

A MOTION VISION SYSTEM FOR A MARTIAN MICRO-ROVER

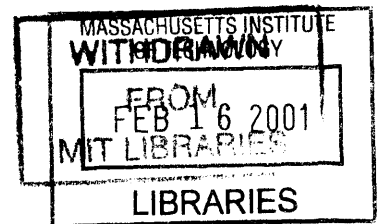
by

Stephen William Lynn

Submitted to the
**DEPARTMENT OF ELECTRICAL ENGINEERING
AND COMPUTER SCIENCE**
in partial fulfillment of the requirements
for the degree of
MASTER OF SCIENCE IN ELECTRICAL ENGINEERING
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
February, 1994

© Stephen William Lynn, 1994

ENG



Signature of Author _____
Department of Electrical Engineering and Computer Science
January 14, 1994

Certified by _____
W. Eric L. Grimson
Thesis Supervisor

Certified by _____
David S. Kang
Project Supervisor

Accepted by _____
F. R. Morgenthaler
Chair, Department Committee on Graduate Students

A MOTION VISION SYSTEM FOR A MARTIAN MICRO-ROVER

by

Stephen William Lynn

Submitted to the Department of Electrical Engineering and Computer Science on January 14, 1994 in partial fulfillment of the requirements for the Degree of Master of Science in Electrical Engineering.

Abstract

This thesis describes a monocular, motion vision system that was designed to meet the size, power, and processing constraints of a small, robotic, planetary explorer. The computational requirements of the brightness gradient approach, which is commonly used in motion vision systems, were found to be too great for the Micro-Rover. Instead, a pattern matching approach was proposed because most of the computations can be done in real-time in hardware, leaving significantly less computations for a microprocessor.

The results of processing a pair of images using pattern matching is an optical flow pattern, which is also a good representation of the motion field. As with most optical flow patterns, there are some incorrect flow vectors in the optical flow patterns generated from pattern matching. Various methods of weighting the flow vectors to enhance the correct vectors and attenuate the incorrect vectors were explored, and a suitable weight was found.

The results show that this vision system can assist the Micro-Rover in both navigation and hazard avoidance. The motion estimates, obtained from a Least-Squares fit of the optical flow pattern to the motion field, were within 10% of the actual values. Additionally, obstacle maps, which show the locations of potential hazards on the ground plane, were generated from a sequence of images.

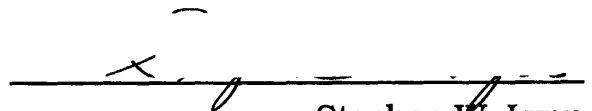
Thesis Advisor: W. Eric L. Grimson
Professor of Electrical Engineering

Project Advisor: David S. Kang
Member of Technical Staff at Draper Laboratory

Assignment of Copyright to Draper Laboratory

This thesis was supported by The Charles Stark Draper Laboratory, Inc. through the Lunar/Mars Micro-Rover Project. Publication of this thesis does not constitute approval by The Charles Stark Draper Laboratory, Inc. of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

I hereby assign my copyright of this thesis to The Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts.


Stephen W. Lynn

Permission is hereby granted by The Charles Stark Draper Laboratory, Inc. to the Massachusetts Institute of Technology to reproduce part and/or all of this thesis.

Acknowledgments

I would like to thank all the students that made this Micro-Rover project possible and wish them success in their studies here at M.I.T. and in their future careers. The names on the PROBE Lab Honor Roll are Seán Adam, Shane Farritor, Stefan Feldgoise, Matt Fredette, John Gilbert, Bill Kaliardos, Rob Kim, Brenda Kraft, Giang Lam, Kent Lietzau, Anthony Lorusso, Eric Malafeew, and Kimball Thurston. I would also like to thank Ashok Patel and Richard Regueiro for the anti-rover humor they brought to our office. Dave Kang is the leader of this infamous pack of hoodlums and deserves many thanks for opening the doors of Draper to me and allowing me to explore the world of robot vision.

I would like to thank two professors who helped me through this project - Berthold Horn and Eric Grimson. Professor Horn taught the Robot Vision class and worked with me in the initial stages of this project. Professor Grimson took over after Professor Horn left for sabbatical and offered good, to-the-point advice to help me put the project in perspective.

I would like to thank the Charles Stark Draper Laboratory, the National Science Foundation, and the Department of Electrical Engineering and Computer Science for their generous financial support of my studies.

