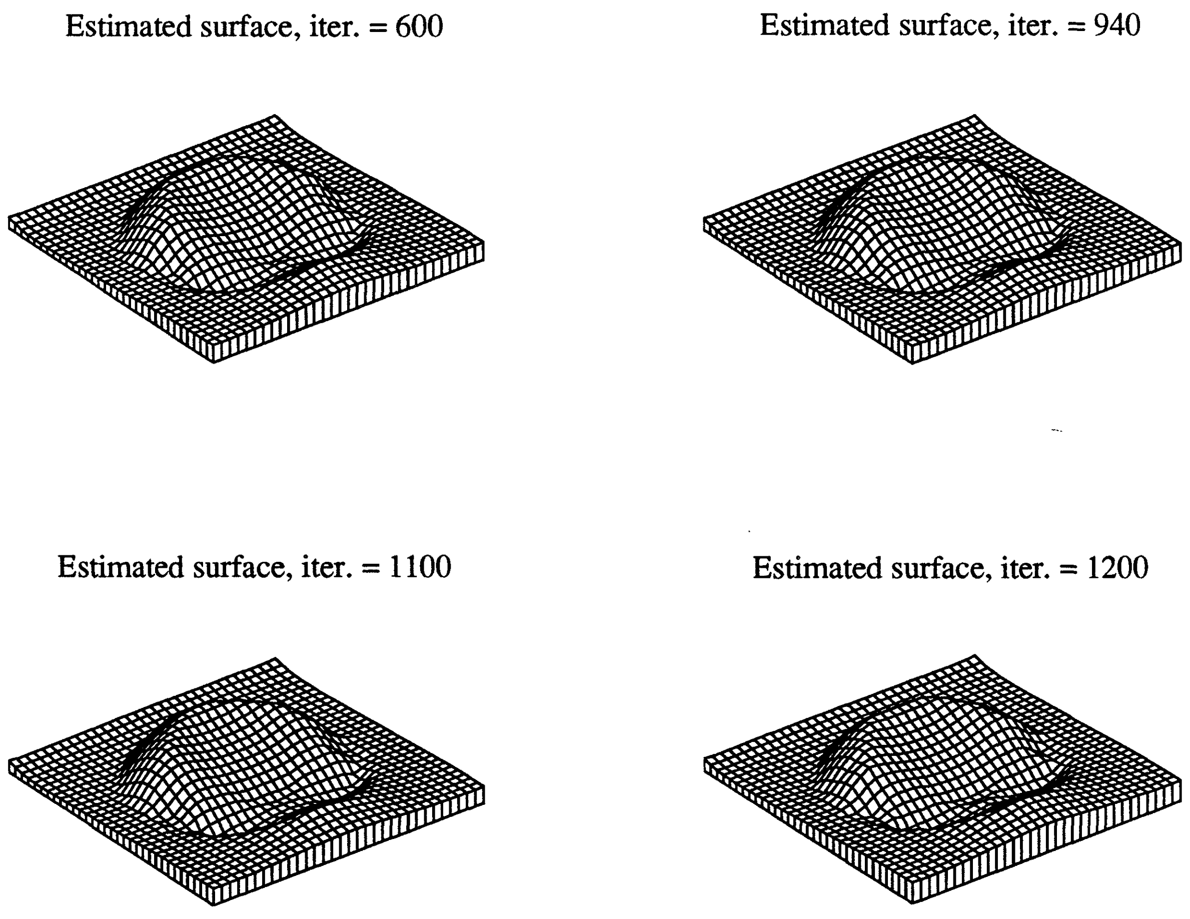


Figure 4-14: *z-only* DFSS estimation at various iteration steps on Hard Crater Images

4.2.3 Hill Images

The third set of images is of an undulating surface similar to erode hills and was generated using a fractal technique. Figure 4-15 shows the camera geometry, true surface, reflectance contours. This set of images is more representative of the type of terrain the DFSS algorithm are likely to encounter. As shown in Figure 4-15, the left cameras is directly over the surface and the light camera views the surface obliquely. The light sources are separated, as in the ease crater case. This results in the images with strong shading information. The DFSS algorithm performed rather well in this case, correctly interpreted the surface.

The Figure 4-18 shows the results of applying the DFSS algorithm to the hill images. *z-only* DFSS algorithm performs well, correctly estimated the complicated topology. Due to the complexity, the images requires more iterations to obtain a satisfactory estimate of the surface than the easy crater case.

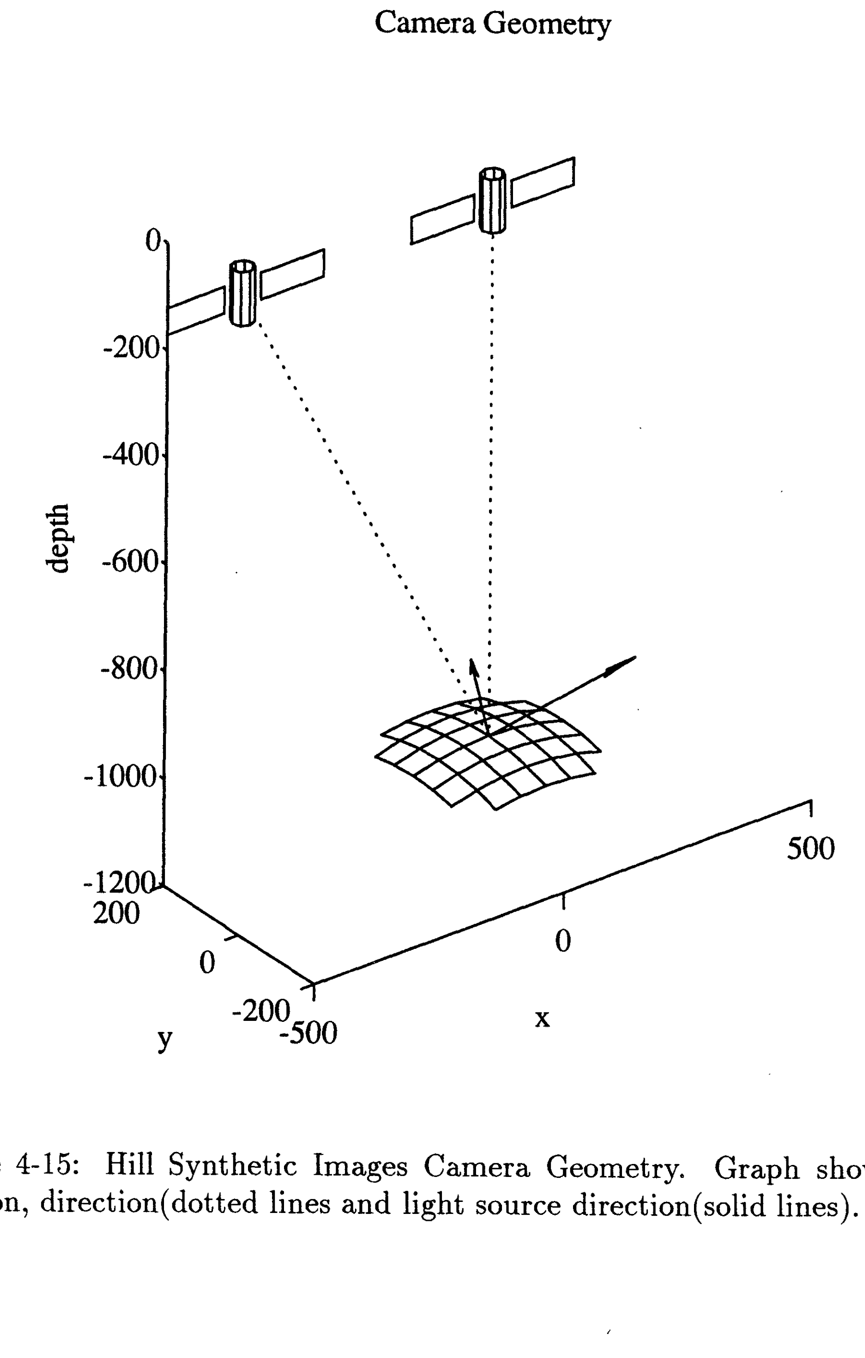


Figure 4-15: Hill Synthetic Images Camera Geometry. Graph shows the camera position, direction(dotted lines and light source direction(solid lines).

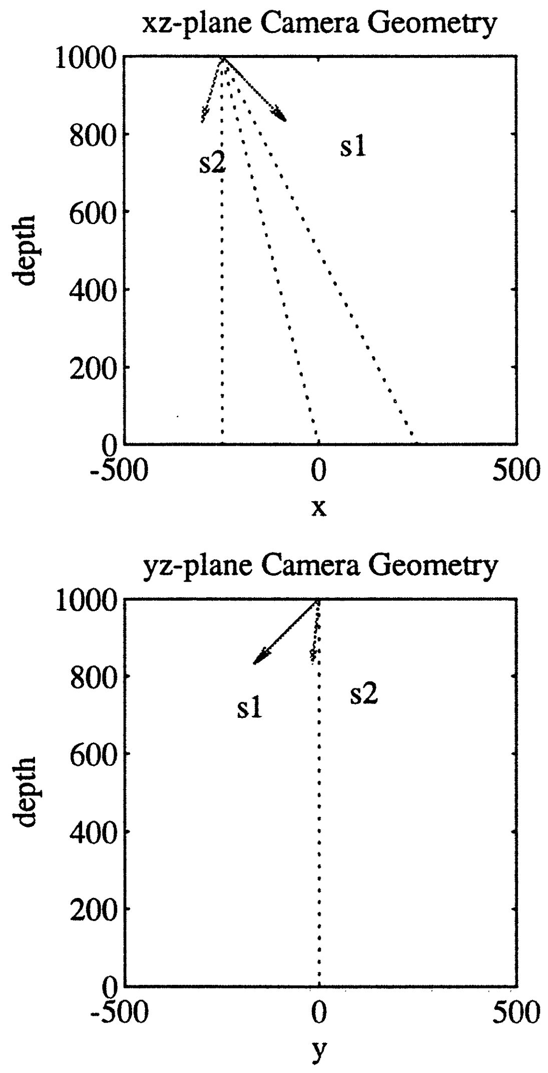
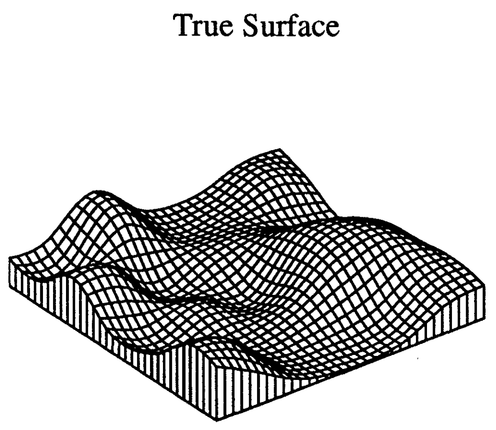
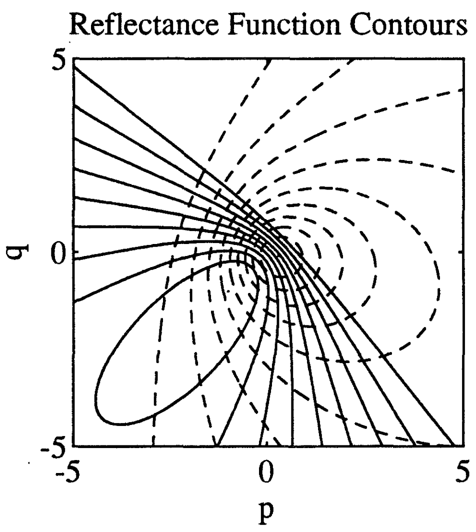


Figure 4-16: Hill Synthetic Images Camera Geometry projected in xz and yz plane





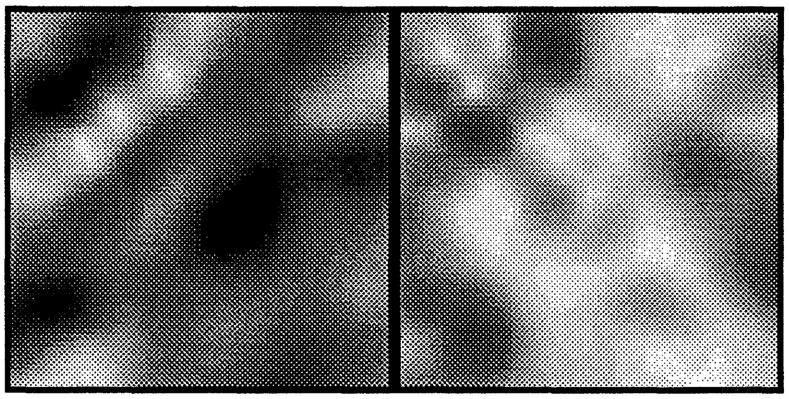


Figure 4-17: Hill Synthetic Images

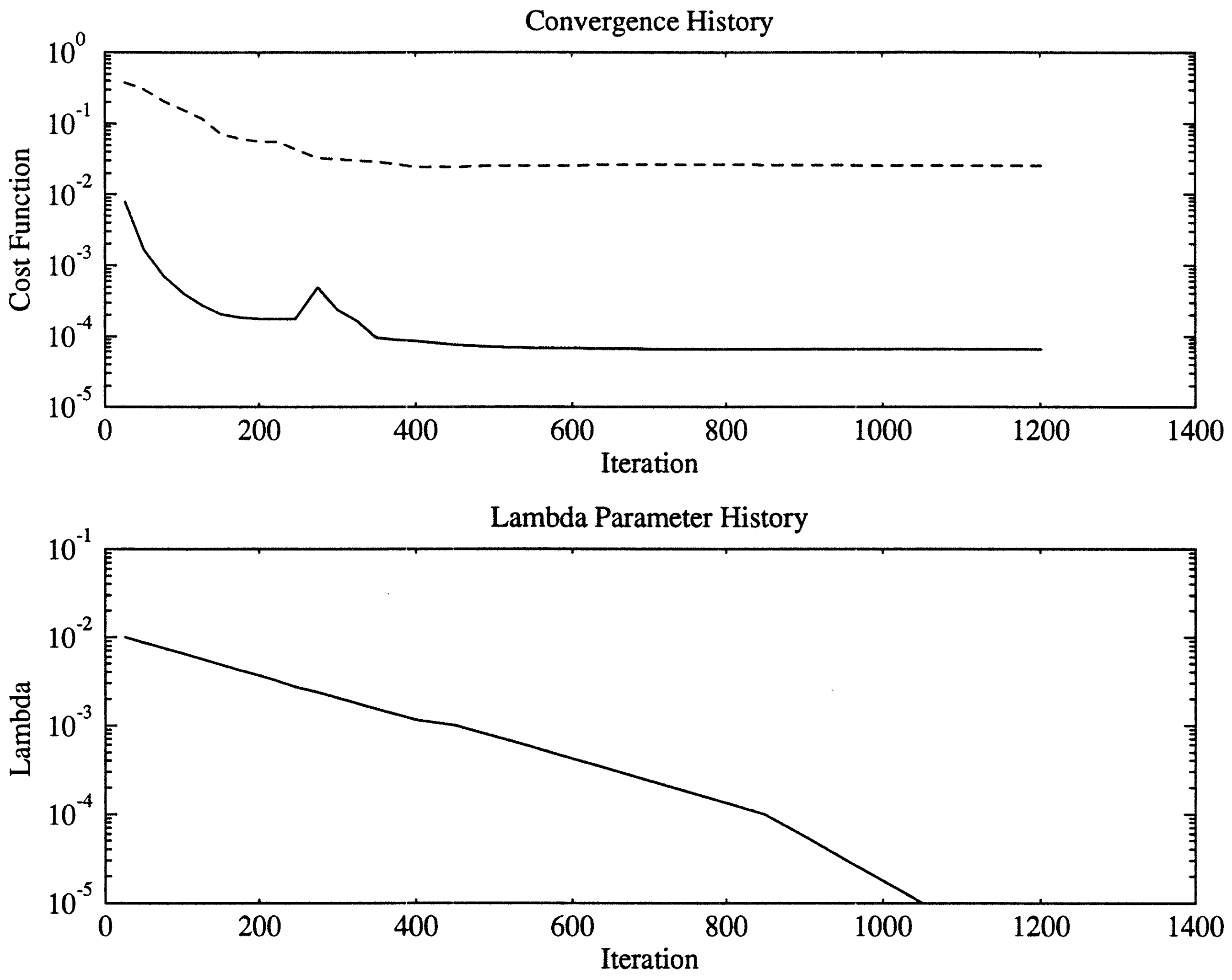


Figure 4-18: *z-only* DFSS iteration history on Hill Images. Graph shows the *z-only* DFSS algorithm performance on the hill image pair. The upper graph shows the cost function(solid line) and RMS error(dashed line) of the estimated surface. The lower graph shows history of the smoothness weighting term λ

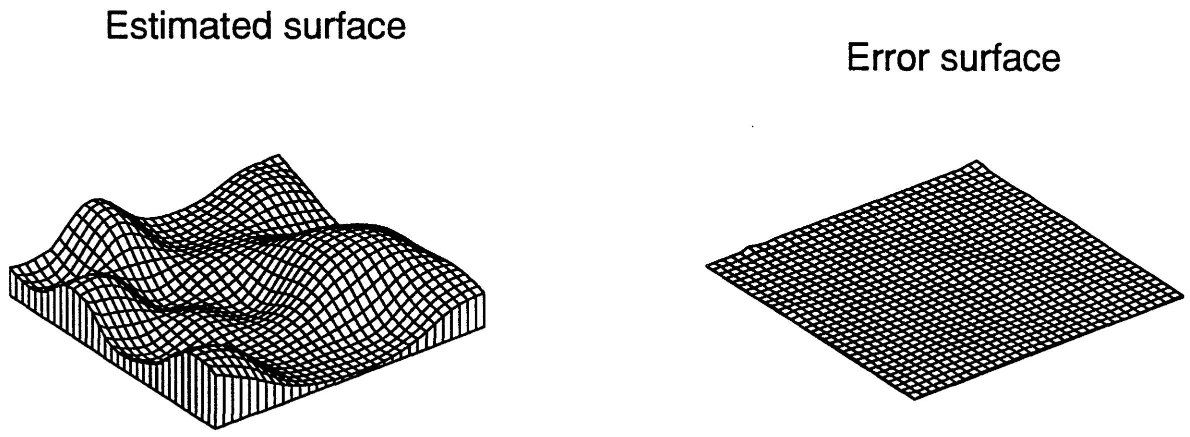


Figure 4-19: *z-only* DFSS Estimated Surface and Error on Hill Images at the end of iterations

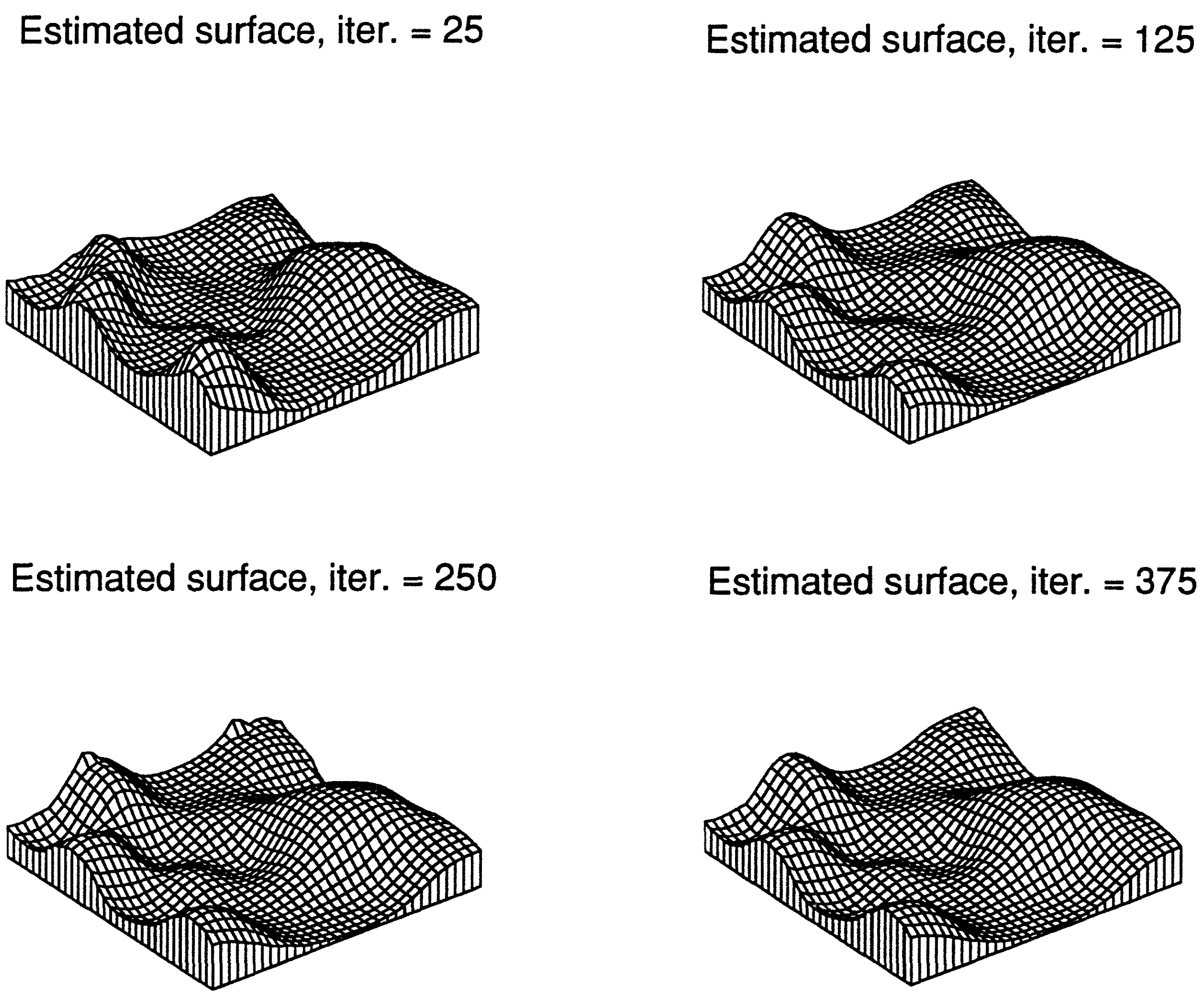


Figure 4-20: *z-only* DFSS estimation at various iteration steps on Hill Images

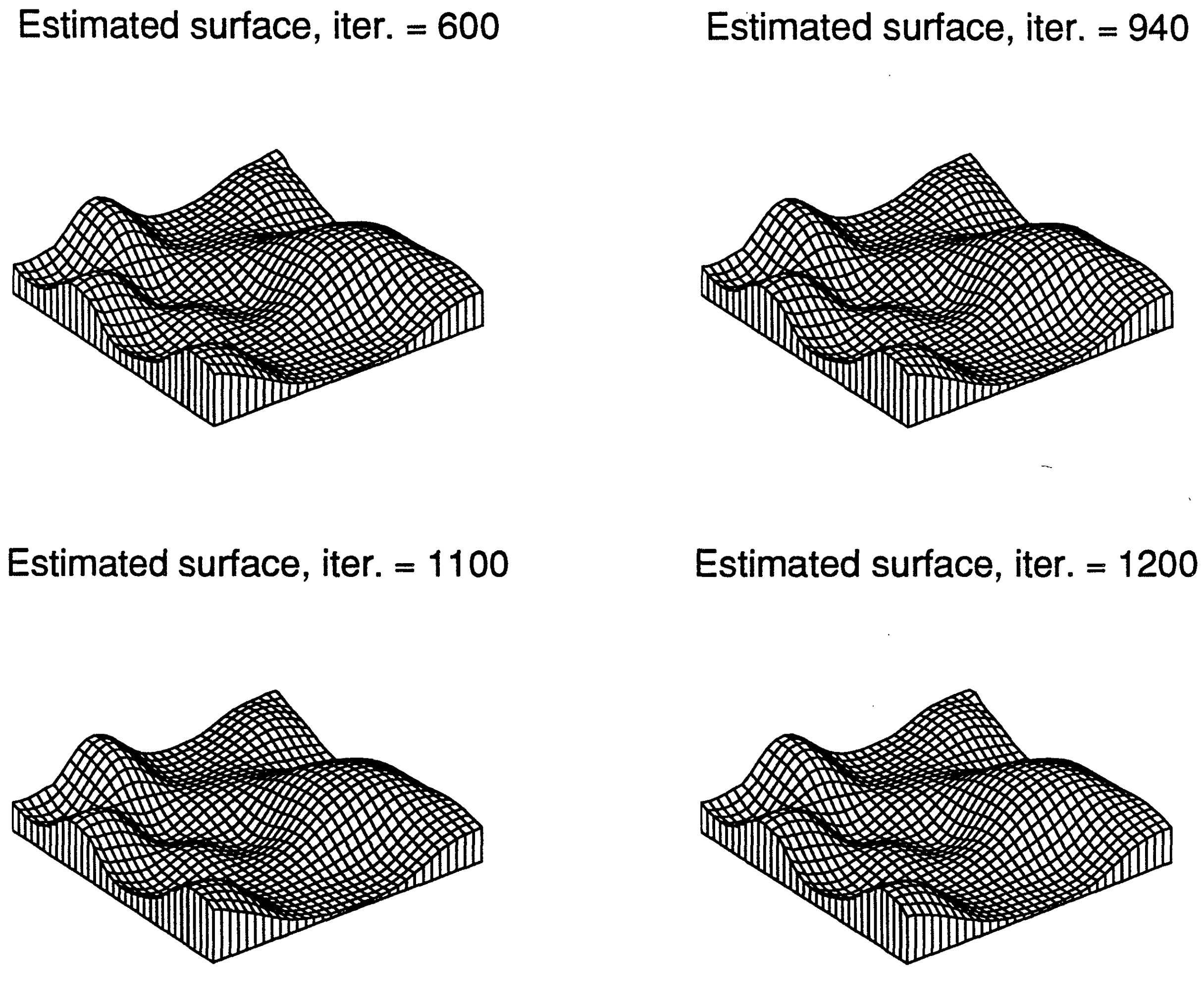

Estimated surface, iter. = 600
Estimated surface, iter. = 940
Estimated surface, iter. = 1100
Estimated surface, iter. = 1200

Figure 4-21: *z-only* DFSS estimation at various iteration steps on Hill Images

4.2.4 Mountain Images

The fourth set of images is of a highly mountainous surface and was created to show the performance on steep terrain. The steep terrain requires that the baseline distance from the cameras to be small so that all the surface points are visible from both cameras. Thus it is a good example to show the merit of the algorithm when the stereo baseline is small. As shown in Figure 4-22, The light source position are widely separated and generate deep shadows on this steep terrain. The reflectance maps are flat within a shadow so no helpful gradient is available to the algorithm. In addition, knowledge that a particular pixel is in shadow only constrains the set of possible gradient directions to a sub-pane of gradient space. Thus, within a shadow region, much more influence is given to the brightness values from the other image. This results in a slower convergence.

The Figure 4-25 shows the results of applying DFSS algorithm. The *z-only* DFSS algorithm performed reasonably well, correctly interpreted the surface. Due to the complexity of the surface. the algorithm requires many more iterations to achieve a satisfactory solution. Never the less, the algorithm correctly estimates the surface. It proves the algorithm is robust and can handle small base line difference.

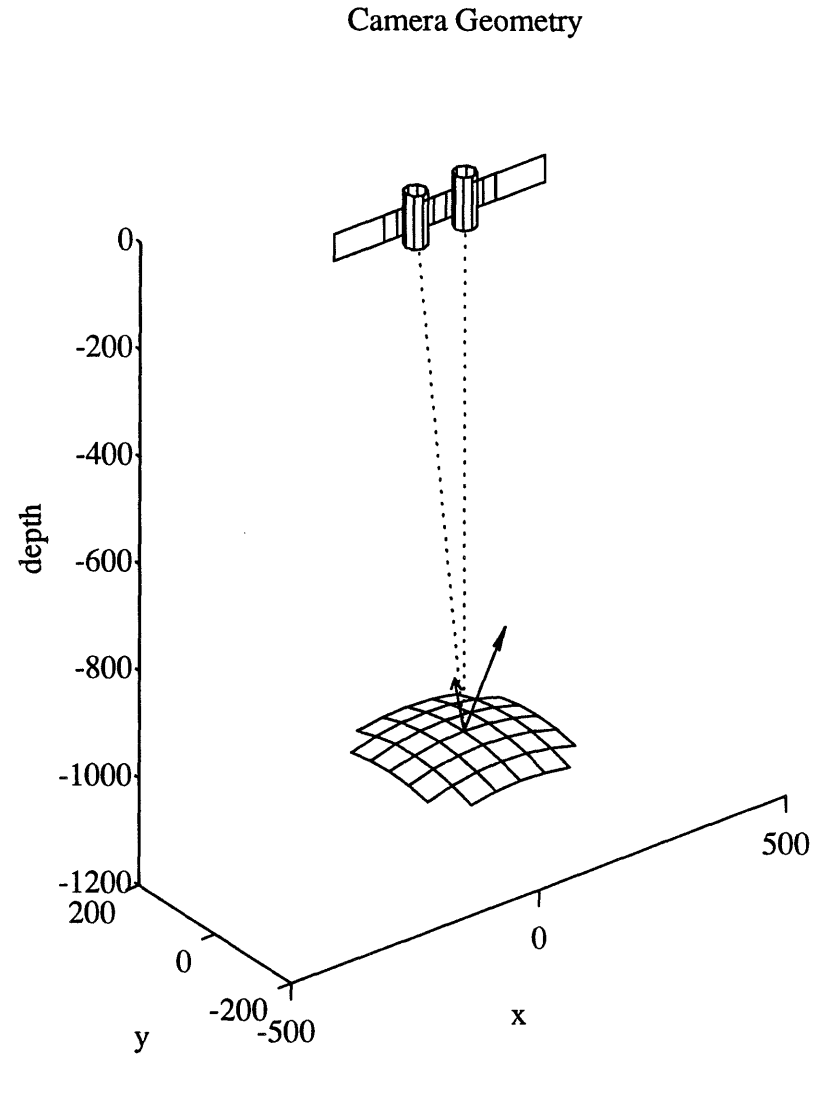
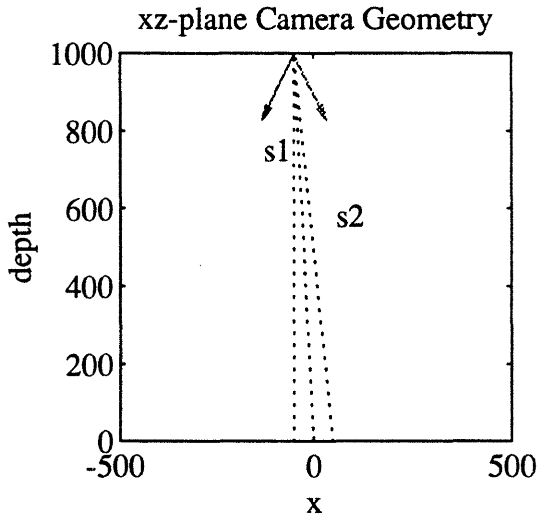


Figure 4-22: Mountain Synthetic Images Camera Geometry. Graph shows the camera position and direction(dotted lines) and light source direction(solid lines).



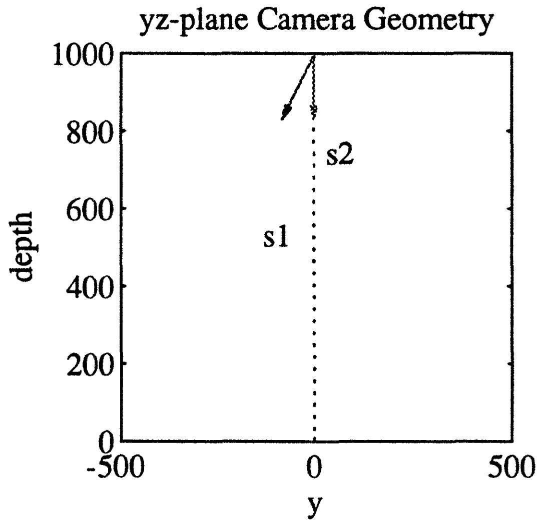
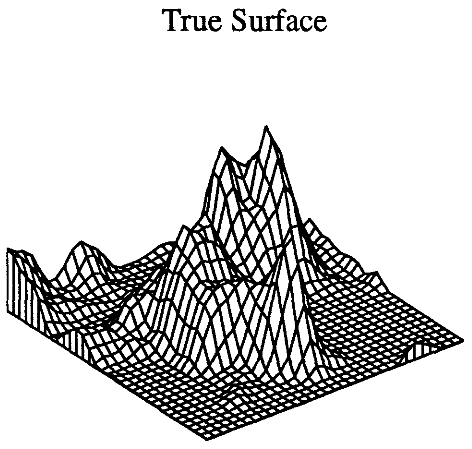
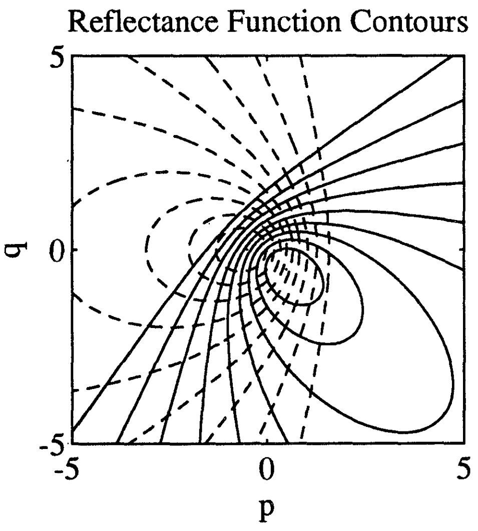


Figure 4-23: Mountain Synthetic Images Camera Geometry projected in xz and yz plane





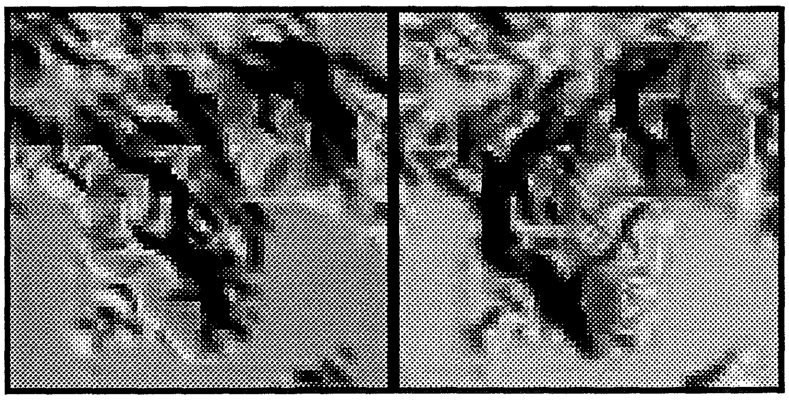


Figure 4-24: Mountain Synthetic Images

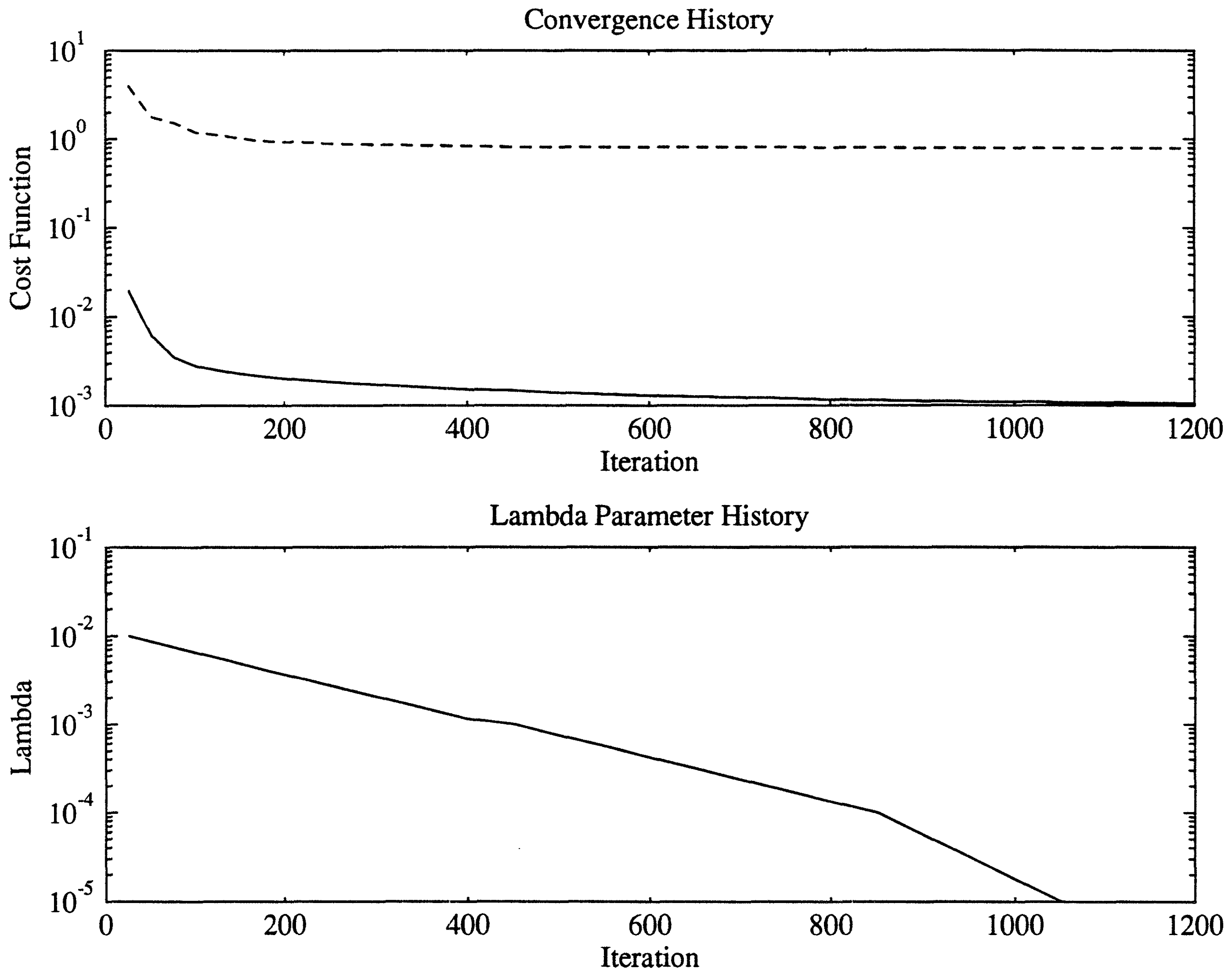


Figure 4-25: *z-only* DFSS iteration history on Mountain Images. Graph shows the *z-only* DFSS algorithm performance on the mountain image pair. The upper graph shows the cost function(solid line) and RMS error(dashed line) of the estimated surface. The lower graph shows history of the constraint λ

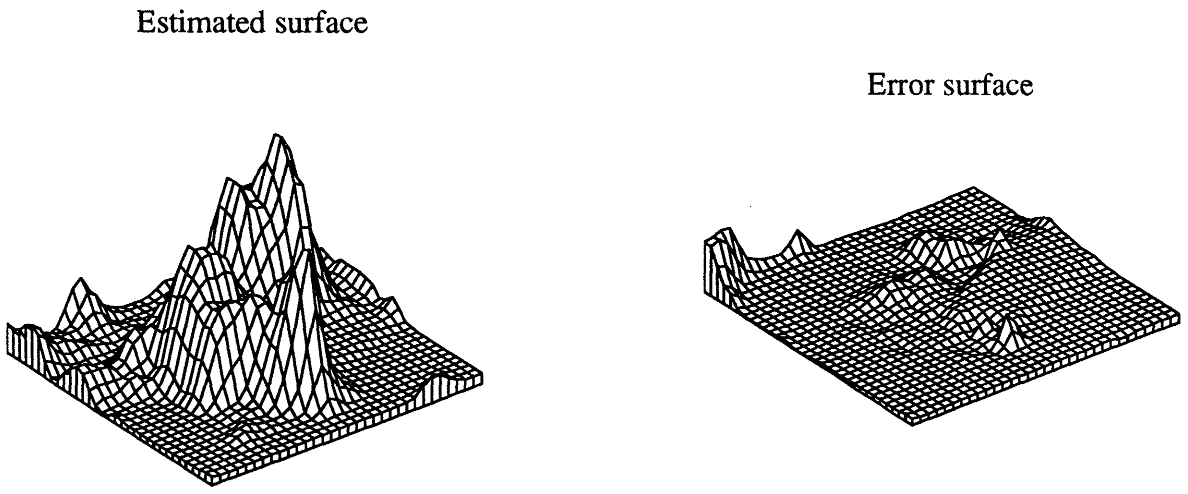


Figure 4-26: DFSS Estimated Surface and Error on Mountain Images at the end of iterations

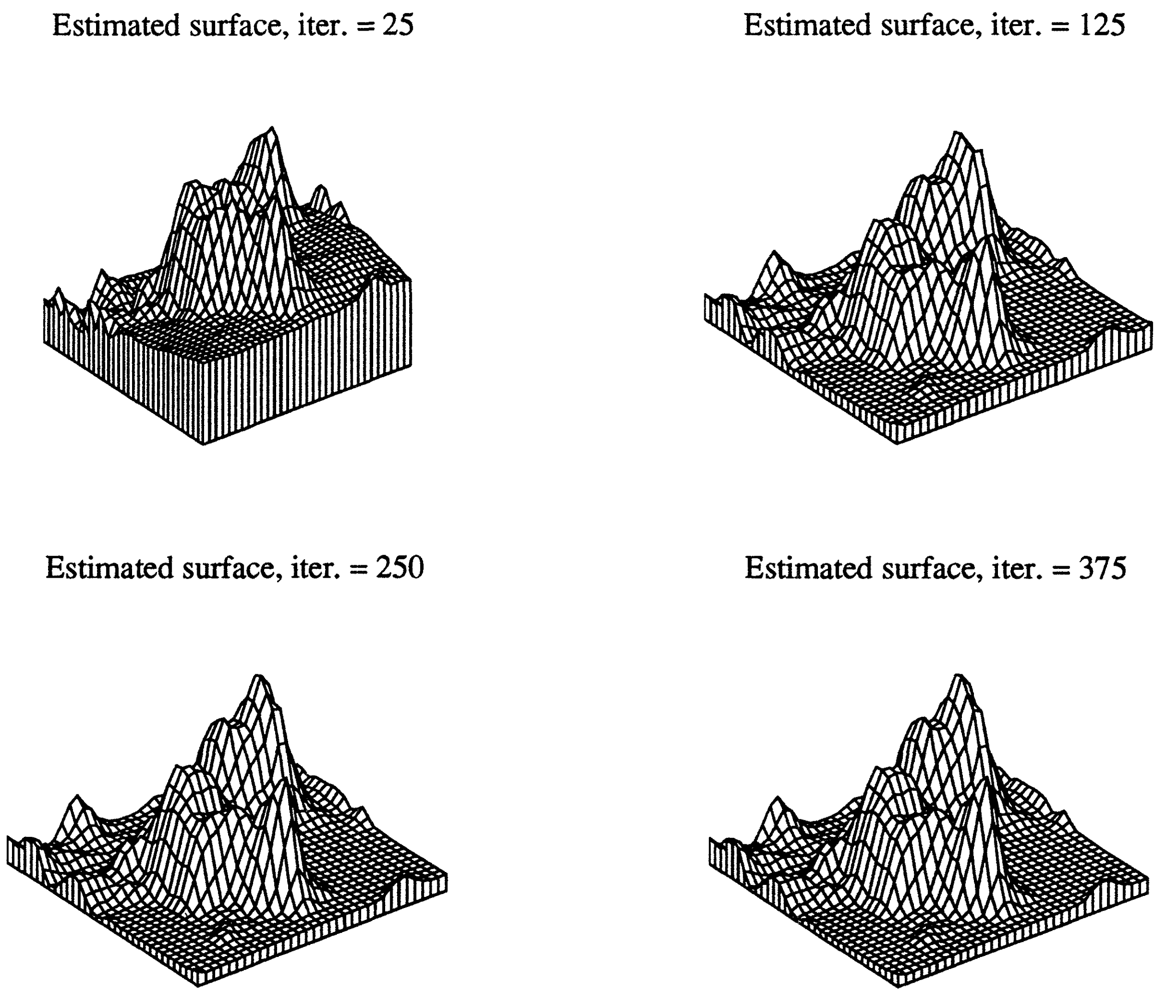

Estimated surface, iter. = 25
Estimated surface, iter. = 125
Estimated surface, iter. = 250
Estimated surface, iter. = 375