I would like to thank all the friends that kept me from going crazy during my time at M.I.T. and while I was working on this thesis. I don't have enough room to mention all of them, but there are a few that I must mention. Augustine Fou, an all-around classy guy, was always there to assist me in my duties as THRA President and to prod me for not being civilized. Carlos Cabrera, a good friend, put up with all the shit I gave him and then still went out and partied with me (after I twisted his arm). Michelle Hoo Fatt erased Friday nights from her calendar so that we could go dancing at the Cask. My good friend back in Philly, Clyde Torrence, kept me up to date on what was going on back home (mostly about his cars) and was ready to party when I came back to town. Rich Orsini sometimes partied with us (although never at Pin-Ups, Gail), so that both of us could bust on Clyde. Ubaid Aktar was always ready to talk about the good old days back at Central High or to beat me in tennis.

I would like to thank my brother for calling me so many times during my 3 years at M.I.T., even when I neglected to call him. And, when I went back home, Ken was always ready to go to Fizz or to get his ass kicked in golf. It's too bad that he has to suffer another year in Happy Valley!

I would like to thank my parents for all the support they have given me during my 3 years at M.I.T. Being fortunate enough to have parents as wonderful as they are can only be a blessing from God!

Biography

Stephen William Lynn was born on April 2, 1969, in Abington, PA, just outside of Philadelphia. He grew up in Philadelphia and attended the James Russell Lowell Elementary School and the Julia R. Masterman Junior High School before attending Central High School. He earned a Bachelor of Arts Degree in 1987 at Central, which is the 2nd oldest public high school in the nation, and graduated as valedictorian of the 246th graduating class. Also at Central, he played doubles on the varsity tennis team that won the Philadelphia Public League Championship in 1987.

At the University of Pennsylvania, Stephen was enrolled in the Management and Technology Program, which he entered at the end of his sophomore year. In this program, he simultaneously pursued degrees in engineering and business. His majors were Electrical Engineering and Finance. In 1991, Stephen earned a Bachelor of Science in Engineering from the Moore School and a Bachelor of Science in Economics from the Wharton School. At graduation, he was awarded the A. Atwater Kent Prize for the most promising Electrical Engineer and the Delta Sigma Pi Scholarship Key in business. Stephen is a member of Tau Beta Pi, Eta Kappa Nu, and Beta Gamma Sigma honor societies.

For the past 2-1/2 years, Stephen has been pursuing an elusive Master's degree in Electrical Engineering at M.I.T. He has been working on the Micro-Rover project at the Charles Stark Draper Laboratory for 1-1/2 years. Stephen has been involved with the Tang Hall Residents' Association for the past 2 years as Common Rooms Officer and then President. He has also been involved with Project Awareness, a crime prevention group that works with Campus Police.

When he is not studying or working on his thesis, Stephen enjoys tennis, golf, bicycling, mangling his guitar, listening to music, reading books, and staying abreast of current events and business trends.

Dedication

This thesis is dedicated to the memory of my grandparents who have given me so much.

***Ida D. Lynn
Joseph M. Lynn
Emil Ziegele
Maria Ziegele***

Table of Contents

Chapters		Page
1	Introduction to Micro-Rovers	15
2	Introduction to Vision Systems	23
3	Testing the Pattern Matching	55
4	Motion Estimation	97
5	Obstacle Recognition	125
6	Conclusion	143
 Appendices		
A	The Code That Performs the Matchings	151
B	The Code That Calculates Motion	167
C	The Code That Generates Obstacle Maps	187
D	Maximum Rotation Between Images	195
E	Binocular Stereo Using Pattern Matching	199
References		201

Chapter 1 Introduction to Micro-Rovers

Micro-Rovers are small robotic explorers that will someday traverse the terrain of Mars. They are semi-autonomous, meaning that they will receive a destination from mission control here on earth and use their own abilities to reach that location safely. The scientific data and pictures that they send back will allow us to learn more about the environment of Mars.

A group of Massachusetts Institute of Technology students working at Draper Laboratory in Cambridge, Massachusetts is building several Micro-Rover prototypes. They are part of the PROBE lab, which stands for Planetary ROver Baseline Experiment. The team is comprised of both graduate and undergraduate students. The aim of the PROBE lab is not to develop a fully functional Micro-Rover for a future mission to Mars, but to make significant developments that can be incorporated on the actual mission rover.