Figure 4-27: DFSS estimation at various iteration steps on Mountain Images

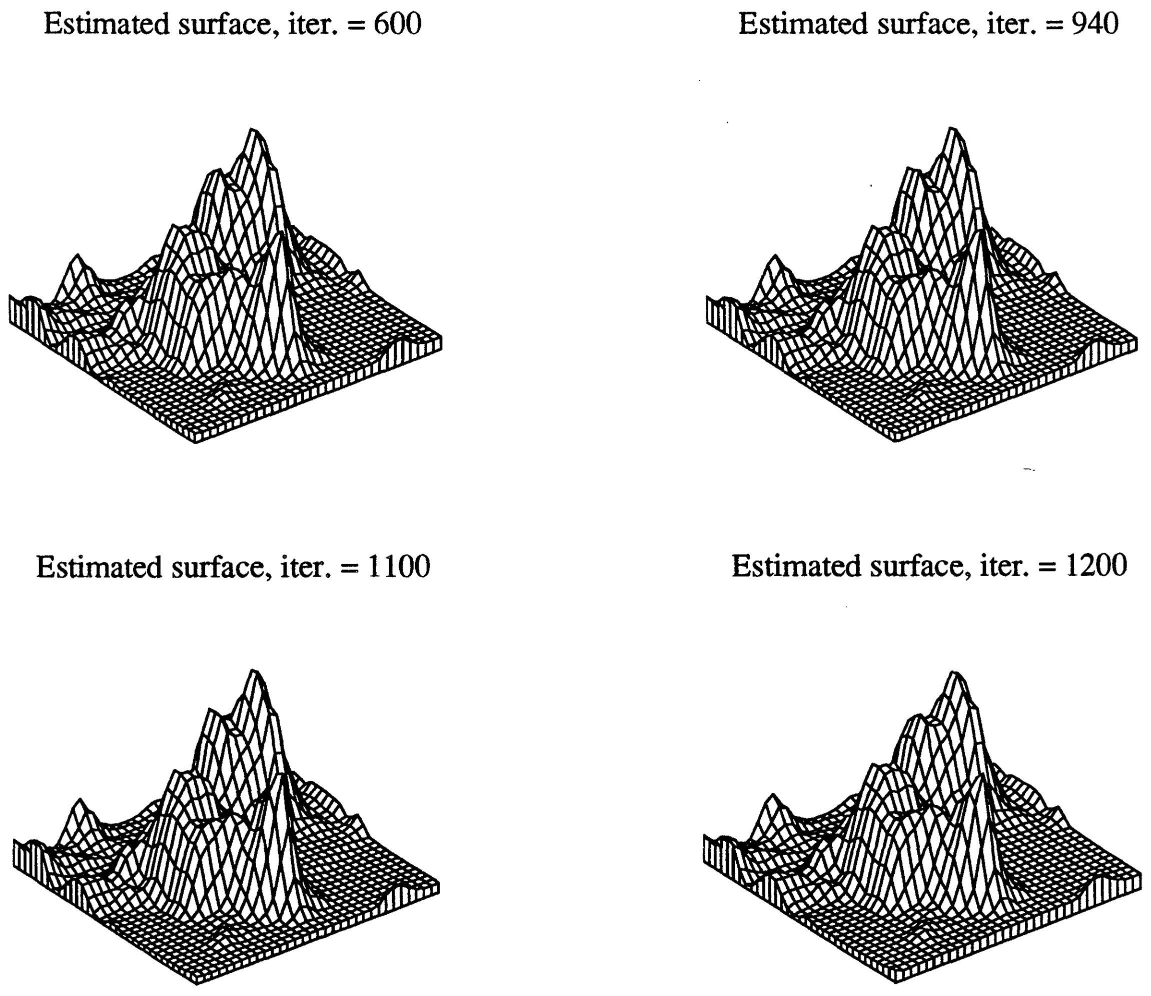


Figure 4-28: DFSS estimation at various iteration steps on Mountain Images

	Rel. Error	Abs. Error	Iterations
Easy Crater	0.172	0.173	1200
Hard Crater	1.072	1.073	1200
Hills	0.018	0.025	1200
Mountain	0.718	0.780	1200

Table 4.2: Performance Comparison

4.3 Summary of the Results

A summary of the results of DFSS algorithms running on easy crater, hard crater, hill and mountain images are presented in Table 4.2. The relative and absolute error between the true and estimated surface at the last iteration are calculated as

$$J_{abs} = \sqrt{\frac{1}{NM}\sum_{x,y}(z - z_{true})^2}, \tag{4.1}$$

and

$$J_{rel} = \sqrt{\frac{1}{NM}\sum_{x,y}((z - \bar{z}) - (z_{true} - \bar{z}_{true}))^2}. \tag{4.2}$$

where $\bar{z}$ and $\bar{z}_{true}$ are the average of z and $\bar{z}_{true}$ respectively.

4.4 Performance

From the test results on the synthetic easy crater, hard crater, hill and mountain images, clearly *z-only* DFSS can achieve very good estimation on surface topology in a reasonable number of iterations, in the easy crater, hill and mountain image cases. *z-only* algorithms also proves to be robust, especially in small baseline situation. The only difficulty is dealing with images which has very weak shading information, as represented by the hard crater case.

It was shown that the *z-only* DFSS algorithm has much better performance using the two images that a simple shape from shading algorithm which uses only one image.(see [24]) This performance increase validates the fusion approach to obtaining better vision algorithm.

The *z-only* algorithm was also shown to be robust, able to accurately estimate the synthetic surface in the presence of several types of errors. The performance of the algorithm based on images that contains noise, geometry error, or reflectance errors was shown in [24] chapter 7. In Most case, The algorithm was able to form a close estimate.

It is clear the algorithm has considerable difficulties with the hard crater images. The optimization got caught in the local minimum. The reason is clearly shown in the Figure 4-8 for a given brightness level(i.e., along one of the contours), there are two viable solutions with different surface orientations. The local minimum has a orientation in the "dipped" region that is viable but incorrect orientation, and the stereo information is not strong enough to pull it out of the local minimum.

The C version of *z-only* DFSS algorithm runs 2-3 times faster that its counterpart in Matlab, memory requirement is 30 percent less. It is also made more portable. The C functions has been tested on Sun, and IBM compatible PCs.

Chapter 5

Viking Image Results

In the previous chapter, the *z-only* Depth from Shading and Stereo algorithm has been shown to be efficient and robust. In this chapter, the algorithm will be applied to real images, namely areal photos taken during Viking Space Project.

5.1 Viking Space Project

Viking Space Project was started by the National Aeronautics and Space Administration on November 15, 1968. The main objectives of the project was to achieve a soft landing on the surface of the Mars and to relay scientific data back to the Earth. The main function of the orbital cameras, whose pictures are displayed in this thesis were to aid in the selection of safe landing sites to establish the geologic and dynamic environments in which the lander experiments were performed.

Two previous missions, using Mariner 4, mariner 6 and 7 obtained a blend of low-resolution, wide area pictures. Mariner 9, the final predecessor to Viking was the first spacecraft to go into orbit about another planet. It took more than 7300 images of Mars, covered the entire surface at a resolution of 1-3 km, and selected area down to 100 meters.

5.1.1 Viking Mission

The Viking Mission consisted of four spacecrafts: two identical orbiters and two identical landers. One of the orbiter experiment was the Visual Imaging Subsystem (VIS), which acquired the images that are used in this thesis. The major objective of the VIS experiment was to characterize potential landing site.

Viking Orbiter I was launched from Kennedy space center at Cape Canaveral on August 20, 1975, and arrived at Mars on June 19, 1976. Initially, the spacecraft was put into a Mars-synchronous elliptical orbit with a period of 24.66 hours, an apsapsis of 33,000 km and a periapsis of 1513 km. During the first month, Viking I began systematically imaging Mars on July 20, 1976. Its highly elliptical orbit was particularly suited for studying the surface because it allowed a mix of close-up, detailed views at periapsis and long range synoptic view near or at apoapsis. More that 30,000 pictures were taken.

Viking Orbiter II was launched September 9, 1975, and arrived at Mars on august 7, 1976. A major difference in the orbit of this spacecraft compared to that of Viking Orbiter I is its high inclination, which allowed Viking Orbiter II to observe the complex, enigmatic polar regions at relative close range. Viking Orbiter II returned nearly 16,000 pictures of Mars and its satellites.

The Viking orbiter spacecrafts operated in orbit around Mars from 1976 to 1980. The orbiter imaging systems imaged all of the terrains on Mars, collected some color and stereo images, and made observations of phobos and deimos. Some image sequences acquired by the VIS experiment include systematic medium and high resolution coverage of large portions of the surface, stereo images, observations of Phobos and Demios, color images of the equatorial regions, observations of the polar regions, and monitoring dust storm activity.

5.1.2 Viking Orbiter Visual Subsystem

Each Viking Orbiter, Viking Orbiter I and II, was equipped with two identical vidicon cameras, called the Visual Imaging Subsystem (VIS) [26], [14], [1]. Each VIS camera consisted of a telescope, a slow scan vidicon, a filter wheel, and associated electronics. The angular field of view of the camera as defined by the reseau pattern was 1.51 by 1.69 degrees. The ground area covered by an image varies as the function of spacecraft altitude and emission angle. A digital image was generated by scanning the vidicon face plate. A full resolution, uncompressed Viking orbiter image consists of an array of 1056 lines with 1204 samples per line. There are only 1182 samples in each line are valid. The extra 22 are consist of dark bands on the left and right edge of each image, produced by an opaque mark in front of the camera. The images then transmitted back to earth station.

Many Viking orbiter images have missing data and contain some amount of noise [14]. The missing data are mainly due to sample intervals, resulted from the raw data being stored on the spacecraft and transmitted to earth in packets that contain every seventh pixel. The noise found in these images include single-pixel random noise and several source of coherent noise. The random noise is usually due to telemetry errors. Techniques exist to remove the random noise and missing data [4]. The coherent noise arises from shuttering of the adjacent camera, filter wheel stepping, and scan platform movements [14]

5.2 Data selection

Mars provides many features are suitable to use DFSS algorithms to analysis the photo images. Great canyons are incised into the surface, huge dry river beds show the changes in topology features, large volcano tower distinct itself on a flat surface.

However, huge volume of Mars images also pose difficulties for image selection along with some natural obstacles, such as seasonal dust storm, caps od carbon dioide at the poles.

Image pair selection are done by Mike Caplinger. Using their local facilities, Mike Caplinger is able to browse through a large amount of Mars images and pick out the images of the same longitude and latitude. The associated Sun position and space craft position and angles are obtained. These geometry information and the image data are then sent to us to use with DFSS algorithm.

5.3 Data Processing

Two preliminary processing steps are always done to radiometrically and geometrically calibrate the images:

- Viking Orbiter images are radiometrically calibrated by converting the digitized signal from the camera into a quantity that is proportional to the radiance reaching the sensor. Each Viking camera was calibrated before flight. In addition, changes in the calibration over time has been estimated from analyses of images of deep space and dust storms. The radiometric calibration procedure applies additive and multiplicative corrections that account for the varying sensitivity of the vidicon across the field of view and over time. The calibrated values are proportional to radiance factor, which is defined as the ratio of the observed radiance to the radiance of a normally illuminated Lambertian reflector of unit reflectance at the same heliocentric distance.

- Geometric calibration of Viking Orbiter images removes electronic distortions and transforms the point perspective geometry of the original image into a map projection. The electronic distortions are barrel-shaped distortions from the electron beam readout and complex distortions from interactions between the charge on the vidicon face plate and the electron beam. The electronic distortions are modeled be comparing the predicted locations of undistorted reseau marks with the actual locations in an image.

During the processing, reseau marks are removed, bit errors and/or tape errors are corrected. The raw Mars images are projected using the nominal imaging geometry to a projection such as sinusoidal or Mercator, relative to a reference spheroid. This has the effect of removing any gross effects caused by the curvature of the planet, but since the small scale topography is not modeled, the disparities it induces remain. The images are also rotated so that epipolar lines are parallel to the scanlines. These selected and processed data are then subsampled to 256 by 256 pixel for DFSS analysis. The geometry information are transformed into the aligned global coordinate system, as described in section 2.5. The down-sizing is done only because limited time and CPU power to process full resolution images. [1]

5.4 Data Analysis

In this section, I present two typical examples of Viking image pairs. The DFSS result are shown to demonstrate the effectiveness and robustness of *z-only* Depth from Shading and Stereo algorithm.

5.4.1 Example 1

The first pair of images is of an terrain surface similar to eroded hills and river bed. Figure 5-1 and Figure 5-2 shows shows the camera geometry and its projection in xz and yz plane. The light sources come from almost the same direction. Fortunately, the light source direction is very oblique and different from the camera direction results in much stronger shading information comparing with hard crater case. Table 5.1 lists the geometric information about this pair of images.

The Camera Angles are listed in the right ascension, declination, and twist of the camera. The Spacecraft Vector is the position of the spacecraft/camera in the reference frame of the planet. The Planet Angles are the orientation of the planet in the

[1]At 256 by 256 pixel, it takes about 20 hours to finish 1200 iterations

	Camera Angle	Spacecraft Vector	Planet Angle	Sun Vector
Left	-68.449593	-1324.127686	52.694637	154312733.273824
	313.565796	1092.557129	317.313019	159262758.890330
	274.407928	5395.773438	225.757187	68965427.707157
Right	-63.648964	-1796.010010	52.694637	152618273.748284
	339.906433	853.954346	317.313019	160370845.199162
	297.853882	5504.281250	225.624329	69519512.685777

Table 5.1: Geometric Information about the first pair of Viking Images

same reference frame as the Camera Angles, also in right ascension, declination and twist. The Sun Vector is the position of the Sun in the planet reference frame. These information of orientation and position are converted to the geometric information in the global reference frame and shown in Figure 5-1 and Figure 5-2. Figure 5-3 shows the reflectance map contour and the image pair.

Figure 5-4 to Figure 5-7 shows the results of applying the DFSS algorithm to this pair of images. *z-only* DFSS algorithm performs well, correctly estimated the complicated topology. Obviously, considerable more iterations are required to obtain a satisfactory estimate of the surface than the synthetic case. Even so, the algorithm formed a stable estimation after about 500 iterations. Also, the cost function remains higher than that in synthetic image case.

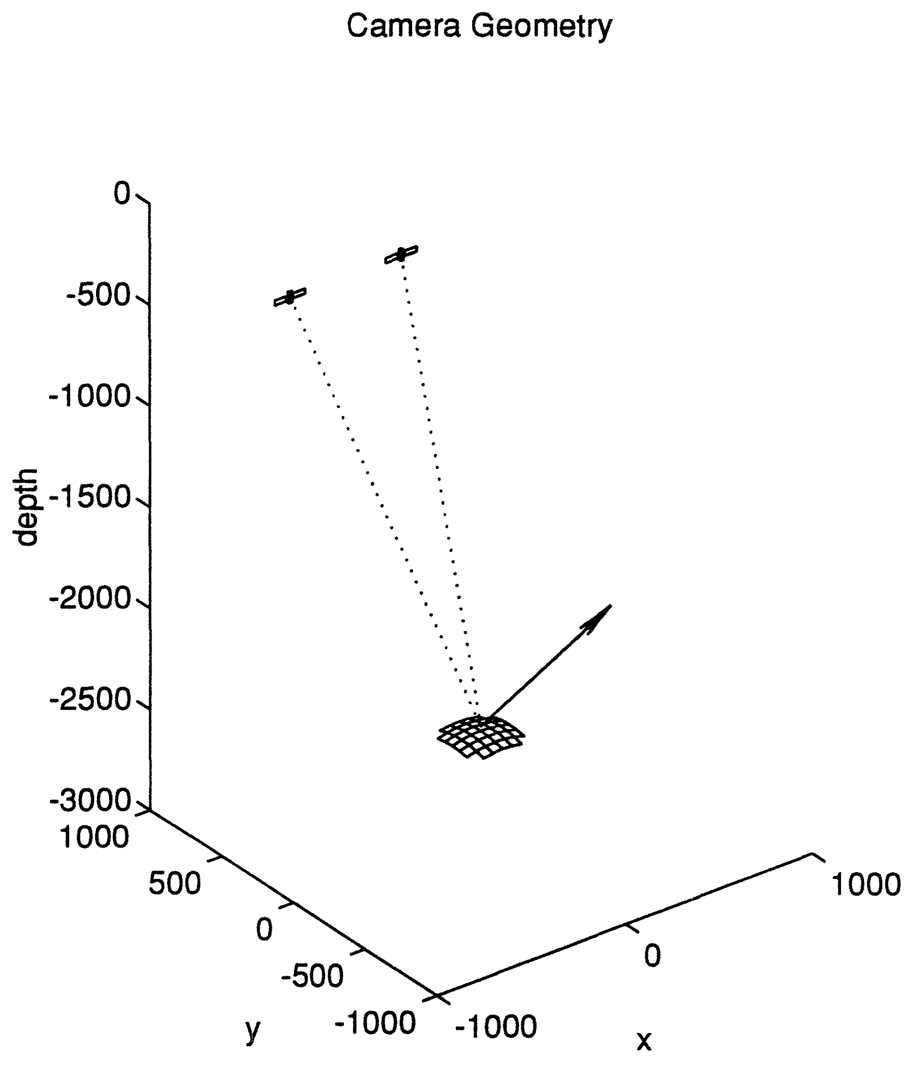
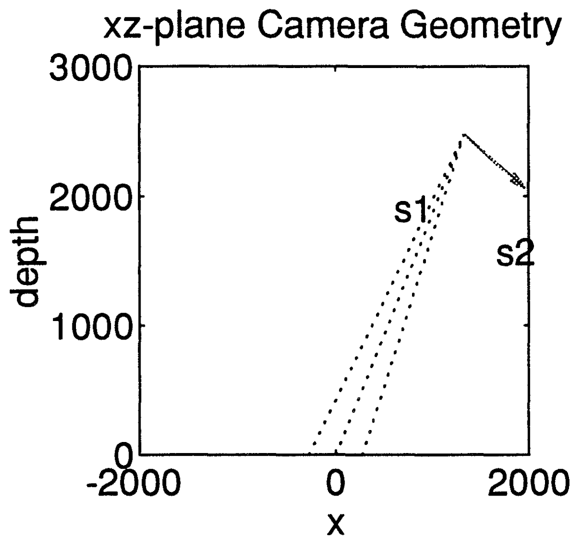


Figure 5-1: Mars Viking Images pair Camera Geometry. Graph shows the camera position and direction(dotted lines), as well as the light source direction(solid lines)



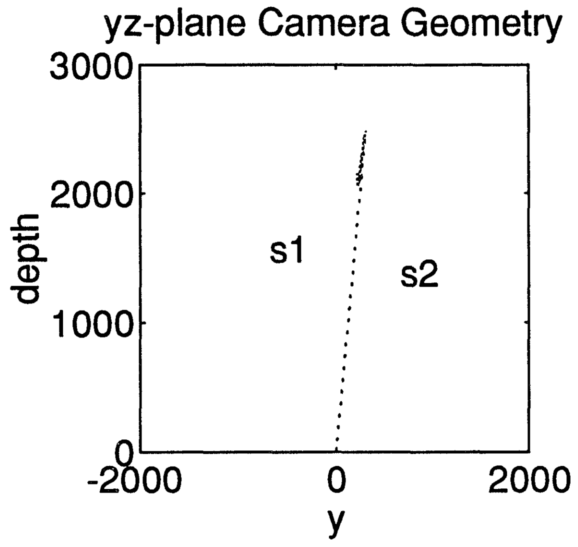
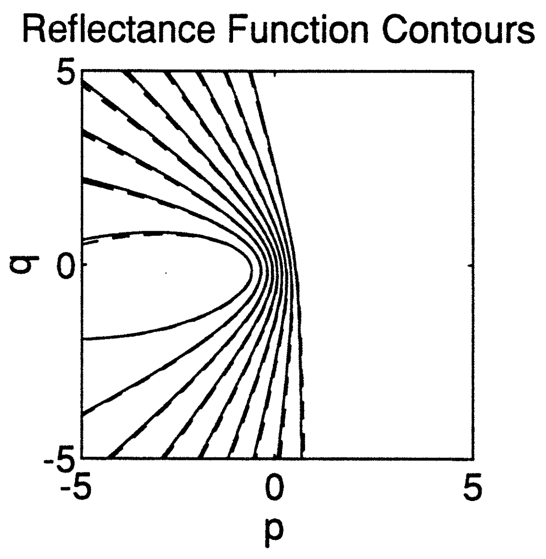