The Micro-Rovers are smaller in size than previous planetary rovers. The Micro-Rovers measure about two-thirds of a meter long, a third of a meter wide, and a quarter of a meter high. A Micro-Rover weighs about 10 kilograms. Figure 1.1 shows one of the Micro-Rovers built at Draper Laboratory.

The mechanical structure of the Micro-Rovers consists of 3 platforms interconnected by spring steel. Each of the 3 platforms supports some of the electronic circuitry that runs the rover. The platforms are connected by two pieces of spring steel, which allow maximum flexibility while still maintaining structural integrity. Two wheels are mounted on each of the platforms.

The Micro-Rover is driven by small, DC motors located inside each of the 6 wheels. Placing a motor in each wheel allows maximum drive power because all the wheels help to move the vehicle. The DC motors are geared down to provide more hill-climbing torque and to reduce the no-load speed to a reasonable value for a small vehicle such as a Micro-

Rover. The speed of each motor is independently controlled so that when the Micro-Rover is navigating a turn, the outer wheels can move faster than the inner wheels.

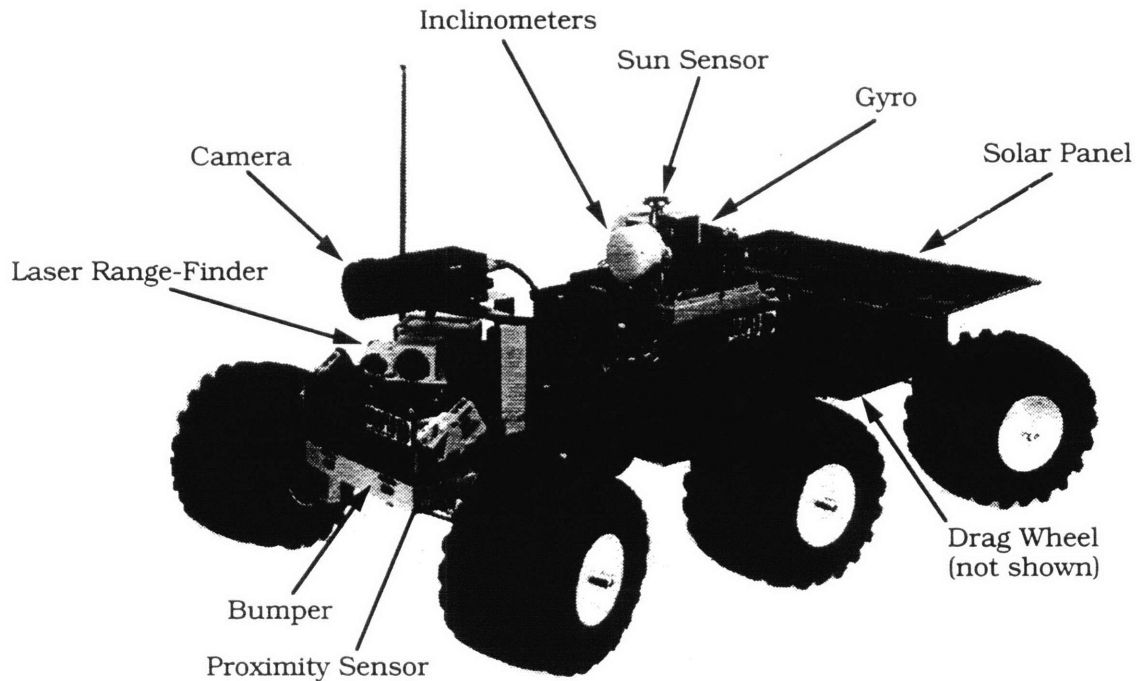


Figure 1.1 Picture of Mity2 Micro-Rover

The Micro-Rover is controlled by a main computer that executes the control code. The main computer collects the sensor data and steers the Micro-Rover toward its destination while avoiding obstacles reported by the sensors. The control computer functions semi-autonomously; mission control sends a destination to the control computer, which then acts independently to achieve that goal.

Sensors are integrated into the Micro-Rover to provide various information to the control computer. A laser ranging system is mounted on the front of the vehicle to provide distances to obstacles. Inclinometers are mounted on the center platform to provide pitch and roll data. A sun sensor is also mounted on the center platform to provide heading information to the control computer. In addition, a gyro is mounted on the center platform to integrate the turning of the Micro-

Rover and also provide heading information. Various low-level bump sensors are mounted on the Micro-Rover to provide high-priority obstacle information.