Figure 5-2: Mars Viking Images pair Camera Geometry projected in xz and yz plane



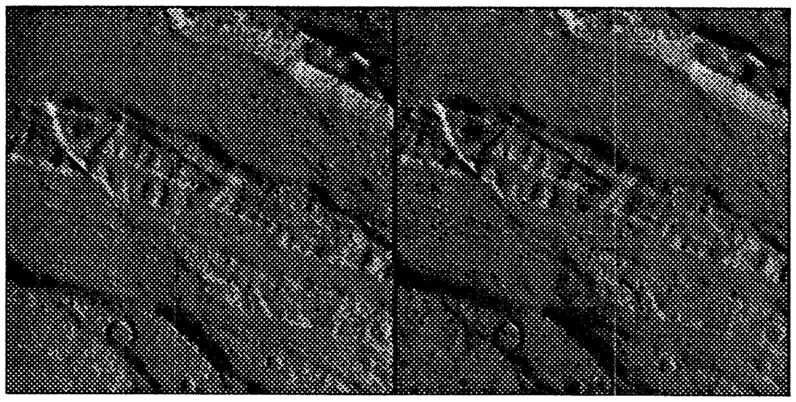


Figure 5-3: Mars Viking images pair and gradient contour

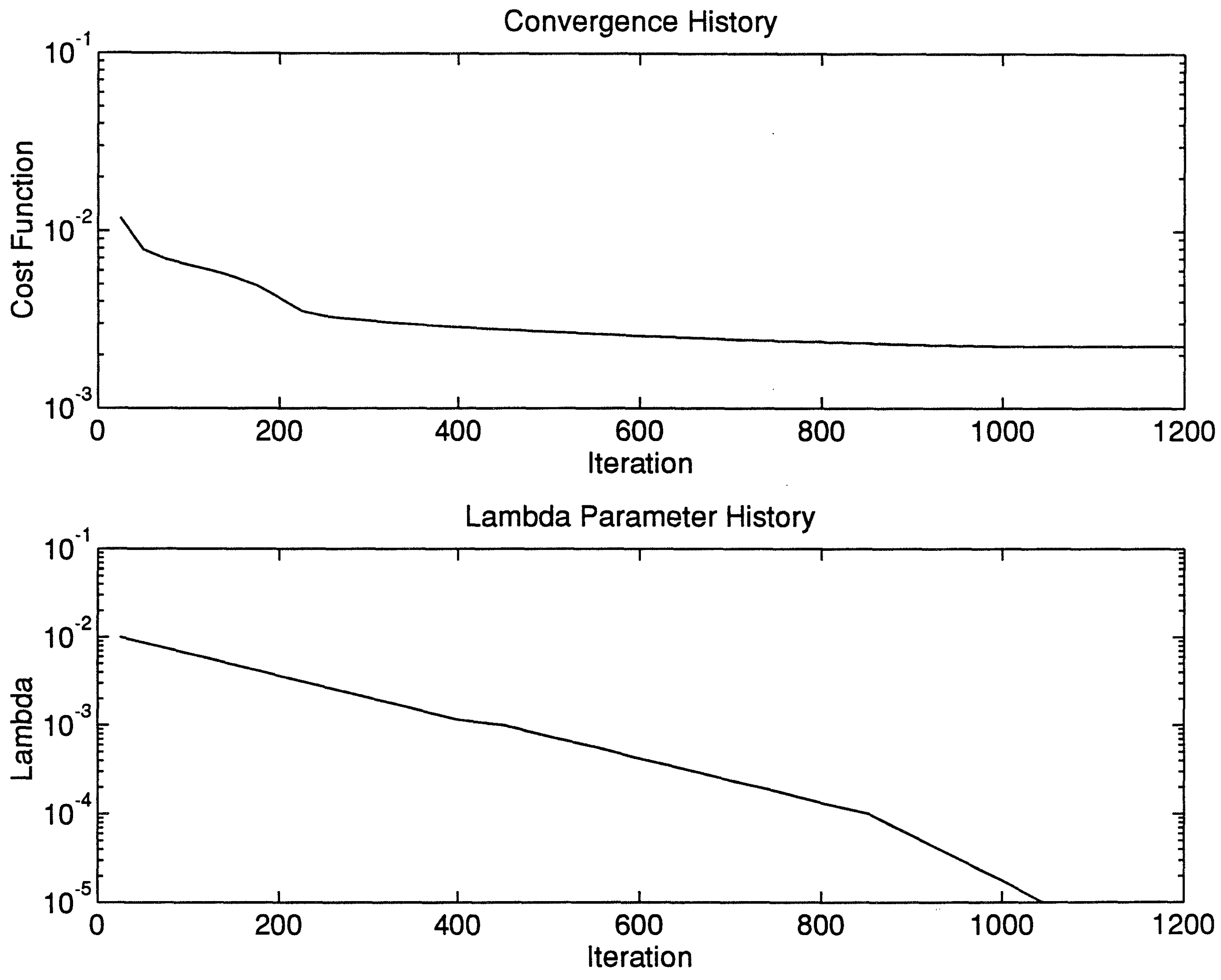
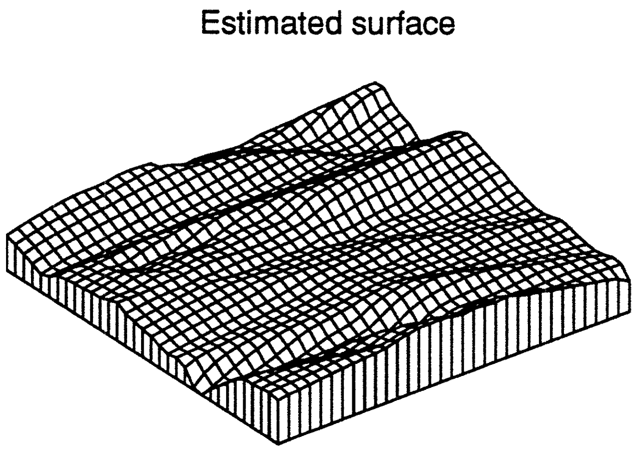


Figure 5-4: DFSS iteration history on Mars Viking Image pair. Top graph shows the history of cost function. Bottom graph shows the history of the smoothness weighting term λ



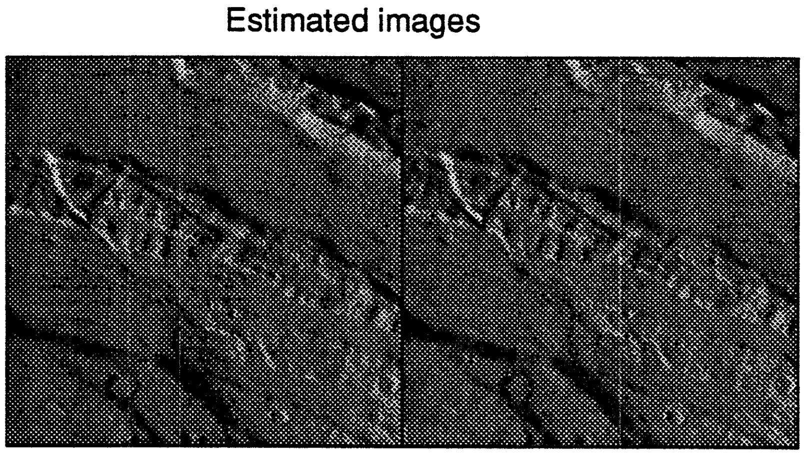


Figure 5-5: DFSS Estimated images and Surface on Mars Viking Image pair at the end of iterations

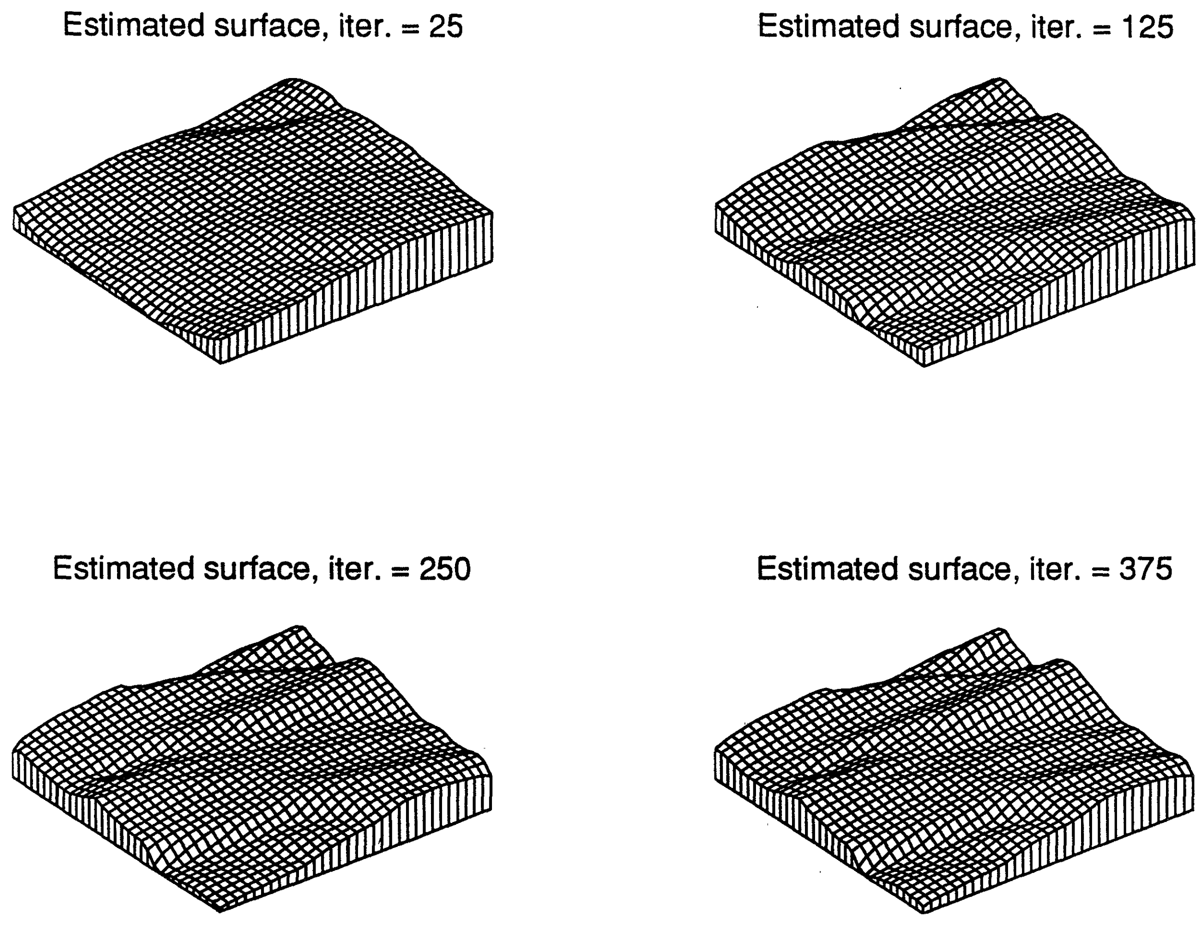

Estimated surface, iter. = 25
Estimated surface, iter. = 125
Estimated surface, iter. = 250
Estimated surface, iter. = 375

Figure 5-6: DFSS estimation at various iteration steps on Mars Viking Image pair

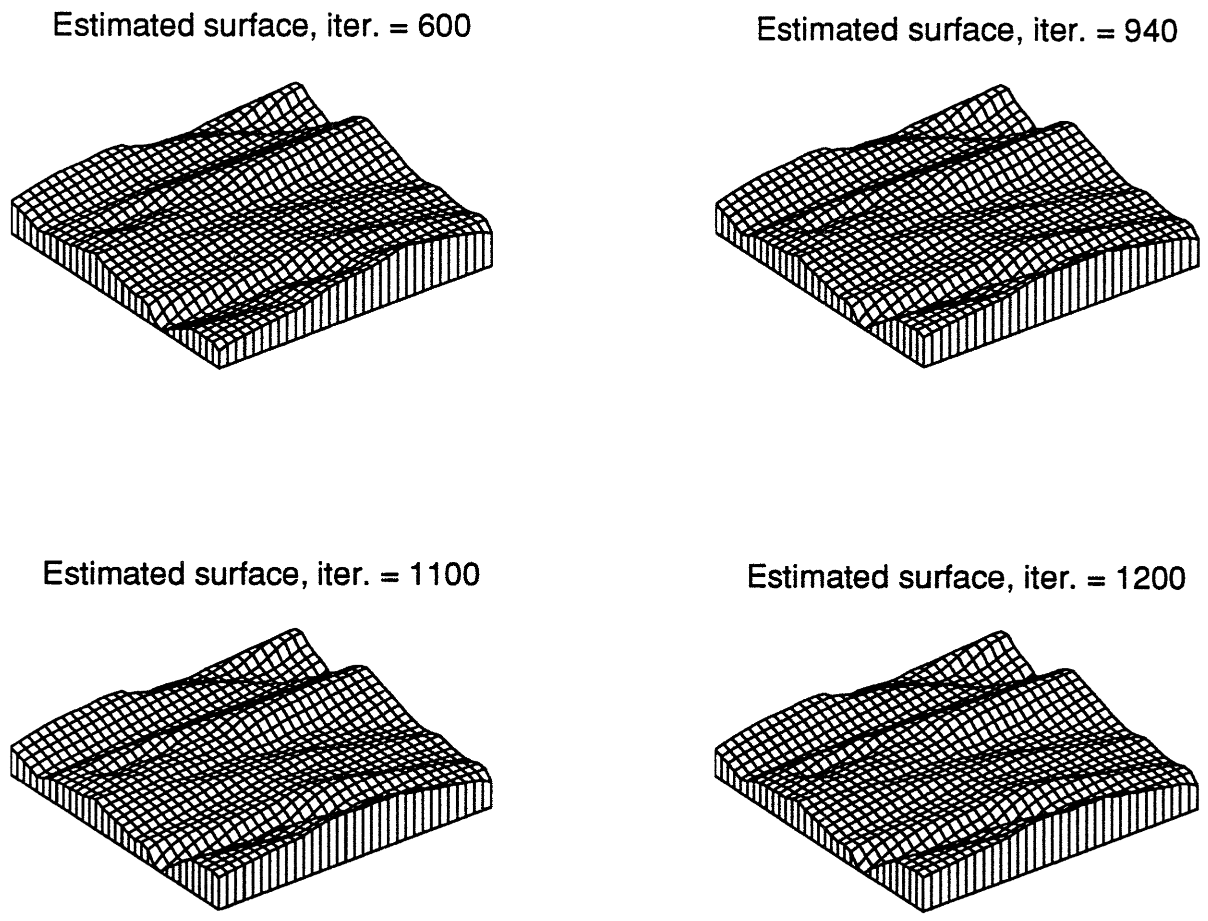

Estimated surface, iter. = 600
Estimated surface, iter. = 940
Estimated surface, iter. = 1100
Estimated surface, iter. = 1200

Figure 5-7: DFSS estimation at various iteration steps on Mars Viking Image pair

	Camera Angle	Spacecraft Vector	Planet Angle	Sun Vector
Left	-38.500999	3031.000000	52.694817	247003002.182980
	-114.101997	3252.000000	317.313354	-6496592.410365
	-153.830994	2154.000000	60.560833	-9607699.669818
Right	-16.581617	3431.279785	52.694775	241693638.758362
	221.748749	3228.368896	317.313263	37984763.457939
	250.999435	1721.439697	66.486511	10952544.914326

Table 5.2: Geometric Information about the first pair of Viking Images

5.4.2 Example 2

The second pair of images is also of an terrain surface similar to eroded hills and river bed. Figure 5-8 and Figure 5-9 shows the camera geometry and its projection in xz and yz plane. The light sources come from almost the same direction. Similar to example 1, there is little separation in the direction of the light sources. This also presents a challenge to the DFSS algorithm. The table below Table 5.2 lists the geometric information about this pair of images. In Figure 5-10, the reflectance map contour and the image pair are shown.

The Camera Angles are listed in the right ascension, declination, and twist of the camera. The Spacecraft Vector is the position of the spacecraft/camera in the reference frame of the planet. The Planet angles are the orientation of the planet in the same reference frame as the Camera Angles, also in right ascension, declination and twist. The Sun Vector is the position of the Sun in the planet reference frame. These information of orientation and position are converted to the geometric information in the global reference frame and shown below.

Figure 5-11 to Figure 5-14 shows the results of applying the DFSS algorithm to this pair of images. *z-only* DFSS algorithm performs reasonably well, closely estimated the complicated topology. Obviously, considerable more iterations are required to obtain a satisfactory estimate. Similarly to example 1, after 500 iterations, DFSS algorithm formed a stable estimation.

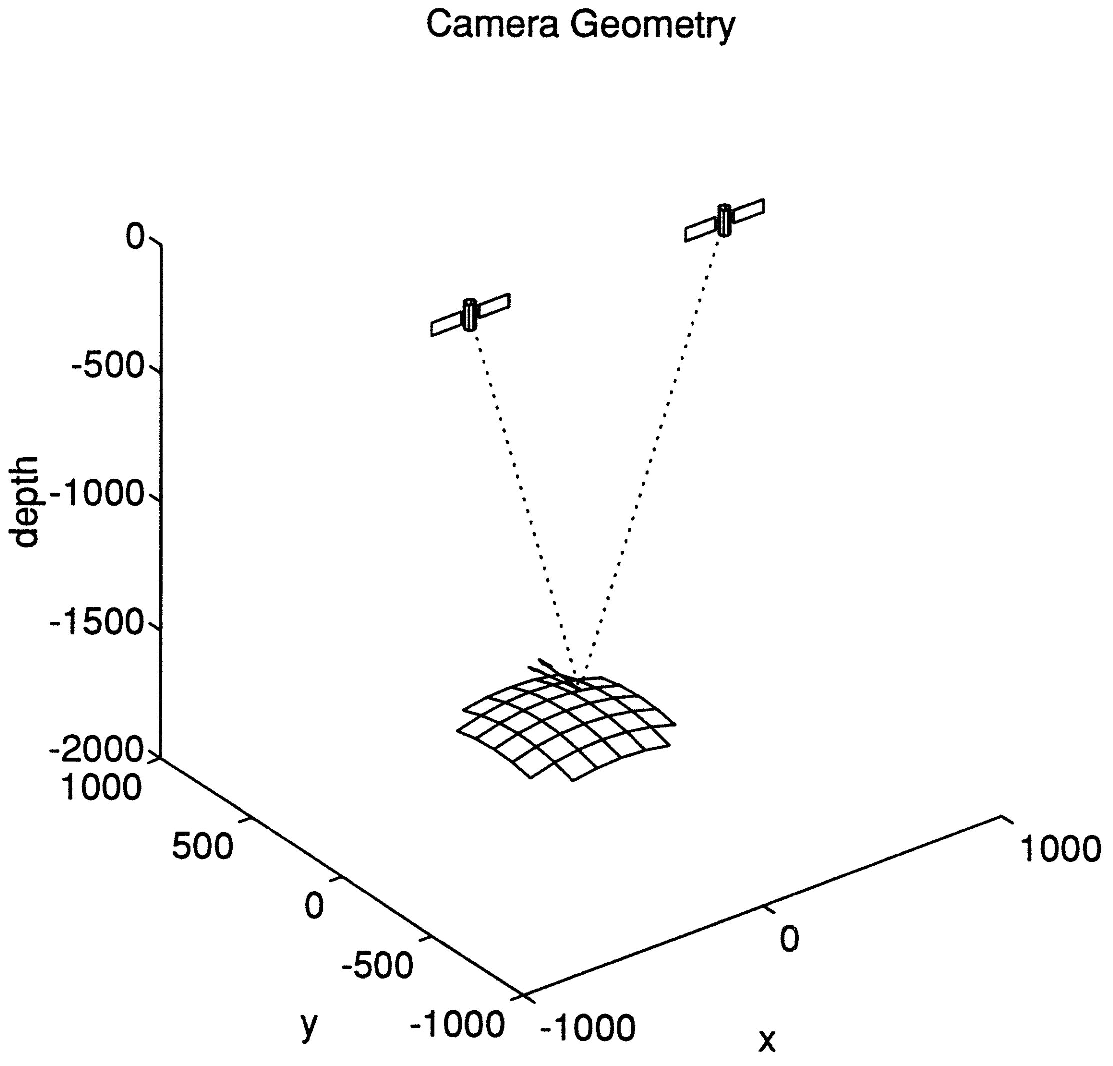
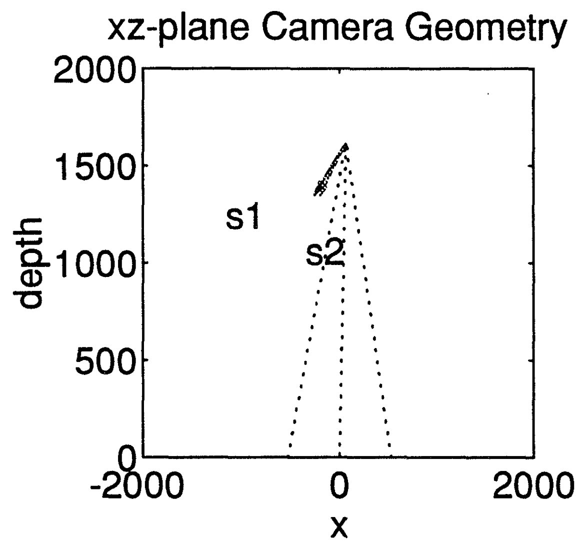


Figure 5-8: Mars Viking Images pair Camera Geometry. Graph shows the camera positions and directions(dotted lines), also the light source direction(solid lines).



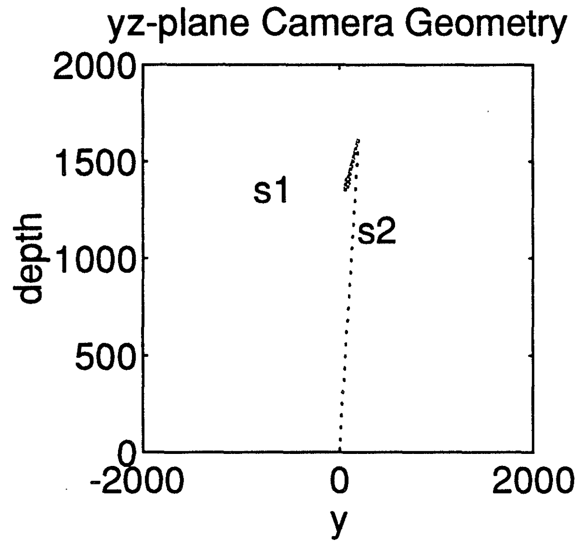
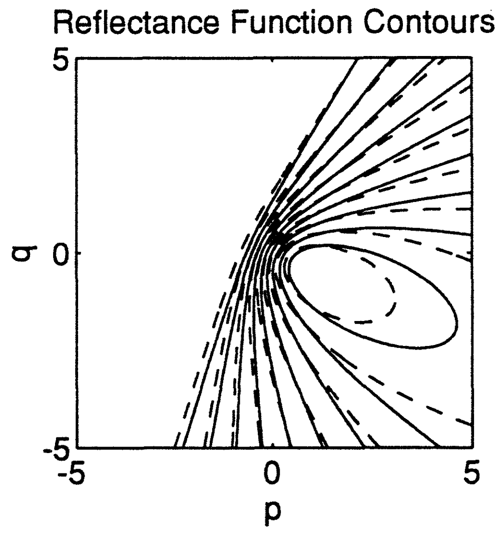


Figure 5-9: Mars Viking Images pair Camera Geometry projected in xz and yz plane



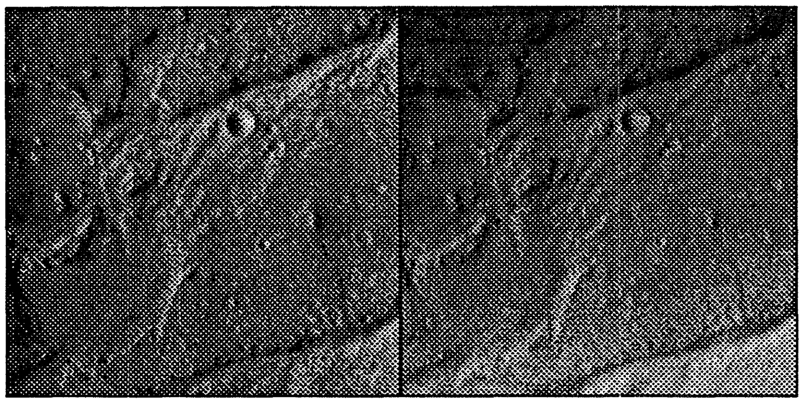


Figure 5-10: Mars Viking Images pair and gradient contour

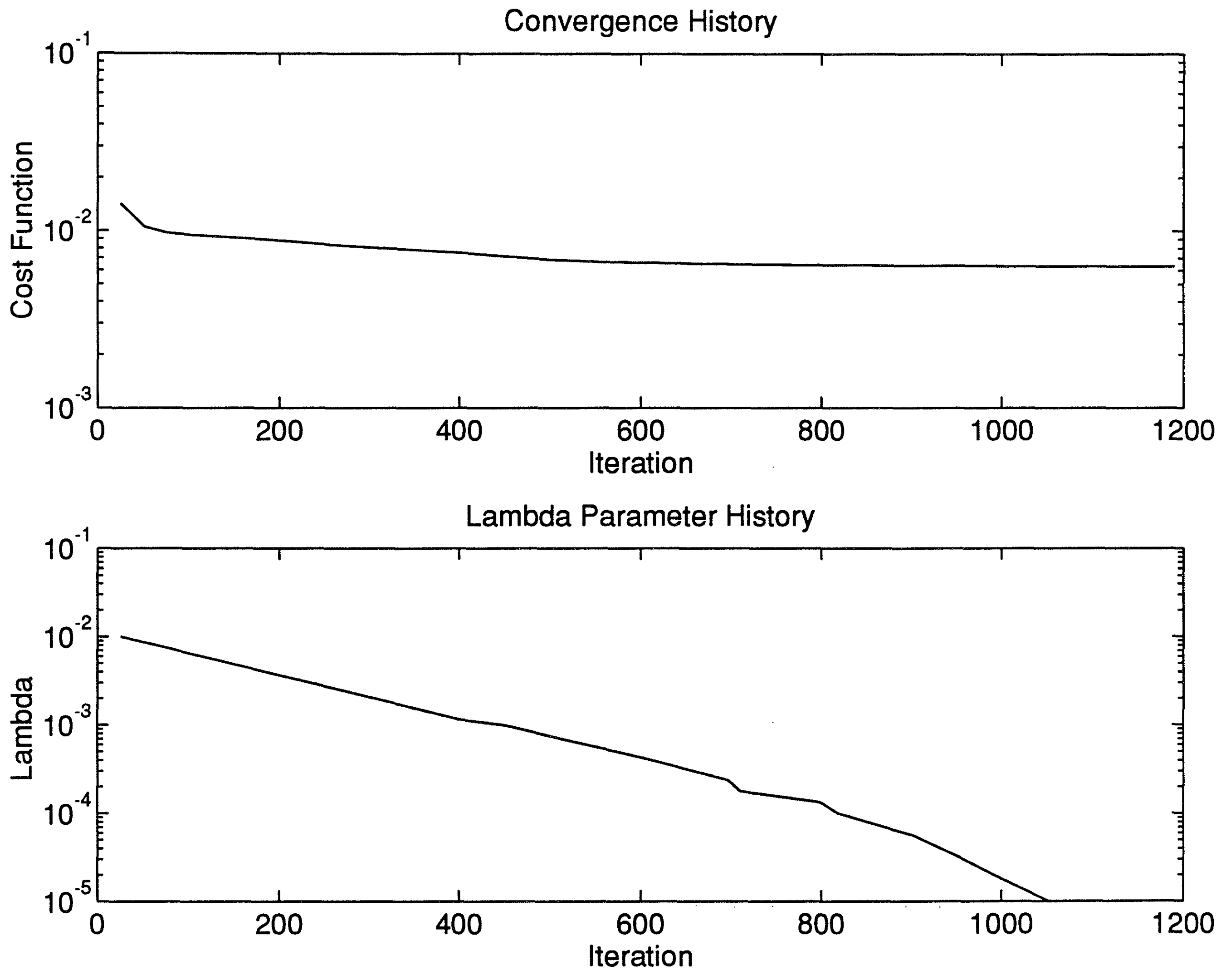
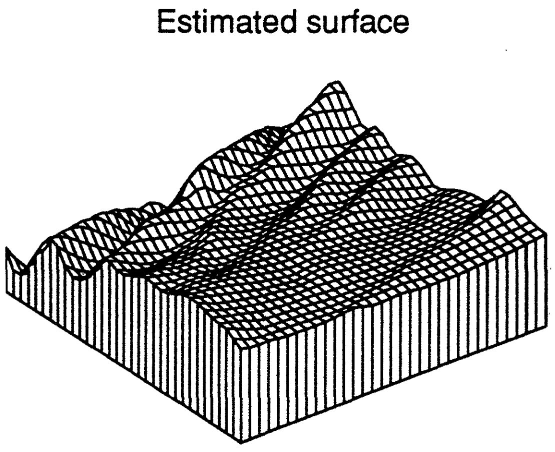


Figure 5-11: DFSS iteration history on Mars Viking Image pair. Top graph shows the history of cost function. Bottom graph shows the history of smoothness weighting term λ



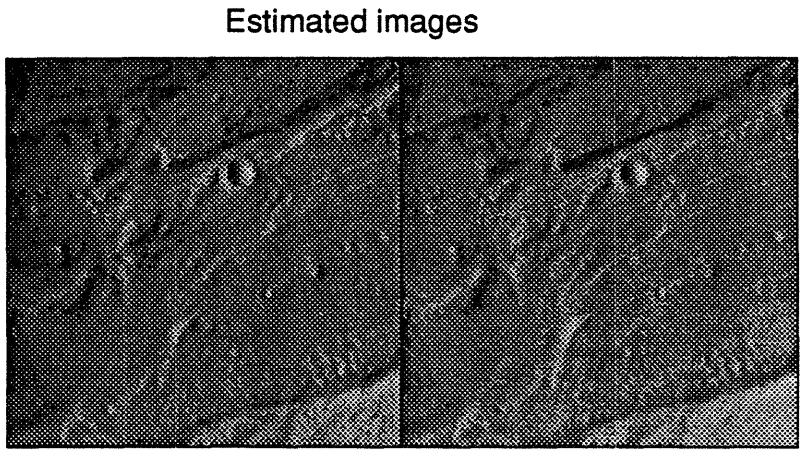


Figure 5-12: DFSS Estimated Surface on Mars Viking Image pair at the end of iterations

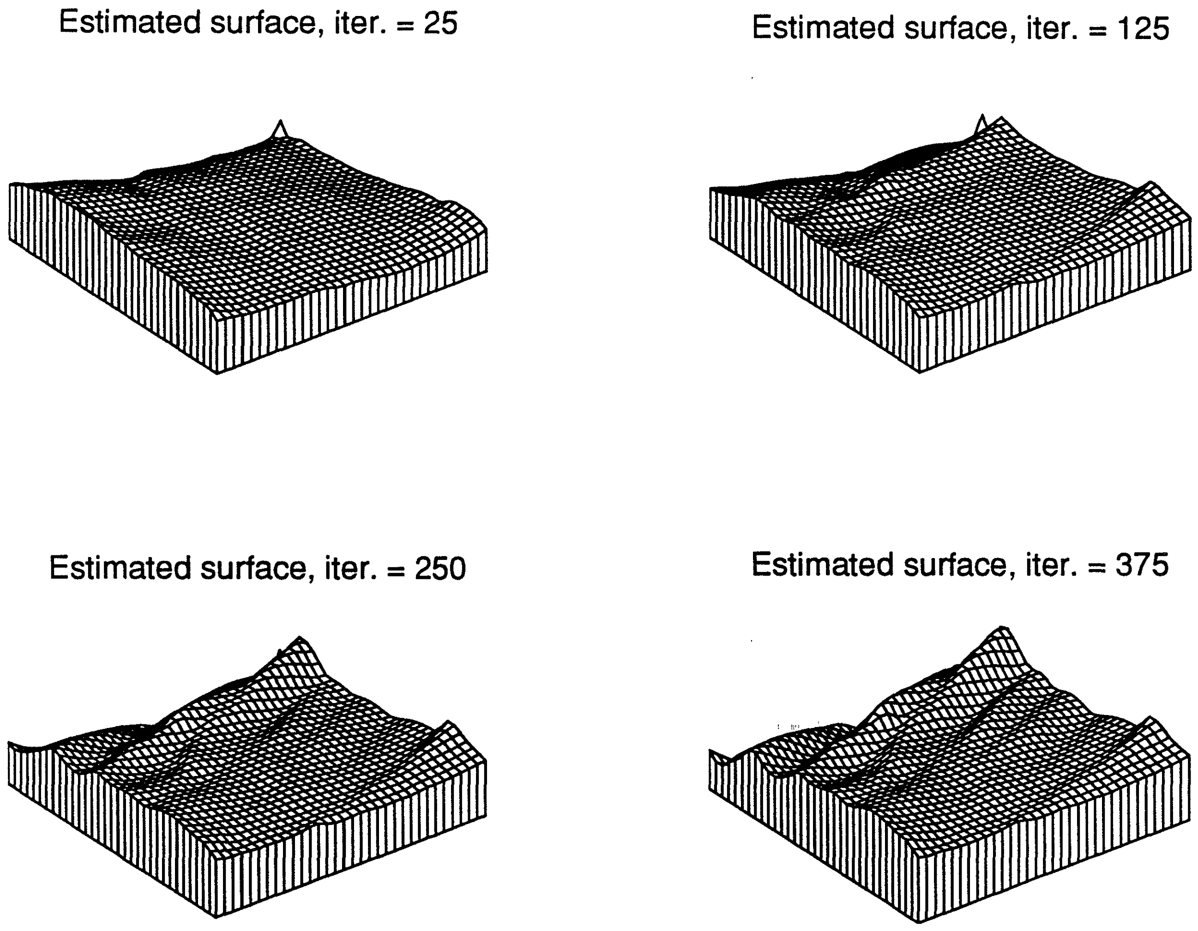

Estimated surface, iter. = 25
Estimated surface, iter. = 125
Estimated surface, iter. = 250
Estimated surface, iter. = 375

Figure 5-13: DFSS estimation at various iteration steps on Mars Viking Image pair

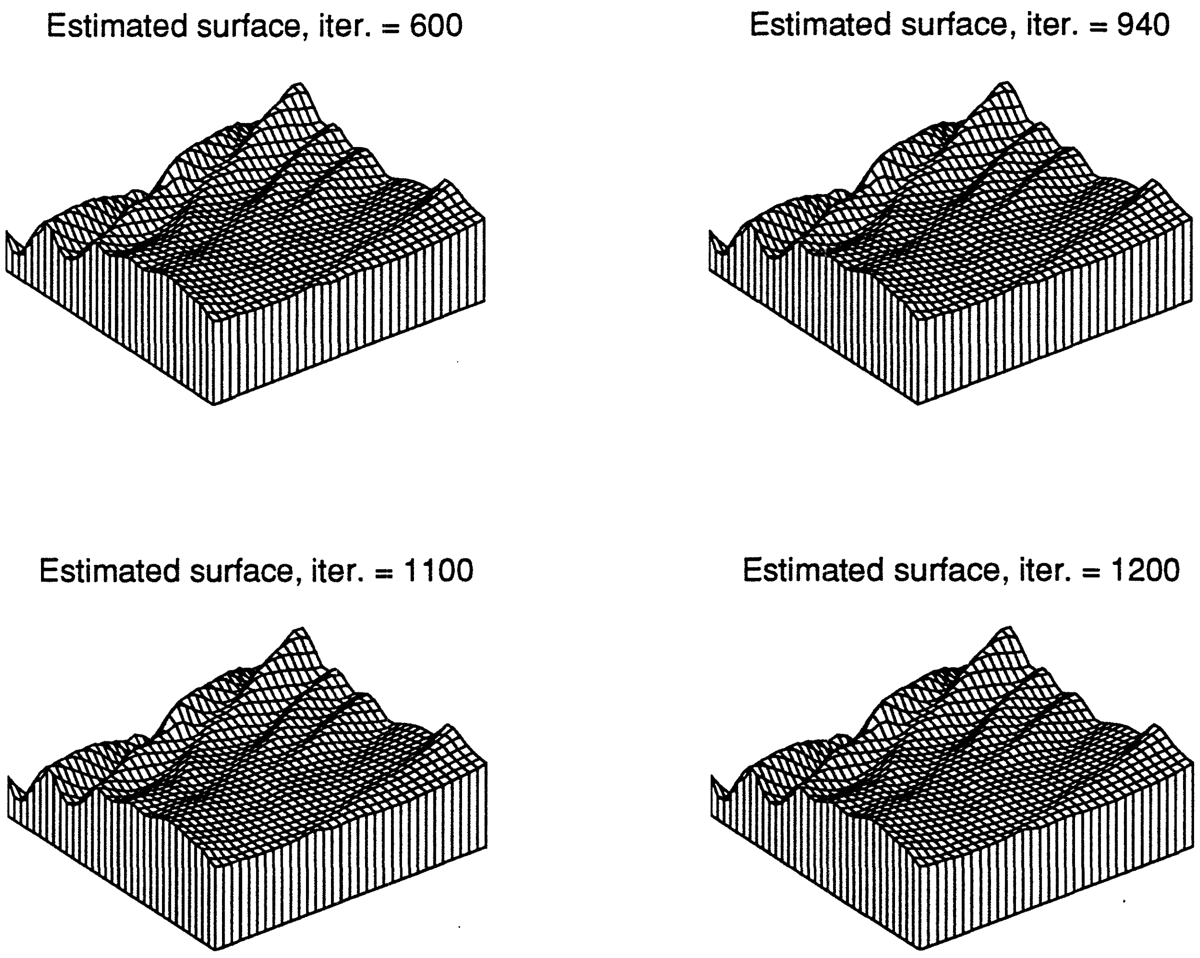


Figure 5-14: DFSS estimation at various iteration steps on Mars Viking Image pair

5.5 Summary of the Results

The results of *z-only* DFSS algorithms running on two example Mars Viking image pairs are presented in the previous sections. They've both demonstrated the adequate performance and robustness of this algorithm. Especially, the algorithm's performance under which the two light sources come in from the same or almost the same direction.

From the test results on the real Viking image pairs, clearly *z-only* DFSS can form close estimation on surface topology in a reasonable number of iterations. There is a general increase of the iterations needed comparing with synthetic image case. But after 500 iterations, the estimate is already quite stable and quite close to the true surface.

The result also shows that *z-only* algorithm is robust, able to estimate the topological surface in the presence of several types of errors.

One thing is not addressed clearly by the earlier two examples, that is one of the advantages of Shape from Shading is to be able to detect details of the shape. Comparing the estimated images and the real images, one can see most of the features are represented. In the mesh plot, however, it is not obvious. This is largely due to the coarseness of the mesh plot. To demonstrate that the detailes are there, we can zoom in on one particular feature. In this case, I chose the crater like feature in the second example.

Clearly the crater feature is vivid in the enlarged portion of the real image. Both the estimated image and surface present such a feature, however, a cluster of much smaller craters which are visible in real image is smeared and hardly recognizable in the estimated image and surface.

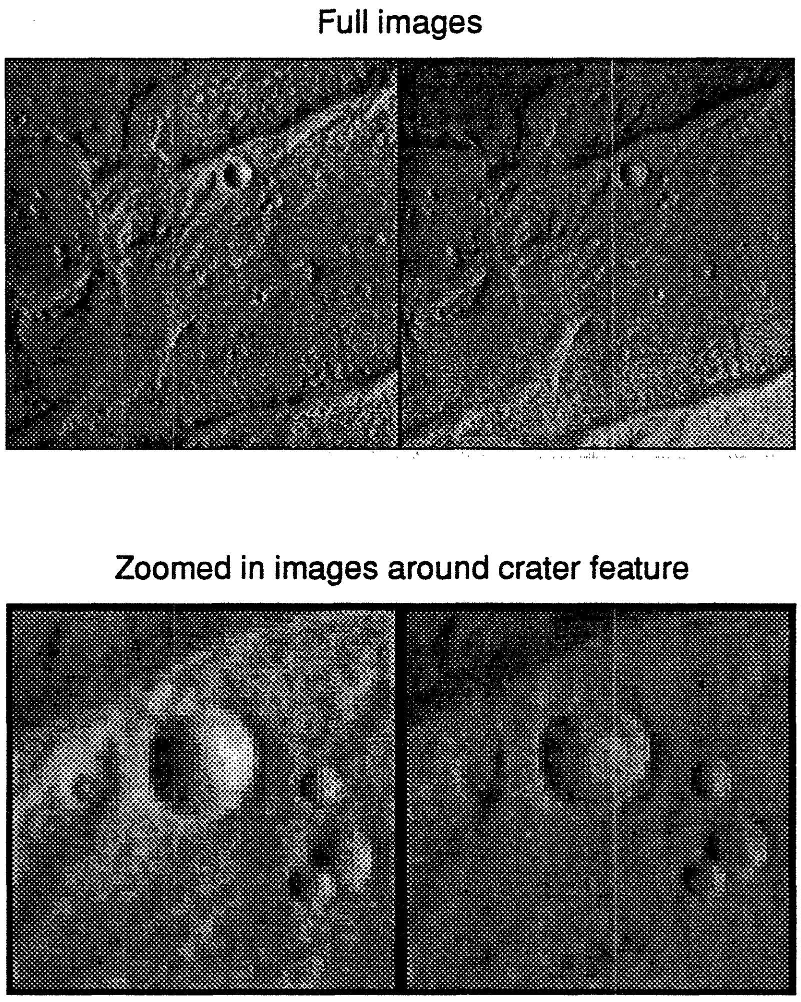


Figure 5-15: Detailed look of the crater like feature in example two, real images and zoomed in images around the crater.

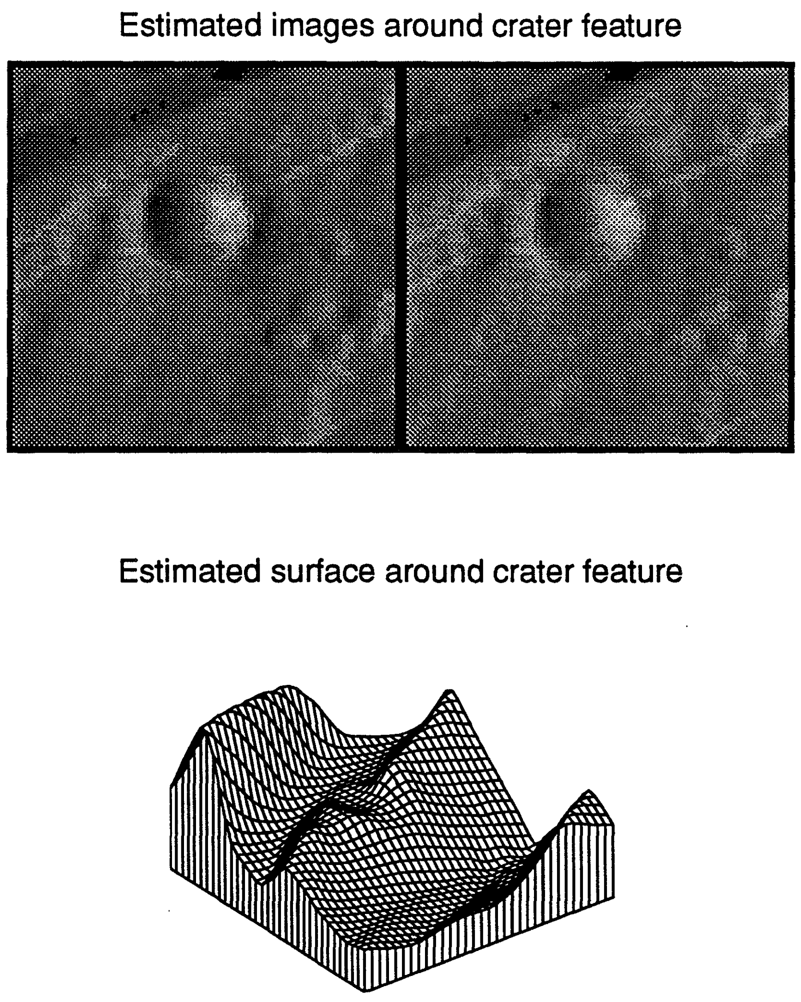

Estimated images around crater feature
Estimated surface around crater feature

Figure 5-16: Detailed look of the crater like feature in example two, estimated images and surface around the crater.