The laser ranging system illuminates objects in front of the Micro-Rover and detects the reflection of the laser beam to triangulate a distance to the object. A laser diode radiates a collimated beam of light that strikes objects in front of the Micro-Rover. The light is scattered in all directions and is picked up by a position sensitive detector (PSD) after passing through an optical filter and a lens. The PSD is a resistive network from which the location of the incident laser light can be calculated via the output currents. The position of the incident light and the focal length of the lens are used to calculate an angle to the object. This angle and the distance from the PSD to the laser diode are used to obtain the distance to the object.

The sun sensor detects the position of the sun in the sky and uses this position to indicate the heading of the Micro-Rover. A fish-eye lens is used to capture the sky with a 180° field of view. The sun is then focused on a PSD, and the (x,y) currents are used to find the angle to the sun relative to the Micro-Rover. Since the sun moves very slowly, its position can be used as an indicator of heading as long as periodic corrections are made for its motion.

A gyro is used as another indicator of heading which becomes useful when the sky is cloudy. The gyro spins and detects rotation of the Micro-Rover. The signal relating to the turning is a voltage, which is integrated to obtain heading information. Gyros typically have some drift which causes the error in the integrated heading information to grow over time.

A drag wheel is used to measure the distance traveled by the Micro-Rover. An optical detector/emitter pair passes light through little holes in a disk attached to the drag wheel and signals when light is detected. The on/off signaling is counted and converted from rotation of the drag wheel to distance traveled by the Micro-Rover.

A battery pack supplies power to the Micro-Rover to run the processors and the sensors, as well as the drive motors. The battery pack would contain enough energy to power the Micro-Rover throughout the day. Secondary batteries are being considered as the power source to allow for practically indefinite mission durations when combined with solar arrays for recharging.

Solar arrays integrated into the Micro-Rover's housing will recharge the battery pack throughout the day. Primary batteries, which contain a finite amount of energy and cannot be recharged, were thought to be too limiting. The utility of a Micro-Rover lies in its versatility and flexibility, so secondary batteries recharged through solar arrays were accepted as the preferred method of powering the Micro-Rover. The solar arrays will replenish the energy removed from the battery pack throughout the day, thus extending the mission duration indefinitely.

A Vision System as a Sensor

The aim of this project is to develop a motion vision system that can be used as a sensor and implemented on the Micro-Rover. It is desired to use the camera that we already have on-board the Micro-Rover to estimate its motion and recognize obstacles. The motion estimates could be used in the dead-reckoning navigation system, and the obstacles flagged by the vision system could be fused with hazard information from other sensors.

A dead-reckoning navigation system integrates periodic, incremental motions to track the position of the vehicle. This is opposed to a fixed-beacon navigation system, where the position of the vehicle is estimated relative to a fixed, external source. A vision system can provide periodic updates to the dead-reckoning system by estimating the translation and rotation of the Micro-Rover between images.

The translation and rotation estimates provided by a motion vision system will allow it to replace the sun sensor, gyro, inclinometers, and drag wheel. The sun sensor, gyro, and inclinometers provide pitch, roll,

and yaw information which can be obtained from the vision system by summing the rotation estimations. The drag wheel only provides the absolute distance moved by the Micro-Rover, but the motion vision system can provide not only the distance, but also the translation vector.

The hazard avoidance system can use the information contained in an obstacle map generated by a motion vision system. An obstacle map indicates the locations of obstacles on a 2-dimensional map of the ground plane. This map can be used in conjunction with other sensors to allow the Micro-Rover to plan a path around potential hazards.

The only requirement for a motion vision system is some measurement of the environment to eliminate the scale factor ambiguity problem. Since a camera can only measure the angle to an object and not the longitudinal and lateral distances, it is impossible to differentiate one motion from a motion of twice the magnitude in a world twice as large. Therefore, some measure needs to be taken - either the absolute translation of the Micro-Rover, obtained from the drag wheel, or the depth to a scene point, obtained from the laser ranging system. The latter is preferred since the laser ranging system is integral to the Micro-Rover while the drag wheel is expendable.

The requirements of the motion vision system are that it produce motion estimates that are reasonably accurate and obstacle maps that can be used to avoid hazards. The proposed Micro-Rover mission, as outlined by the Jet Propulsion Laboratory in Pasadena, CA, requires a navigational accuracy of 10%, i.e. the error in its position should be at most 10% of the distance traveled. Using the obstacle maps to avoid hazards require that almost all hazards are recognized and very few false alarms are generated.

Thesis Outline

Chapter 2 examines a common method of implementing a motion vision system, for which the computational requirements are too high, and proposes an alternate method, which relies upon hardware to do most of

the processing and, as a result, requires significantly less computations. The first method uses the brightness constraint equation and requires taking brightness derivatives at each pixel in the image. The computational requirements of this method, however, prohibit its use on a small, low-power vehicle, such as the Micro-Rover. An alternate method which relies upon matching patterns between successive images is proposed. The matches between successive images produce an optical flow pattern. The optical flow is assumed to represent the motion field, and the Least-Squares equations to estimate the motion from the motion field are derived.

In Chapter 3, various pairs of successive images are tested to determine if pattern matching can produce reasonable optical flows. The results show that the optical flow patterns are pretty good, but there are some incorrect flow vectors. Several weighting schemes to enhance correct flow vectors and eliminate incorrect flow vectors are examined and one is selected as the best weighting scheme.

The Least-Squares equations for the motion of the Micro-Rover are examined in Chapter 4. They are tested on both a planar and a contoured scene. It is found that the narrow field of view of the camera makes it impossible to distinguish between lateral rotation and rotation. The 6 degree of freedom model for the motion is scrapped in favor of a 4 degree of freedom model that eliminates the lateral motions. The Least-Squares equations for the 4 degree of freedom model are tested, and the correct motions are found. Finally, a sensitivity analysis is performed on the motion estimates from a sequence of images.

Chapter 5 analyzes the use of the depth maps, which are a by-product of the motion estimation, as an aid in hazard avoidance. The scene points represented by the depth values are translated onto a ground plane to mark the locations of obstacles. These points are represented as regions on the ground plane because there is some uncertainty as to their actual locations. Obstacle maps from a sequence of images are then fused together to enhance obstacles and eliminate noise.

Finally, Chapter 6 concludes this thesis by presenting the benefits and short-comings of the pattern matching approach to motion vision. The use of such a system on a Micro-Rover to estimate motion and avoid obstacles is analyzed. Weakness that need to be examined and improvements that need to be made are discussed.

Chapter 2 Introduction to Vision Systems

This chapter introduces the reader to vision systems and various components that make up a vision system. The mathematical structure of the environment and the image plane is introduced, and a convenient coordinate system for transformation between world and image coordinates is presented. The motion field produced by a relative motion between the scene and the camera is discussed. Optical flow, which is movement of the brightness values in an image, is presented. The CCD camera, which is common to most vision systems, is discussed. The common method of implementing a motion vision system, using the brightness constraint equation, is discussed, and its computational requirements are analyzed. An alternate method, using a constant pattern equation, is presented, and the relevant Least-Squares motion equations are derived.

Imaging the Environment

The first task in implementing a vision system is to image the environment. The scene, or environment, consists of various objects in a three-dimensional space. For the Micro-Rover, the scene will consist of the surface of Mars and various rocks and craters.

The three-dimensional environment is imaged onto a two-dimensional image plane through a point called the center of projection. The center of projection is like the aperture in a pinhole camera. Light rays from various scene points pass through the center of projection and irradiate the image plane, forming an image as shown in Figure 2.1.

There are various components that describe the image plane as shown in Figure 2.2. The optical axis is perpendicular to the image plane and passes through the center of projection. The intersection of the optical axis with the image plane is the principal point. The distance from the center of projection to the principal point is the principal distance.