Chapter 6

Error Analysis

It is of crucial importance to understand the various errors and assumptions associated with the images and the methods used to analysis them in earlier chapter.

6.1 Theoretical Error

It is important to review the assumptions behind those simplified equations. The perspective projection equations assume perfect lenses and perfect knowledge of the camera principal points and optical axes. The surface radiance equation assumes we have perfect knowledge of the surface reflectance properties, light source directions, and all surface points as visible from both cameras. The simplified form of the image irradiance equation assumes we either have a perfect sensor or we can perfectly calibrate the sensor to remove any abnormalities from the sensor & lens combination. The stereo equations assume we know the relative position and orientation of the two camera perfectly. the only assumption that are truly artificial are the assumptions of perfect knowledge of the reflectance maps and the the assumption of no self-illumination. With more careful measurements and more expensive equipment, it is possible to approach perfect knowledge of other assumptions. The assumption that all surface points be visible merely restricts the roughness of the surface that this research is applicable to.

It is as well as important to go over the simplified equations and point out the their effects. There are several simplifications that have been made to the equation.

1. A special global coordinate system that is halfway between the two camera position.
2. All surface points are assumed to be visible from the two cameras.
3. The reflectance properties of the surface are assumed to be constant and *Lambertian* allowing the viewing direction dependence to be dropped from the reflectance equations. In addition, it is assumed that there is no interflection between different parts of the surface.
4. The surface is assumed to have constant albedo allowing the position dependent term to be dropped.
5. The cameras optical axis are assumed to be aligned with each other allowing the rotational transformation to be set to identity.

With all the above simplifications, the set of equations are:

$$\begin{aligned} E^{(1)}(\mathbf{r}_1) &= R^{(1)}(\hat{\mathbf{n}}) \\ E^{(2)}(\mathbf{r}_2) &= R^{(2)}(\hat{\mathbf{n}}) \end{aligned} \tag{6.1}$$

where

$$\begin{aligned} \mathbf{r}_1 &= \frac{f(\xi + \mathbf{b}/2)}{(\xi + \mathbf{b}/2)\cdot\hat{\mathbf{z}}_1}, \\ \mathbf{r}_2 &= \frac{f(\xi - \mathbf{b}/2)}{(\xi - \mathbf{b}/2)\cdot\hat{\mathbf{z}}_2}, \end{aligned} \tag{6.2}$$

If we define $z = \xi \cdot \hat{\mathbf{z}}_G$ and $\mathbf{r} = f\xi/z$, then the constraint function can be written as

$$\begin{aligned} E^{(1)}(\mathbf{r} + \frac{f\mathbf{b}}{2z}) &= R^{(1)}(\hat{\mathbf{n}}) \\ E^{(2)}(\mathbf{r} - \frac{f\mathbf{b}}{2z}) &= R^{(2)}(\hat{\mathbf{n}}) \end{aligned} \tag{6.3}$$

Use explicitly $\mathbf{r} = (x, y, f)$ and the normal vector $\hat{\mathbf{n}}$ is parameterized using gradient component p and q,

$$\hat{\mathbf{n}} = \frac{(-p, -q, 1)}{\sqrt{p^2 + q^2 + 1}} \tag{6.4}$$

where

$$\begin{aligned} p &= \frac{f z_x}{x z_x + z} \\ q &= \frac{f z_y}{y z_y + z} \end{aligned} \tag{6.5}$$

The photo-topography equations then become

$$\begin{aligned} E^{(1)}(x + \frac{fb}{2z}, y) &= R^{(1)}(p, q) \\ E^{(2)}(x - \frac{fb}{2z}, y) &= R^{(2)}(p, q) \end{aligned} \tag{6.6}$$

In order to relate positions in the image direction vectors in 3-D space, the origin of the camera coordinate system must be known. Finding this origin is the classical *interior orientation* problem. In the special coordinate system we use here, see Figure 2-8, the position of each camera coordinate system origin can be specified by a vector $\mathbf{v} = (v_x, v_y, f^T)$. these vector specify the offset of the camera coordinate origin for each image. Suppose $\mathbf{v}_0$ is the offset to an object point in the global coordinate system, then the offset to this same point in the camera images are

$$\mathbf{v}_1 = \mathbf{v}_0 + \frac{fb}{2z_0}, \tag{6.7}$$

$$\mathbf{v}_2 = \mathbf{v}_0 - \frac{fb}{2z_0}. \tag{6.8}$$

the values of $\mathbf{v}_1$ and $\mathbf{v}_2$ can be quite large in the aligned coordinate system indicating that the images must be shifted far away from the camera coordinate system origin. While this is not possible physically, it is a consequence of re-projecting real images into the aligned coordinate system.

In removing viewing direction dependency, we use orthographic projection for the reflectance function while retaining perspective projection for everything else. This introduced an error into the calculations since orthographic and perspective projection do not produce the same viewing direction in general. This error is so small for planet photo-topography, since the cameras are so far away from the surface. The large viewing distance results a very small relative change in depth. This is especially true here due to the small viewing angle.

Also, this algorithm only deals with uniformly colored surface, in another word, no varying albedo. This assumption is quite valid in the case we are discussing.

The above simplified equations helped formulate the algorithm and produce reasonable result, it's also hidden some important information, such as the normal vector is parameterized using gradient components p and q, instead of vector. Also it imposed restrictions such as all the surface point are visible from two cameras.

6.2 Image Errors

As discussed in earlier chapter, the images taken during Viking Project consisted of many error, noise and needs calibration and corrections.

Many Viking Orbiter images are missing data and contain some amount of noise. a common pattern of missing data is a series of vertical bars with zero value pixel spaced at an interval of 7 samples. In addition, data for a few horizontal image lines may be missing and such lines are filled with zero values.

The noise found in these images include single-pixel random noise and several source of coherent noise. The random noise is usually due to telemetry errors. The coherent noise arises from shuttering of the adjacent camera, filter wheel stepping, and scan platform movements. The coherent noise typically exists in the top or bottom 100 lines of an image and appearing as a "herring bone" pattern. Box filtering techniques that fill in zero values or average the bright and dark spikes of random noise are often successful and used.

As described in chapter 5, the Viking images are radiometrically calibrated by converting the digital signal received from the camera to a quantity that is proportional to the radiance reaching the sensor.

6.3 Lighting Condition, Camera Position and the effects on DFSS results

More important here is the effects of the geometry information under which the images are taken. To illustrate the effects, we regenerate the synthetic easy crater case, under similar lighting, camera geometry and baseline condition of to that of the real images, in particular example 1 in Chapter 5.

Similar to the set used in chapter 4, these images are based on a crater on a flat surface. It is very simple, and thus serves a good test bed. As shown in Figure 6-1, comparing this pair with the pair in chapter 4, contour plots of the reflectance map as well as true images clearly show the effect of the change of lighting. Examing the estimated images and surface, apparently *z-only* DFSS algorithm estimated this test crater surface. But notice the flat background is distorted, which means under this kind of lighting condition, a small anomaly can affect the estimated surface and the programs' convergence. The DFSS results is shown in Figure 6-4.

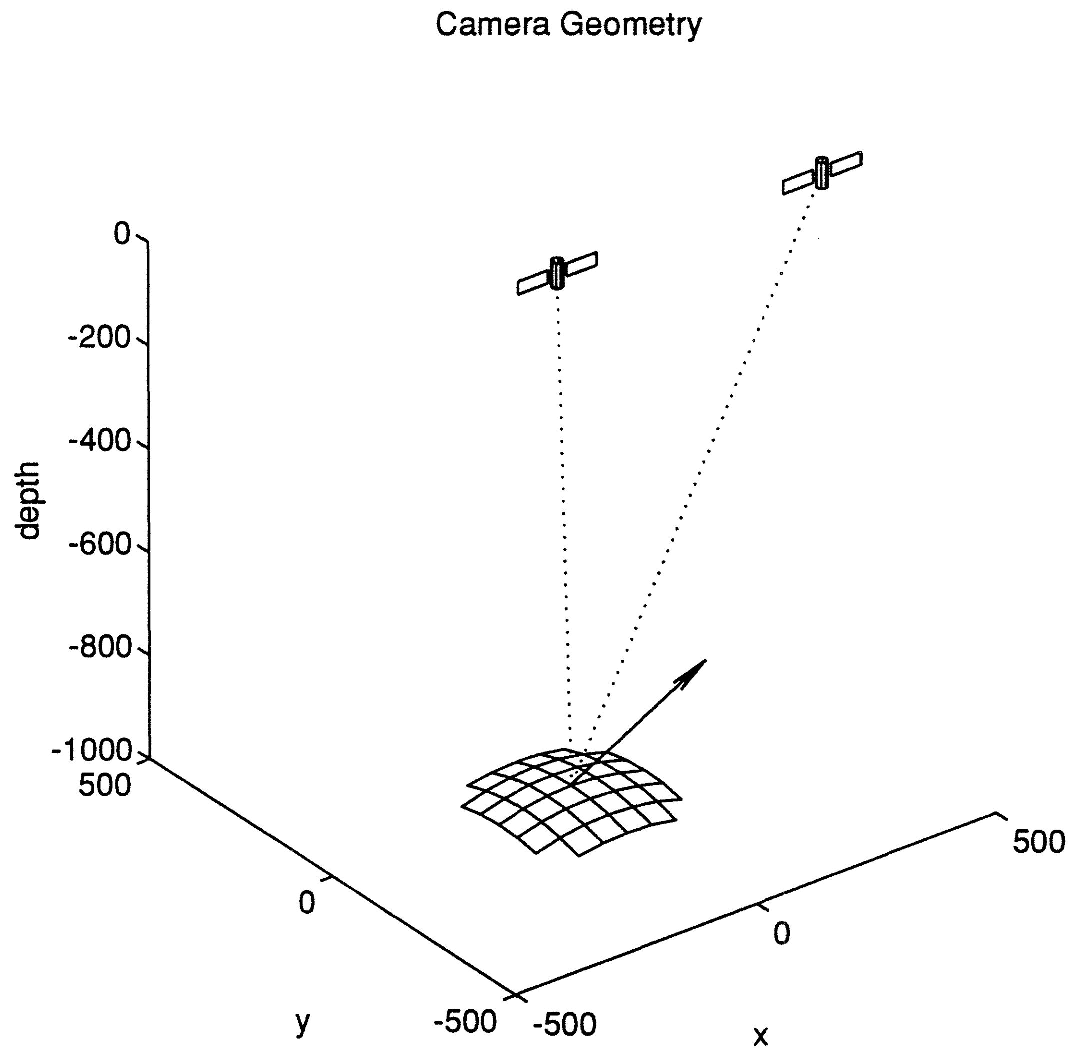
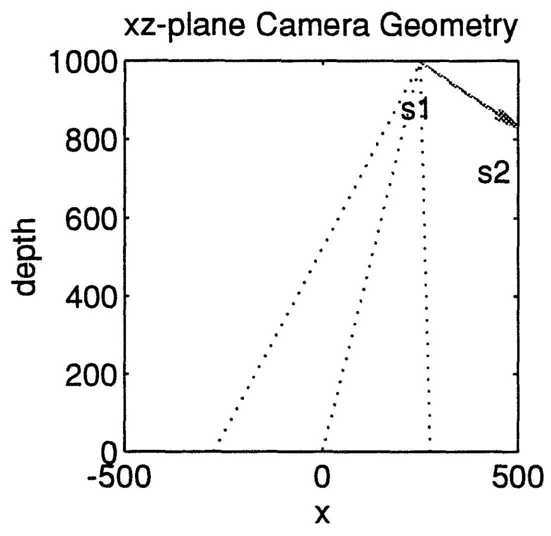


Figure 6-1: Test Crater Synthetic Images Camera Geometry. Graph shows the camera position and direction(dotted lines) and light source direction(solid lines)



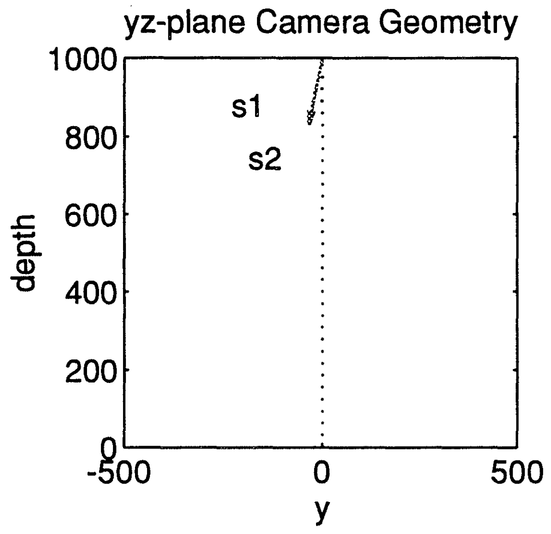
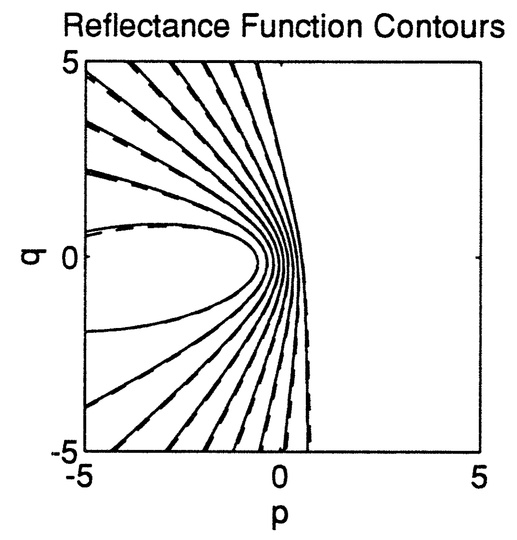


Figure 6-2: Test Crater Synthetic Images Camera Geometry projected in xz and yz plane



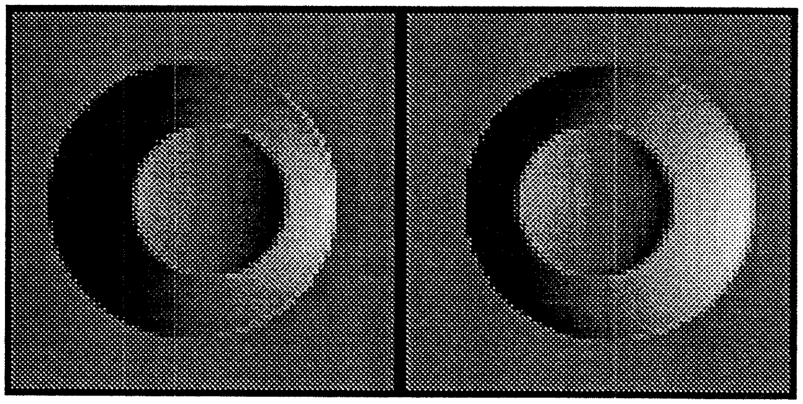


Figure 6-3: Test Crater Synthetic Images

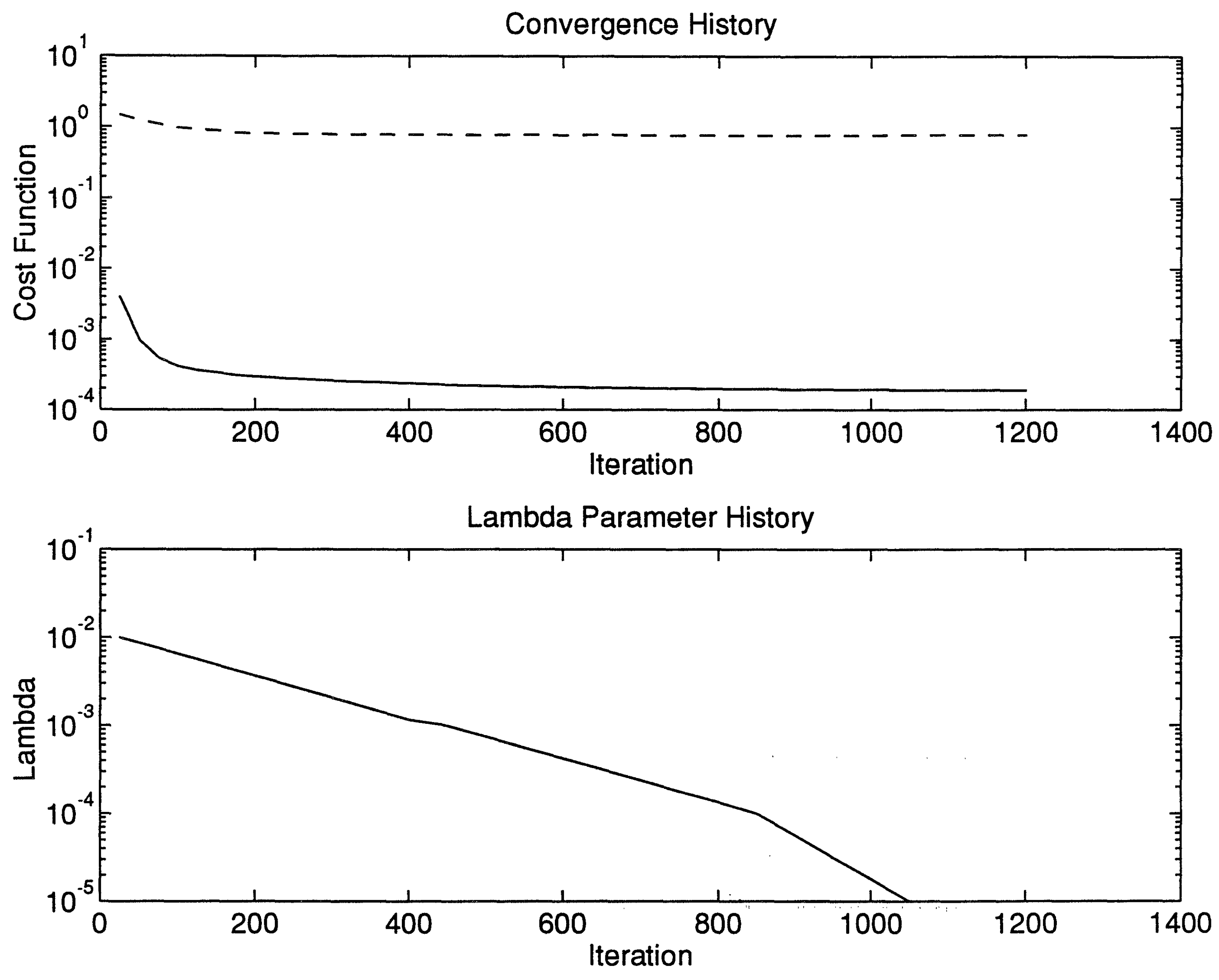
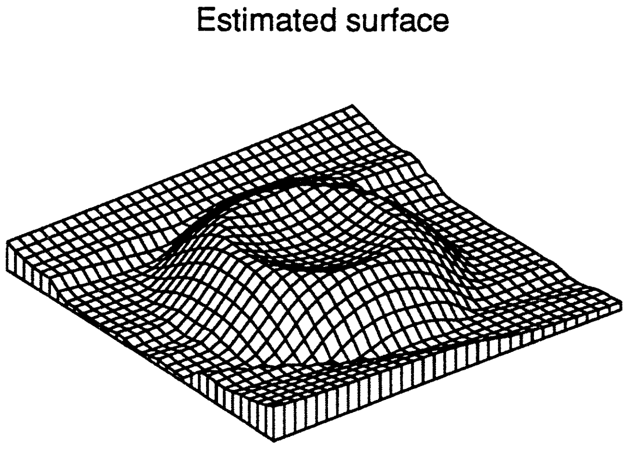


Figure 6-4: *z-only* DFSS iteration history on Test Crater Images. Graph shows the *z-only* DFSS algorithm performance on the test crater image pair. The upper graph shows the cost function(solid line) and RMS error(dashed line) of the estimated surface. The lower graph shows history of the smoothness weighting term λ



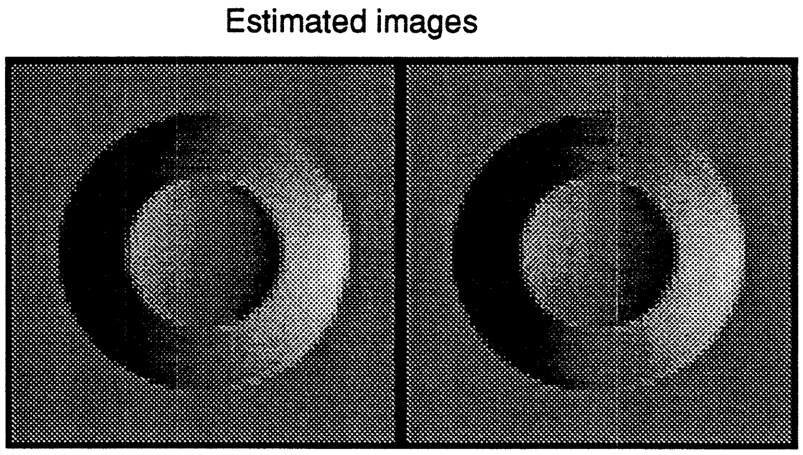


Figure 6-5: *z-only* DFSS Estimated Surface and Error on Test Crater Images at the end of iterations