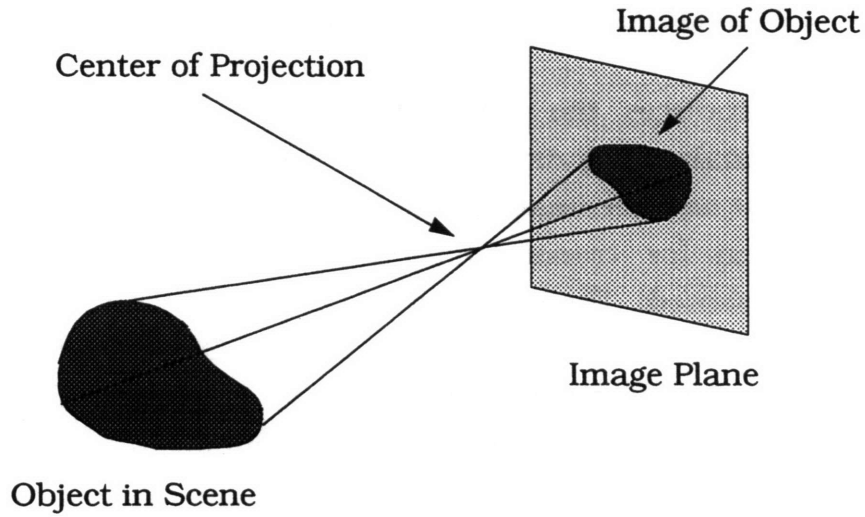


Figure 2.1 Object in Scene and Image of Object

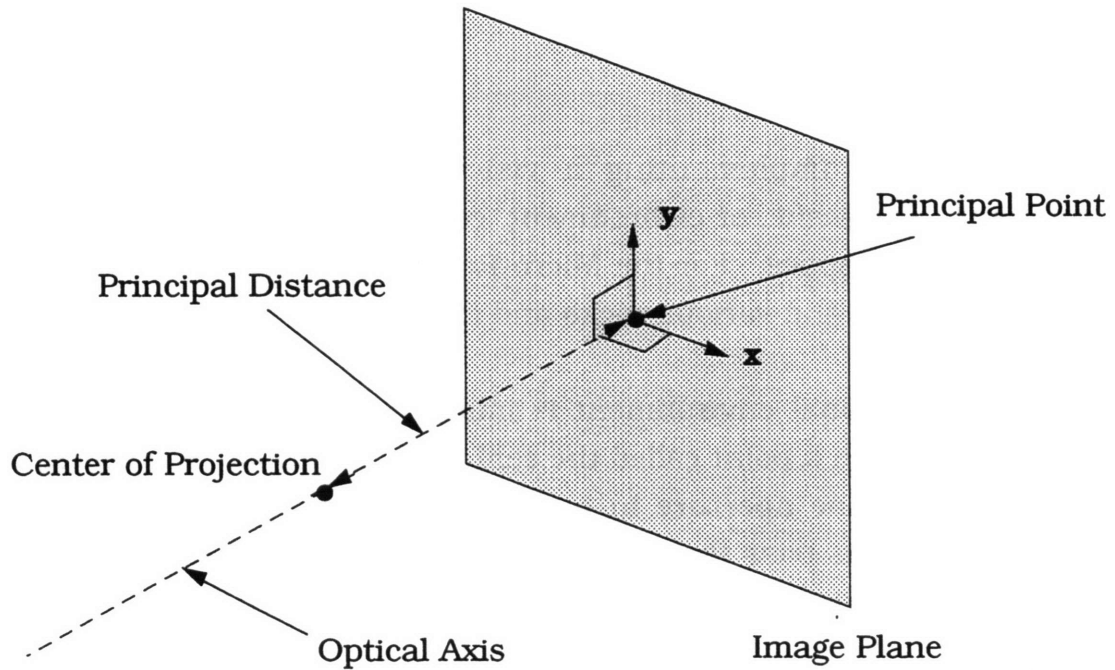


Figure 2.2 Parameters Describing the Image Plane

The image plane and the scene being imaged are on opposite sides of the center of projection, however the image plane can be reflected through the center of projection to define its coordinate system. The scene point, \mathbf{P} , the image point, \mathbf{p} , and the center of projection lie on a line. This

coordinate system allows easy transformation between world and image coordinates since they are proportional. Figure 2.3 shows the two coordinate systems.

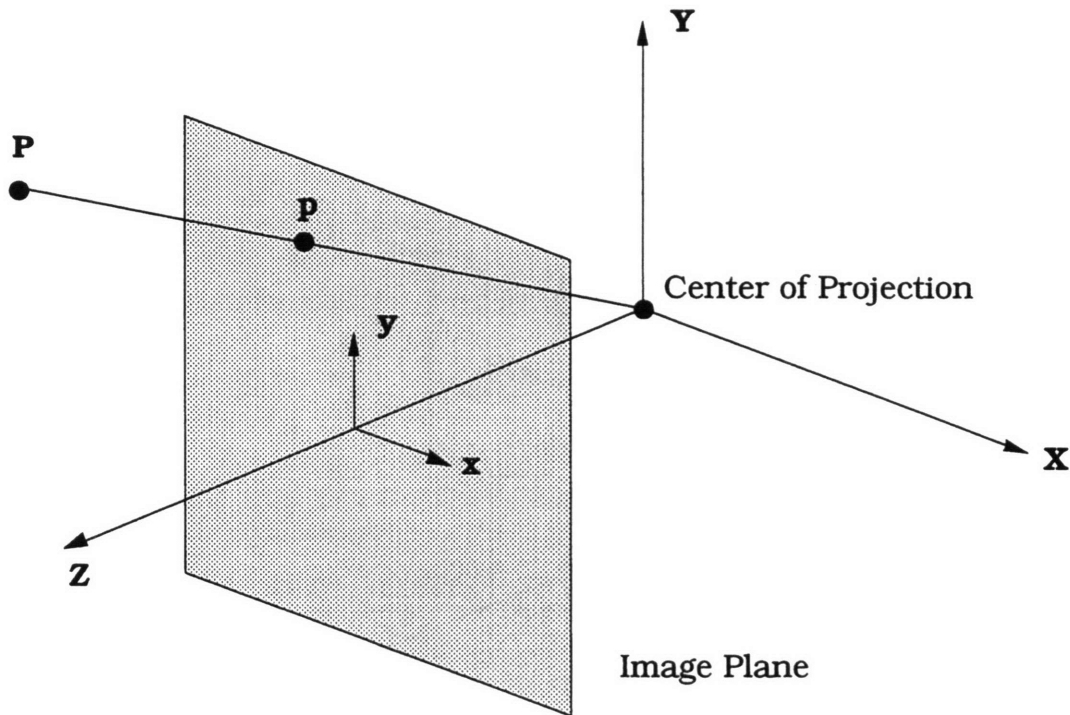


Figure 2.3 World and Image Coordinate Systems

Relative Motion Between the Camera and the Scene

The image will change when the position of the camera changes relative to the scene. This occurs when either objects in the scene move or the camera moves. A change in relative position causes the object's world coordinates as defined in Figure 2.3 to change. The light rays from the object pass through the center of projection at a different angle and produce an image of the object at a different location on the image plane.

Relative motion can be described by a two-dimensional motion field in the image plane. The scene is assumed to be static, i.e. there are no moving objects in the scene. It does not matter whether the camera moves or the scene moves; only relative motion is relevant.

The translation and rotation of the camera will change the world coordinate system since it is defined by the center of projection and optical axis of the camera. The center of projection is the origin of the world coordinate system, and the Z-axis is parallel to the optical axis. Translation of the camera causes translation of the origin, and rotation of the camera causes rotation of the axes. The translation vector and the directions of rotation about the axes are shown in Figure 2.4.

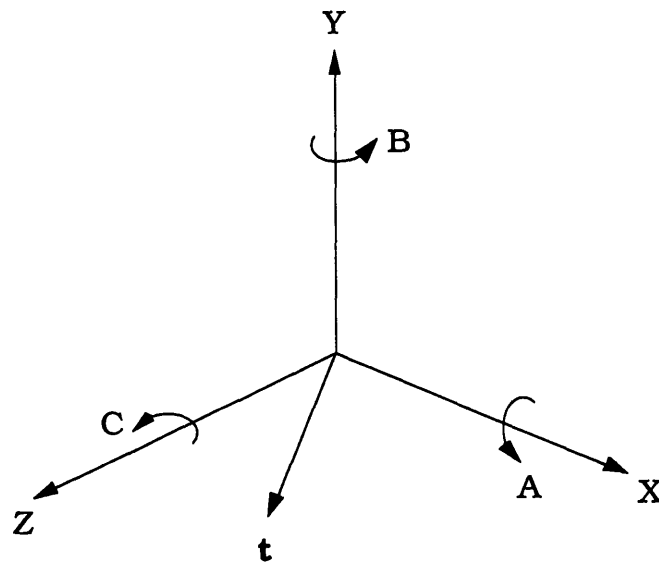


Figure 2.4 Directions of Rotation about the Axes

Translation is described by a vector, $\mathbf{t} = [U \ V \ W]^T$, with each component representing a motion along an axis. The new world coordinates of each scene point are obtained by subtracting the translation vector from the old world coordinates.