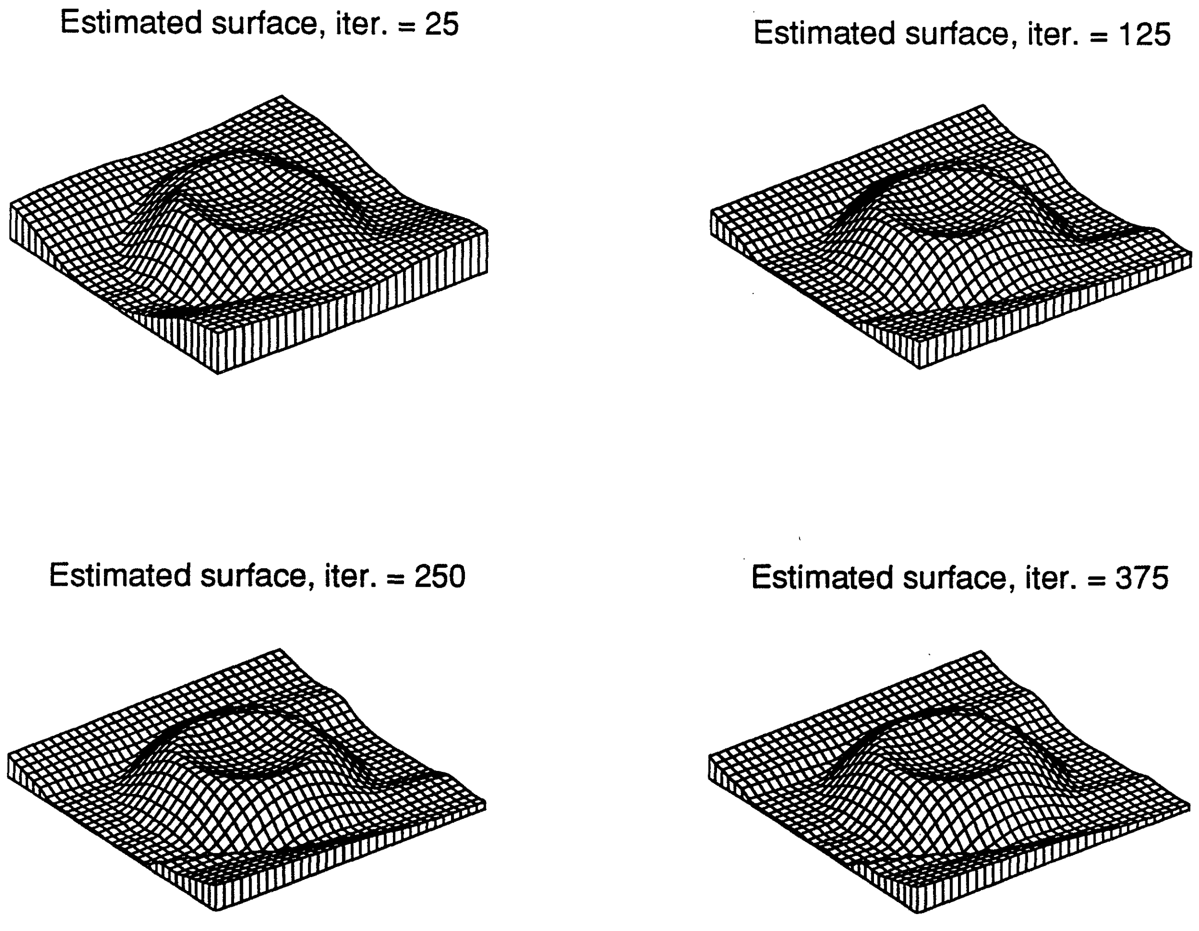

Estimated surface, iter. = 25
Estimated surface, iter. = 125
Estimated surface, iter. = 250
Estimated surface, iter. = 375

Figure 6-6: *z-only* DFSS estimation at various iteration steps on Test Crater Images

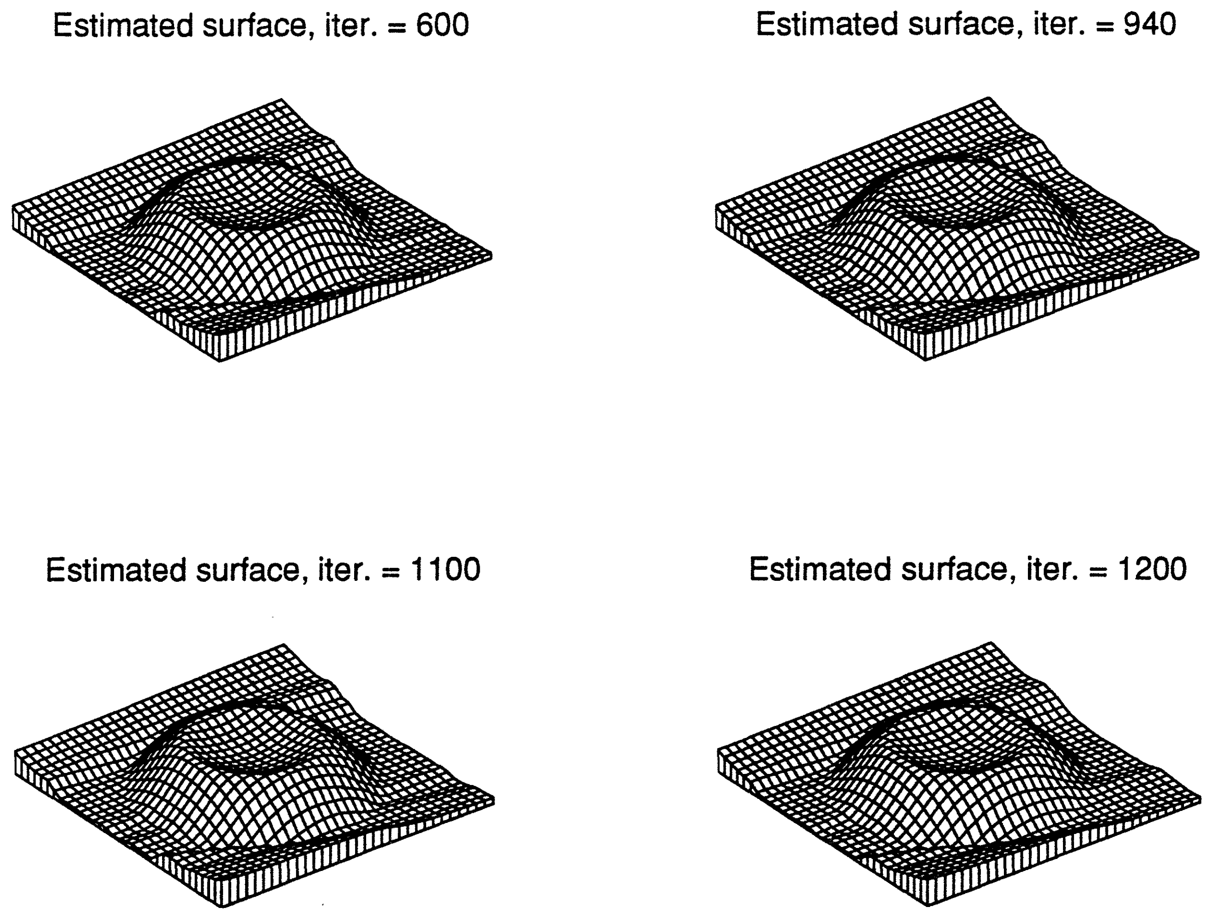

Estimated surface, iter. = 600
Estimated surface, iter. = 940
Estimated surface, iter. = 1100
Estimated surface, iter. = 1200

Figure 6-7: *z-only* DFSS estimation at various iteration steps on Test Crater Images

6.4 Comparison between DFSS results and that of Shape from Shading

The earlier two examples presented in chapter 5 prompt us two questions. The first one is that the two images are taken under similar lighting and camera position, the two images look alike except slight change in shading, therefore what kind result the Shape from Shading algorithm will provide? The second is that from photometric stereo, we know that the more different the lighting condition is, the better the result is. In this way, we can regard Shape from Shading as photometric stereo under exactly the same lighting condition. DFSS algorithm on the other hand has integrated photometric stereo into itself. So can DFSS algorithm produce better result?

These two questions are actually the same, they can be rephrased as "is DFSS algorithm working better than Shape from Shading in dealing the real images we used here, and why?". The answer is yes. The reason is that DFSS fused photometric stereo, binocular stereo and shape from shading. It can take advantage of the difference in shading and gradient in the image pairs. This lends another handle to constrain the estimated surface, particularly at the estimated of overall trend of the surface, where Shape from Shading is weak. Even though the differences are small in the two examples in this thesis. The benefit is obvious.

To show and prove the advantage of DFSS, we can go back to previous section, and run one image through Shape from Shading. The reason to do so is that those images pairs are synthetically generated according to the same lighting and geometry information of that of example 1 in chapter 5. This is better and easier than the real images because they are simple structure and we know the ground truth.

Clearly, Shape from shading closely estimated the surface. Comparing the estimated surface with that from DFSS, DFSS predicted a much flat background surface and the crater feature is better shown.

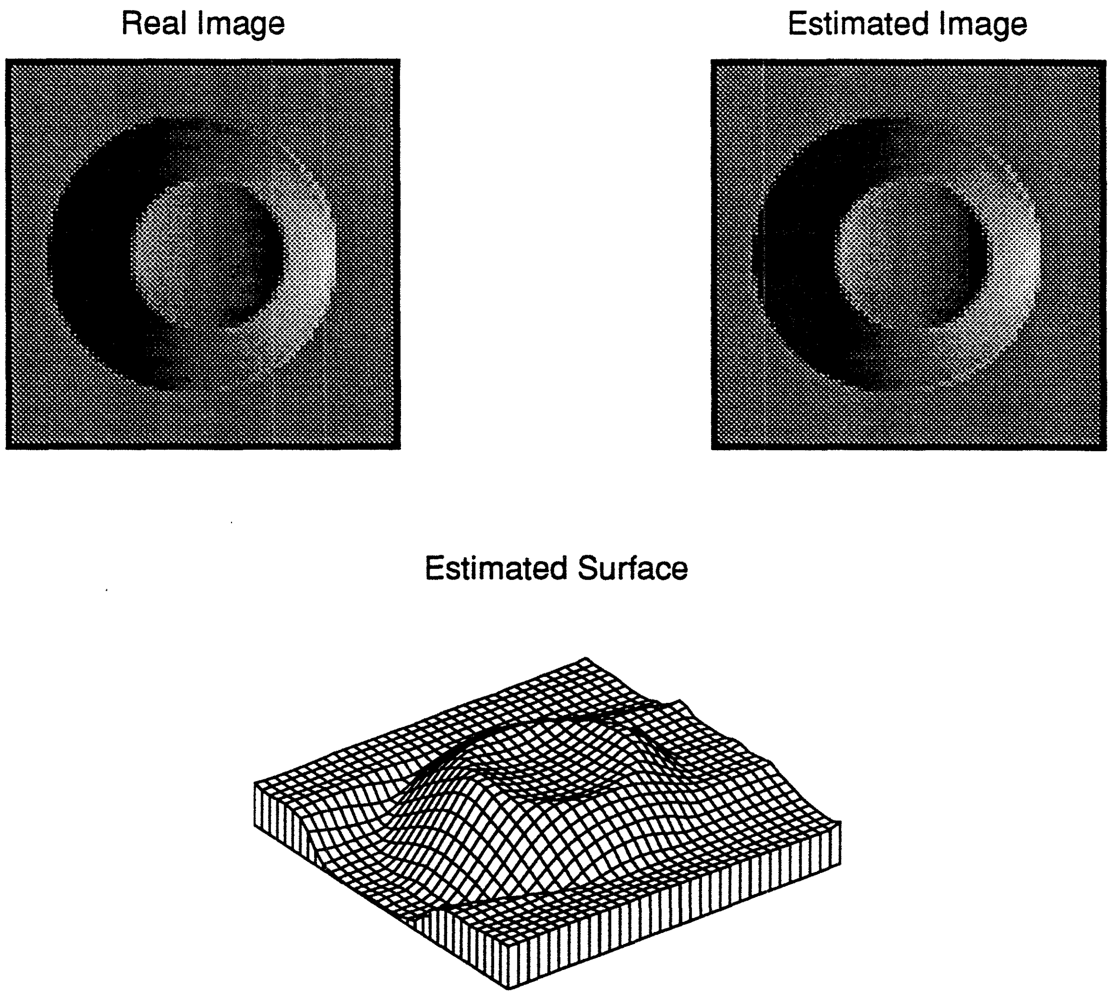


Figure 6-8: Test Crater Synthetic Image (left camera) and the estimated surface at the end of iteration.

Chapter 7

Discussion

7.1 Merits

$z - only$ algorithms demonstrate a much better performance using the two photoclinometry images that a simple shape from shading algorithm which uses only one image. This Performance increase validates the fusion approach to obtaining better performance vision algorithms.

The $z - only$ algorithm was also shown to be robust, it is able to accurately estimate the synthetic surface in the presence of several types of errors. The performance of the algorithm based on images that contains noise, geometry error, or reflectance errors was demonstrated in real Mars images. In which case, the algorithm is able to form a close estimate.

The $z - only$ DFSS algorithm shows very good performance on real Mars Viking image pairs. Especially it shows its performance under real measurement errors, calibrations error, distortion, and noise. It also shows its robustness under relatively weak shading information.

The C implementation of $z - only$ DFSS algorithm makes it more portable and efficient. Better Than three times the performance is obtained than its version in matlab. Large image(256x256) can now be processed on a SPARCstation.

7.2 limitations

There are many limitations in the Depth from Shape from Shading and Stereo algorithm.

Though $z-only$ algorithm is the most robust one among the proposed zpq, Dual-z and Disparity map. It does not handle varying albedo, nor does it handle very weak shading situation very well. Along these, there are some technical deficiencies, such as reflectance map depends on scalar p and q, rather than vector P and Q.

DFSS has strong bias towards Shape-from-Shading algorithm. Since it is variational method based, easy to integrate when shading information is weak, DFSS algorithm can get stuck in a local minimum, as seen in hard crater case. The stereo information presented in both the synthetic and real images are strong and can be used to pull the algorithm out of the local minimum. However since binocular stereo is feature based, makes it difficult to incorporate into the cost function.

In dealing with real images, it is found that DFSS algorithm performs not very satisfactory when lighting are coming from similar direction. Though this is not DFSS was developed for, it must also be able to cope with real situations.

The DFSS algorithm is based on simplified geometry, It works the best when the distance from the object to the two cameras are roughly equal. This is not always true or easily obtainable in the real world. This also contributes to the difference of the performance between the real and the synthetic images.

There is also an implementation issue. For 256 by 256 pixel image, the program runs roughly 20 hours to finish 1200 iteration with maximum number of hierarchical functions. To process full resolution images, 1204 by 1056 pixel, is not practical.

Since the thesis' main goal is to test $z - only$ DFSS algorithm on real Viking Images images. I am not going to discuss about many extensions possible to this algorithm. But any extension which may result significant memory and CPU time requirement increase, such as the write reflectance maps as a function of vector P and Q. Instead focus should be on improving optimization process.

Appendix A

DFSS Functions

In this Appendix, I will briefly explain all the DFSS core processing functions and their relationship. From these functions, one can build a program to process and analysis any images fit DFSS feature, as disscused in earlier chapters, by providing proper I/O functions.

The source code of the following functions and a makefile are made available on ftp.ai.mit.edu(128.52.32.11), in /pub/yan, along with a copy of this thesis.

The program loops over the conjugate gradient optimization function. In the conjugate gradient optimization calculation. It first calculates the cost based on current estimates of the surface. The gradient function returns the gradient associated with it. Base on these, a linear search is performed to find a new minimum, and the cost and gradient are then recalculated to use this new value. This continues until the predetermined number of iteration run out. During this process the hierarchical basis conversion functions are used to speed up calculation. Interpolation, filtering and convolution functions are used to assist the cost and gradient calculation. Reflection map based on current estimates are generated and compared with the actual reflection map to guide the convergence.

dfss.h	header file of DFSS functions
dfss.c	main DFSS analysis function
conjgrad.c	conjugate gradient calculation
cost.c	cost calculation
grad.c	gradient calculation
hbasis.c	hierarchical basis conversion
lsearch.c	linear search for conjugate gradient optimization
conv2.c	2-D convolution calculation
cfilter2d.c	2-D computational molecule filtering with bicubic interpolation
domain2d.c	2-D plaid domain generation
interpx.c	linear interpolation in one direction.
rmap.c	reflection map calculation based on current estimates.

Table A.1: DFSS function list

Appendix B

DFSS I/O Functions

In this Appendix, I present several example I/O and driver functions needed to build a DFSS program to analyze and process Viking Space project images. Since I used Matlab to display and plot the images and the estimated surface, loadmat.c and savemat.c are used to read and write Matlab files.

The source code and a makefile are made available on ftp.ai.mit.edu(128.52.32.11), in /pub/yan, along with a copy of this thesis.

```
/************************************************************************
 * FILE dfss_pds.c                                                      *
 *                                                                      *
 * Depth From Shape from Shading and Stereo Image Analysis Program      *
 * for PC, Vax, Unix and Macintosh systems.                             *
 *                                                                      *
 * it reads synthetic images in matlab file format and output           *
 * estimated topography in matlab file format.                          *
 ************************************************************************/

#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <strings.h>
```

```
#include <matrix.h>
#include "dfss.h"

void main(int argc, char **argv)
{
  char          name[20];
  int           mz, nz, type, mrows, ncols, imagf;
  real          *xi, *xr, *x;
  matrix        ztemp;

/* set up global variables */

  levels = 7;          /* level of hbasis */

  cntl1 = 1.0e-8;      /* Termination tolerance for X. */
  cntl2 = 1.0e-16;     /* Termination tolerance on F. */
  cntl3 = 1.0e-6;      /* Termination criterion on constraint violation. */
  cntl4 = 0;           /* Algorithm: Line Search Algorithm. */
  cntl5 = 0;           /* Function value. (Lambda in goal attainment) */
  cntl6 = 0;           /* Number of Function and Constraint Evaluations. */
  cntl7 = 0;           /* Number of Function Gradient Evaluations. */
  cntl8 = 0;           /* Number of Constraint Evaluations */
  cntl9 = 0;           /* Number of equality constraints. */
  cntl10 = 0;          /* Maximum number of iterations. default, 100 * no of var. */
  cntl11 = 1.0e-8;     /* Min. change in variables for finite diff. grad. */
  cntl12 = 0.1;        /* Max. change in variables for finite diff. grad. */
  cntl13 = 0.0;        /* Step length. (Default 1 or less) */
  cntl14 = 0.0;

/* get host information and input and output files */

  strcpy(inname1, " ");
  strcpy(outname, " ");
  if (argc == 1)
    usage();
  else if (argc == 2 && (strncmp(argv[1],"-help",5) == 0 ||
                   strncmp(argv[1],"-h",2) == 0 ||
                   strncmp(argv[1],"?",1)   == 0))
    usage();
  else {
    strcpy(inname1, argv[1]);
    if (argc == 3) strcpy(outname, argv[2]);
    if (argc > 3) usage();
  }

  host = check_host();
  get_files(host);

/* readin the image file */

  printf("\nReading in  the reflection maps ......\n");

  while (loadmat(infp1, &type, name, &mrows, &ncols, &imagf, &xr, &xi) == 0) {
    if  (strncmp(name, "E1", 2) == 0) {
         e1 = new_matrix(mrows, ncols);
         array_to_matrix(e1, mrows, ncols, xr);
    }
    if  (strncmp(name, "E2", 2) == 0) {
         e2 = new_matrix(mrows, ncols);
         array_to_matrix(e2, mrows, ncols, xr);
    }
    if  (strncmp(name, "z", 1) == 0) {
         mz = mrows; nz = ncols;
         ztemp = new_matrix(mrows, ncols);
         array_to_matrix(ztemp, mrows, ncols, xr);
    }
    if  (strncmp(name, "params", 6) == 0) {
            f = *xr; b = *(xr + 1); z0 = *(xr + 2);
            ps1 = *(xr + 3); qs1 = *(xr + 4); ps2 = *(xr + 5); qs2 = *(xr + 6);
            vx1 = *(xr + 7); vy1 = *(xr + 8); vx2 = *(xr + 9); vy2 = *(xr + 10);
            delta = *(xr + 11);
        }
    }

    ztrue = new_matrix(mz - 2, nz - 2);                /* strip off the boarder */
    matrix_copy_submatrix(ztemp, ztrue, 1, mz - 1, 1, nz - 1);
    free_matrix(ztemp);

    x = calloc((mz - 2) *(nz - 2), sizeof(double));
    matrix_to_array(ztrue, mz - 2, nz - 2, x);
    savemat(outfp, host, "ztrue", mz - 2, nz - 2, 0, x, xi);
    free(x);

    free_matrix(ztrue);

    z = new_matrix(mz - 2, nz - 2);                    /* allocate memory for z */
    matrix_set(z, z0);

/* start DFSS analysis */

    dfss();

/* output the final depth matrix */

    x = calloc(NOROWS(e1) * NOCOLS(e1), sizeof(double));
    matrix_to_array(e1, NOROWS(e1) , NOCOLS(e1), x);
    savemat(outfp, host, "e1", NOROWS(e1) , NOCOLS(e1), 0, x, xi);
    free(x);

    x = calloc(NOROWS(e2) * NOCOLS(e2), sizeof(double));
    matrix_to_array(e2, NOROWS(e2) , NOCOLS(e2), x);
```

```
  savemat(outfp, host, "e2", NOROWS(e2) , NOCOLS(e2), 0, x, xi);
  free(x);

  x = calloc((mz - 2)* (nz - 2), sizeof(double));
  matrix_to_array(z, mz - 2 , nz - 2, x);
  savemat(outfp, host, "z", mz - 2 , nz - 2, 0, x, xi);
  free(x);



/* clear up and close */

  free_matrix(z);
  free_matrix(ztrue);
  free_matrix(e1);
  free_matrix(e2);

  close(infp1);
  fclose(outfp);
}

/* subroutine get_files - get input, output filenames and open them */

void get_files(int host)
{
  if (inname1[0] == ' ') {
    printf("\nEnter name of the file to be DFSS analyzed:  ");
    gets (inname1);
  }
  if (host == 0 | host == 3000) {
    if ((infp1 = fopen(inname1, "rb")) == NULL) {
         printf("\ncan't open input file:  %s\n", inname1);
         exit(1);
    }
  }
  else if (host == 2000 | host == 1000) {
    if ((infp1 = fopen(inname1, "r")) == NULL) {
         printf("\ncan't open input file:  %s\n",inname1);
         exit(1);
    }
  }

/* get output file */

  if (outname[0] == ' ') {
    printf("\nEnter name of output file:  ");
    gets (outname);
  }
  if (host == 0 | host == 2000) {
    if ((outfp = fopen(outname,"wb")) == NULL) {
         printf("\ncan't open output file:  %s\n",outname);
         exit(1);
    }
  }
  else if (host == 1000) {
    if ((outfp = fopen(outname,"w")) == NULL) {
         printf("\ncan't open output file:  %s\n",outname);
         exit(1);
    }
  }
}

void usage(void)
{
  printf("z-only Depth From Shading and Stereo Program\n\n");
  printf("INPUT:\tsynthetic image in matlab file format.\n");
  printf("OUTPUT:\tresults in matlab file format\n");
  printf("\nCommand line format:\n\n");
  printf("dfss_mat [infile] [outfile] \n");
  printf("\tinfile\t - name of synthetic image file.\n");
  printf("\toutfile\t - name of DFSS output file.\n");
}
```

```
/************************************************************************
 * FILE gen_opt.c                                                        *
 *                                                                       *
 * make lambda and step table, used in optimization function            *
 ************************************************************************/


#include <stdio.h>
#include <math.h>
#include <matrix.h>
#include "dfss.h"

matrix gen_opt(real *lambda, int *step, int *nsteps, int tot)
{
  int           i, j;                    /* temp. counter */
  int           r_opt = 0;               /* count number of elements in opts */
  matrix        opt;                     /* lambda and step matrix */
  matrix        steps, a, b, c;          /* temp matrixs */

  opt = new_matrix(tot, 2);              /* allocate memory for opt matrix */
  for (i = 0; i < 5; i++) {
    steps = new_matrix(1, nsteps[i]);
    for (j = 0; j < nsteps[i]; j++) {
        elem_set(steps, 0, j, (real) j+1);
    }
    a = new_matrix(nsteps[i], 1);
    matrix_set(a, log(lambda[i]));
    matrix_scalar_sub(steps, 1.0, steps);
    b = new_matrix(nsteps[i], 1);
    matrix_transpose(steps, b);
```

```
        matrix_scalar_div(b, (real) nsteps[i], b);
        c = new_matrix(nsteps[i], 1);
        matrix_set(c, log(lambda[i+1])−log(lambda[i]));
        matrix_mul(b, c, c);
        matrix_add(a, c, c);

/* generate opt matrix */

        for (j = 0; j < nsteps[i]; j++) {
            elem_set(opt, r_opt + j, 0, exp(elem_ref(c, j, 0)));
            elem_set(opt, r_opt + j, 1, (real) step[i]);
        }
            r_opt += nsteps[i];
    }

/* clean up the temp matrix, release memory */

    free_matrix(steps);
    free_matrix(a);
    free_matrix(b);
    free_matrix(c);
    return(opt);
}
```

```
/*************************************************************************
 * FILE get_host.c                                                       *
 *                                                                       *
 * Check host computer type,                                             *
 * for PC, Vax, Unix and Macintosh systems.                              *
 *                                                                       *
 * it finds out the machine type and byte swap                           *
 *************************************************************************/
```

```
#include <stdio.h>
#include <strings.h>
#include <matrix.h>
#include "dfss.h"

int check_host()
{
  char          hostname[80];
  int           swap, host, bits, var;
  union {
    char  ichar[2];
    short ilen;
  } onion;

  if (sizeof(var) == 4) bits = 32;
  else bits = 16;

  onion.ichar[0] = 1;
  onion.ichar[1] = 0;

  if (onion.ilen == 1) swap = TRUE;
  else           swap = FALSE;

  if (bits == 16 && swap == TRUE) {
    host = 0; /* IBM PC host  */
    strcpy(hostname,
          "Host 1 - 16 bit integers with swapping, no var len support.");
  }

  if (bits == 16 && swap == FALSE) {
    host = 0; /* Non byte swapped 16 bit host  */
    strcpy(hostname,
          "Host 2 - 16 bit integers without swapping, no var len support.");
  }
  if (bits == 32 && swap == TRUE) {
    host = 2000; /* VAX host with var length support */
    strcpy(hostname,
            "Host 3 - 32 bit integers with swapping.");
  }

  if (bits == 32 && swap == FALSE) {
    host = 1000; /* OTHER 32-bit host, such as Sun */
    strcpy(hostname,
            "Host 5 - 32 bit integers without swapping, no var len support.");
  }

  return(host);
}
```

```
/*********************************************************************
 * FILE loadmat.                                                     *
 *                                                                   *
 * C language routine to load a matrix from a MAT-file.              *
 *********************************************************************/

#include <stdio.h>

typedef struct {
  long type;   /* type */
  long mrows;  /* row dimension */
  long ncols;  /* column dimension */
  long imagf;  /* flag indicating imag part */
  long namlen; /* name length (including NULL) */
} Fmatrix;

int loadmat(FILE *fp, int *type, char *pname, int *mrows, int *ncols,
              int *imagf, double **preal, double **pimag)
{
  char *malloc();
  Fmatrix x;
  int mn, namlen;

/* Get Fmatrix structure from file */

  if (fread((char *)&x, sizeof(Fmatrix), 1, fp) != 1) {
    return(1);
  }
  *type = x.type;
  *mrows = x.mrows;
        *ncols = x.ncols;
  *imagf = x.imagf;
  namlen = x.namlen;
  mn = x.mrows * x.ncols;

/* Get matrix name from file */

  if (fread(pname, sizeof(char), namlen, fp) != namlen) {
    return(1);
  }

/* Get Real part of matrix from file */

  if (!(*preal = (double *)malloc(mn*sizeof(double)))) {
    printf("\nError:  Variable too big to load\n");
    return(1);
  }
  if (fread(*preal, sizeof(double), mn, fp) != mn) {
    free(*preal);
    return(1);
  }
```

```
/* Get Imag part of matrix from file, if it exists */

  if (x.imagf) {
    if (!(*pimag = (double *)malloc(mn*sizeof(double)))) {
        printf("\nError:  Variable too big to load\n");
        free(*preal);
        return(1);
    }
    if (fread(*pimag, sizeof(double), mn, fp) != mn) {
        free(*pimag);
        free(*preal);
        return(1);
    }
  }
  return(0);
}
```

```
/**************************************************************************
 * FILE savemat.c                                                         *
 * C language routine to save a matrix in a MAT-file.                     *
 **************************************************************************/

#include <stdio.h>

typedef struct {
  long type;   /* type */
  long mrows;  /* row dimension */
  long ncols;  /* column dimension */
  long imagf;  /* flag indicating imag part */
  long namlen; /* name length (including NULL) */
} Fmatrix;
```

```
void savemat(FILE *fp, int type, char *pname, int mrows, int ncols,
                int imagf, double *preal, double *pimag)
{
  Fmatrix x;
  int mn;

  x.type = type;
  x.mrows = mrows;
  x.ncols = ncols;
  x.imagf = imagf;
  x.namlen = strlen(pname) + 1;
  mn = x.mrows * x.ncols;

  fwrite(&x, sizeof(Fmatrix), 1, fp);
  fwrite(pname, sizeof(char), (int)x.namlen, fp);
  fwrite(preal, sizeof(double), mn, fp);
  if (imagf) {
    fwrite(pimag, sizeof(double), mn, fp);
  }
}
```

Appendix C

The Matrix Library

C.1 Vectors and Matrices

In the following section, I provide some documentation on the functions available in the matrix library for doing vector and matrix manipulation. The documentation covers only the basic functions.

The entire library is written in the C programming language and has been running on Sparc workstations in the AI LAB at MIT, and PCs outside.

In what follows, the library functions deal exclusively with 2-dimensional matrices and 1-dimensional vectors and do NOT deal with multi-dimensional matrices.

Internally, matrices are represented as two dimensional matrices. A vector of dimension n is represented as a column vector or an matrix whose shape is $n \times 1$.

The entire library has been written using a `typedef` which defines the `real` data type to be the C `double` datatype. By redefining this datatype in `matrix.h` and recompiling it should be relatively easy to convert the library to single precision if needed. To use the library the file `matrix.h` must be included in your C source files. The library defines (typedefs) many data types the most used of which is the `matrix` data type.

The source code and a makefile are made available on ftp.ai.mit.edu(128.52.32.11), in /pub/yan, along with a copy of this thesis.

C.2 Creating and Freeing Matrices

The following function creates and returns an matrix all of whose elements are initialized to zero.

new_matrix *rows, cols* *Function*

int rows, cols;

This is the routine used internally by all the others. This creates and returns an matrix with the specified number of rows and columns.

free_matrix *a* *Function*

matrix a;

This routine performs the cleaning up, once you no longer need an matrix. It basically frees up the storage associated with the specified matrix *a*. Future references to a freed matrix will result in erroneous behaviour.

C.3 Accessing Elements within an Matrix

There are many functions to access elements within an matrix, the following two of which are very useful.

elem_set *a, i, j, val* *Function*

matrix a;

int i, j;

real val;

It sets element referred to by `a[i,j]` to the real `val`.

elem_ref *a, i, j* *Function*

matrix a;

int i, j;

This returns the element referred to by `a[i,j]`.

These two functions are implemented as C functions and hence may be slow for some applications. If speed is essential, users can use macros `felem_set` and `felem_ref` with identical arguments as those of `elem_set` and `elem_ref`.

C.4 Common Operations on Matrices

matrix_add *a, b, c* *Function*

matrix a, b, c;

This adds two matrices together, if they are of the same shape. If specified result matrix c is NULL then the results are stored in matrix b.

matrix_sub *a, b, c* *Function*

matrix a, b, c;

Subtracts matrix b from a, if they are of the same shape. If specified result matrix c is NULL then the results are stored in matrix b.

matrix_mul *a, b, c* *Function*

matrix a, b, c;

Multiples matrix b from a, if they are of the same shape. If specified result matrix c is NULL then the results are stored in matrix b. This does NOT do a matrix multiply. It does something like $c[i] = a[i] * b[i]$.

matrix_scalar_add *a, scalarvalue, b* *Function*

matrix a, b;

real scalarvalue;

This adds a scalar value to every element of the specified matrix a. If specified result matrix *b* is NULL, then the result is stored in matrix *a;*

matrix_scalar_sub *a, scalarvalue, b* *Function*

matrix a, b;

real scalarvalue;

This subtracts a scalar value from every element of the specified matrix a. If specified result matrix *b* is NULL, then the result is stored in matrix *a;*

matrix_scalar_mul *a, scalarvalue, b* *Function*

matrix a, b;

real scalarvalue;

This multiplies every element of the specified matrix a by the specified scalar value. If specified result matrix *b* is NULL, then the result is stored in matrix *a;*

matrix_scalar_div *a, scalarvalue, b* *Function*

matrix a, b;

real scalarvalue;

This divides every element of the specified matrix a by the specified scalar value. If specified result matrix *b* is NULL, then the result is stored in matrix *a;*

matrix_scalar_shift_add *a, scalar, b* *Function*

matrix a, b;

real scalarvalue;

This routine is like `matrix_scalar_add` only it shifts the matrix b by the specified amount instead of clobbering it with the result. In short doing a

```
        matrix_scalar_shift_add(a, 3.0, b);
```

is equivalent to writing (in pseudo-code)

```
        for all_elements_of b
           b[i] = b[i] + a[i] + 3.0;
```

Both matrices should be of the same shape.

matrix_scalar_shift_sub *a, scalar, b* *Function*

matrix a, b;

real scalarvalue;

This shifts the elements of matrix a, down instead of up as the add routine does.

matrix_scalar_shift_mul *a, scalar, b* *Function*

matrix a, b;

real scalarvalue;

This is equivalent to the following:

```
        for all_elements_of b
           b[i] = b[i] + a[i] * scalar;
```

matrix_scalar_shift_div *a, scalar, b* *Function*

matrix a, b;

real scalarvalue;

This is equivalent to the following:

```
        for all_elements_of b
           b[i] = b[i] + a[i] / scalar;
```

matrix_eq_internal *x,y,tol* *Function*

matrix x,y;

real tol;

This function checks if two matrices are equal on an element by element basis. The third argument specifies a tolerance value that is to be used in the comparison. It returns 1 if the two matrices are equal within the specified tolerance and 0 if not.

matrix_eq *x,y* *Function*

matrix x,y;

This is really equal to a function call to the previous function with a tolerance value of 0.00001.

matrix_norm_one *a* *Function*

matrix a;

Computes the maximum column sum of the specified matrix (also known as the 1-norm.

matrix_norm_infinity *a* *Function*

matrix a;

Computes the maximum row sum of the specified matrix. (also known as the inf-norm.

matrix_norm_frobenius *a* *Function*

matrix a;

Computes and returns the frobenius norm of the given matrix.

matrix_minmax *a, minval, maxval* *Function*

matrix a;

*real *minval;*

*real *maxval;*

This function computes and returns the maximum value and minimum value stored in the matrix in the result pointers passed in as the second and third arguments.

matrix_max *a,row,col* *Function*

register matrix a;

*int *row, *col;*

This function returns the maximum value in the matrix and sets the passed in `row` and `col` arguments to be the row and column index of the element that corresponds to this maximum value.

matrix_average *a* *Function*

register matrix a;

This function returns the average value of all the elements in an matrix.

matrix_multiply *a, b, c* *Function*

matrix a, b, c;

This is the normal matrix multiply routine. The result matrix *c* must be specified, and must be of the right dimensions.

matrix_multiply_new *a, b* *Function*

matrix a, b;

Returns a new matrix which is the result of matrix *a* post multiplied by *b*. It is upto the programmer to free the storage allocated to this matrix.

matrix_multiply_destructive *a, b* *Function*

matrix a, b;

This routine destructively modifies the matrix *b* to contain *a* post-multiplied by *b*.

matrix_invert *a, b* *Function*

matrix a, b; This is the familiar matrix inversion routine. Handles only square matrices now. The inverse of *a* is stored in *b*.

matrix_multiple_solve *a, b, c* *Function* *matrix a, b, c;*

This routine uses gaussian elimination with partial pivoting to solve the system of equations represented by $Ax = B$. The matrix A is the first argument `a`. The matrix B is passed in as the second argument. It can have many columns. The solution matrix is returned in `c` or `b` destructively if `c` is null.

matrix_identify *a* *Function*

matrix a;

Turns the specified matrix *a* into the identity matrix.

matrix_set *a, value* *Function*

matrix a;

real value;

Sets all elements of the matrix to be the specified value *value.* This function is useful with row or column vectors more than it is with matrices.

matrix_set_submatrix *a, value, startrow, endrow, startcol, endcol* *Function*

matrix a;

real value;

int startrow, endrow, starcol, endcol;

This routine allows the user to selective set a specified submatrix of an matrix to a value specified by *value.* The submatrix must be entirely contained in the matrix and will extend from row *startrow* to row *endrow* and starting from column *startcol* upto column *endcol.*

matrix_zero *a* *Function*

matrix a;

Zeros the elements of the specified matrix *a.*

matrix_zero_submatrix *a, startrow, endrow, startcol, endcol* *Function* *matrix a;*

int startrow, endrow, starcol, endcol;

This routine can be used to set a submatrix of the specified matrix to zero.

matrix_trace *a* *Function*

matrix a;

Computes and returns the trace of the matrix *a*.

matrix_copy *a, b* *Function*

matrix a, b;

This copies the specified matrix *a* into the matrix *b*. The matrices must be of the same shape.

matrix_copy_new *a* *Function*

matrix a;

Copies the matrix into a newly create matrix which has the appropriate shape and returns it.

matrix_copy_submatrix_new *a,startrow,endrow,startcol,endcol* *Function*

matrix a;

int startrow,endrow,startcol,endcol;

This function creates and returns a new matrix whose elements are an identical copy of the specified submatrix from the passed in matrix a.

matrix_copy_submatrix *a,b,startrow,endrow,startcol,endcol* *Function*

matrix a, b;

int startrow,endrow,startcol,endcol;

This is identical to the previous function only it destructively modifies the submatrix b as specified by the last four arguments to contain the elements of matrix a found in the specified submatrix.

matrix_transpose *a, b* *Function*

matrix a, b;

This transposes the matrix `a` into the matrix `b`. `b` must be of the appropriate shape.

matrix_transpose_new *a* *Function*

matrix a;

This creates and returns a new matrix which is the transpose of the argument matrix `a`. It is upto the programmer to free the allocated storage.

matrix_solve_linear *a, b, solution* *Function*

matrix a, b, solution;

This routine takes as input the matrix *a*, and the right hand side column vector in the specified matrix *b*. It solves the linear equation `Ax = b`, and returns the solution in the column vector *solution*. Uses the fast gaussian elimination with partial pivoting to find the solution if one exists.

matrix_row_interchange *a, row, row1, from, to* *Function*

matrix a;

int row, row1, from, to;

This interchanges the elements of the row *row* with the elements of the row *row1* starting from the column *from* upto the column specified by *to*. If *to* is specified to be less than 0, then the change will proceed upto the last column.

matrix_column_interchange *a, col, col1, from, to* *Function*

matrix a;

int col, col1, from, to;

This interchanges the elements of the column *col* with the elements of the column *col1* starting from the row *from* upto the row specified by *to*. If *to* is specified to be less than 0, then the change will proceed upto the last row.

matrix_gem *a, b* *Function*

matrix a,b;

This routine performs gaussian elimination on the input matrix a. RHS is passed in as the second argument matrix *b*. This routine is destructive and modifies the input matrix *a* for speed.

matrix_print *a* *Function*

matrix a;

This prints the matrix on the terminal. Doesn't do a very good job with formatting but serves its purpose.

matrix_pretty_print *header, a, fmt* *Function*

*register char *header;*

register matrix a;

*char *fmt;*

This function prints an matrix to stdout. If header is a non-NULL string, it will be printed first. If `fmt` is specified it will be used as the format specifier to be used while printing out each element in the matrix. Each row of the matrix will be printed out in one line.

matrix_input *a* *Function*

matrix a; Prompts the user for inputting an matrix whose shape is known.

matrix_get *str, a* *Function*

*char *str;*

matrix a;

This again prompts the user to input the elements of the matrix. If specified the `str` argument will be used as the prefix to be typed out as a prompt to the user while requesting him for each element of the matrix.

There are a number of functions that have been written to implement column vectors. These functions rely on the above mentioned matrix functions but are sometimes written to take advantage of the column vector structure of the vectors. The data type used in the functions below is still the same `matrix` data type as above, but to distinguish the vector functions that have been written to expect column vectors as input these functions have the string `vector` as part of their name instead of the string `matrix`.

new_column_vector *rows* *Function*

int rows;

Creates and returns a new column vector with the specified number of rows.

new_row_vector *columns* *Function*

int columns;

Creates and returns a new row vector with the specified number of columns.

new_vector *n* *Function*

int n;

Creates and returns a new column vector with the specified number of rows.

new_3_vector *Function*

Creates and returns a new column vector with 3 rows.

new_4_vector *Function*

Creates and returns a new column vector with 4 rows.

make_3_vector_from *x, y, z* *Function*

real x, y, z;

This creates and returns a 3 vector from the specified x, y and z values.

make_4_vector_from *x, y, z* *Function*

real x, y, z;

This creates and returns a 3 vector from the specified x, y and z values. The fourth element is set to 1.0.

vector_dot *a, b* *Function*

matrix a, b;

Takes two column vectors *a* and *b* and returns the dot product of the two vectors. (This is equivalent to taking the transpose of *a* and then calling `matrix_multiply` with the transposed vector and *b*, but is much faster).

matrix_dot *a, b* *Function*

matrix a, b;

This function is provided only so that you dont have to create a new matrix that is the transpose of a row or column vector and then call matrix_multiply with it – purely for speed and convenience reasons.

vector_magnitude *a* *Function*

matrix a;

This computes and returns the magnitude of a given vector *a*. (i.e. the value returned is the square root of the dot product of the vector with itself).

matrix_magnitude *a* *Function*

matrix a;

Same as the above function that works for both row and column vectors.

vector_angle_between *a, b* *Function*

matrix a,b;

This computes the angle between the two column vectors passed in *a* and *b*. (This uses the acos() function).

vector_normalize_new *a* *Function*

matrix a;

This routine creates and returns a new matrix that is a column vector whose elements are the elements of the input vector *a* normalized by its magnitude.

vector_normalize *a* *Function*

matrix a;

This routine destructively normalizes the input column vector *a*.

vector_cross_3 *a, b, c* *Function*

matrix a,b,c;

This routine computes the cross product of the column vectors *a* and *b* which are assumed to be 3-vectors and stores the results in the column vector *c*.

vector_cross_3_new *a, b* *Function*

matrix a,b;

This routine creates and returns a new 3-vector that is the cross product of the two specified column vectors *a* and *b*.

There are a few functions provided to save and restore matrices from files.

read_matrix_from_file *a, fp* *Function*

matrix a;

*FILE *fp;*

This function assumes that the file has already been opened for reading, and we are at the right point to begin reading in the matrix It expects the matrix to be stored in ascii as one would expect rows by colums like the 2 by 4 example shown below.

```
0.8 0.8 0.8 0.2
0.2 0.1 0.0 0.3
```

make_matrix_from_stream *fp* *Function*

*FILE *fp;*

This function reads, creates and returns an matrix that was read from the file specified by the file pointer `fp`. Comment characters like `:`, `#` or `%` appearing at the beginning of a line cause the entire line to be omitted.

save_matrix *fp, str, a* *Function*

*FILE *fp;*

*char *str;*

matrix a;

This function takes a file pointer and writes out the given matrix to the file in the format so that it can be read in by the previous function. If the second argument is non-null, it will be a string that is written in a line above the matrix.