Rotation is also described by a vector, $\mathbf{p} = [A \ B \ C]^T$, representing rotation about each axis. The components of the rotation vector can be placed into a 3 x 3 matrix which multiplies the old world coordinates.

The new world coordinates are related to the old world coordinates through the following equation:

$$\begin{bmatrix} X' \\ Y' \\ Z' \end{bmatrix} = \mathbf{CBA} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} - \begin{bmatrix} U \\ V \\ W \end{bmatrix} = \mathbf{P} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} - \begin{bmatrix} U \\ V \\ W \end{bmatrix}$$

where the 3 matrices contain the rotation angles:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos A & \sin A \\ 0 & -\sin A & \cos A \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} \cos B & 0 & -\sin B \\ 0 & 1 & 0 \\ \sin B & 0 & \cos B \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} \cos C & \sin C & 0 \\ -\sin C & \cos C & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The order of rotation is very important in backing out the rotation angles because the determinant of the \mathbf{P} matrix is zero. Only 2 of the 3 rows in the \mathbf{P} matrix are independent. The first 2 rows determine the new x and y coordinates. However, the z coordinate is not determined by the third row, but from the distance of the point from the origin. The distance does not change through a rotation, only the individual coordinates change. In this case the point is rotated about the x-axis, y-axis, and z-axis, in that order.

The \mathbf{P} matrix obtained by multiplying the \mathbf{C} , \mathbf{B} , and \mathbf{A} matrices is:

$$\mathbf{P} = \begin{bmatrix} \cos B \cos C & \sin A \sin B \cos C + \cos A \sin C & -\cos A \sin B \cos C + \sin A \sin C \\ -\cos B \sin C & -\sin A \sin B \sin C + \cos A \cos C & \cos A \sin B \sin C + \sin A \cos C \\ \sin B & -\sin A \cos B & \cos A \cos B \end{bmatrix}$$

We want a rotation matrix that involves the rotation angles in linear functions so that we can solve a set of linear equations for the rotation angles. The sine and cosine functions are non-linear and require iterative methods to solve for the rotation angles. Linearizing the matrix would make solving for the rotation angles trivial. Since the rotation

angles are expected to be very small, second-order terms in the Taylor series expansion of the matrix about the rotation angles, $A=0$, $B=0$, and $C=0$, are insignificant. So, our linear approximation is a good approximation.

The Taylor series expansion neglecting the higher-order terms is:

$$\mathbf{P} = \mathbf{I} + \left. \frac{\partial \mathbf{P}}{\partial A} \right|_{P_0} A + \left. \frac{\partial \mathbf{P}}{\partial B} \right|_{P_0} B + \left. \frac{\partial \mathbf{P}}{\partial C} \right|_{P_0} C$$

The partial derivatives of the P matrix are:

$$\frac{\partial \mathbf{P}}{\partial A} = \begin{bmatrix} 0 & \cos A \sin B \cos C - \sin A \sin C & \sin A \sin B \cos C + \cos A \sin C \\ 0 & -\cos A \sin B \sin C - \sin A \cos C & -\sin A \sin B \sin C + \cos A \cos C \\ 0 & -\cos A \cos B & -\sin A \cos B \end{bmatrix}$$

$$\frac{\partial \mathbf{P}}{\partial B} = \begin{bmatrix} -\sin B \cos C & \sin A \cos B \cos C & -\cos A \cos B \cos C \\ \sin B \sin C & -\sin A \cos B \sin C & -\cos A \cos B \sin C \\ \cos B & \sin A \sin B & -\cos A \sin B \end{bmatrix}$$

$$\frac{\partial \mathbf{P}}{\partial C} = \begin{bmatrix} -\cos B \sin C & -\sin A \sin B \sin C + \cos A \cos C & \cos A \sin B \sin C + \sin A \cos C \\ -\cos B \cos C & -\sin A \sin B \cos C - \cos A \sin C & \cos A \sin B \cos C - \sin A \sin C \\ 0 & 0 & 0 \end{bmatrix}$$

The partial derivatives evaluated at $P_0 = (A=0, B=0, C=0)$ are:

$$\left. \frac{\partial \mathbf{P}}{\partial A} \right|_{P_0} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix}$$

$$\left. \frac{\partial \mathbf{P}}{\partial B} \right|_{P_0} = \begin{bmatrix} 0 & 0 & -1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

$$\left. \frac{\partial \mathbf{P}}{\partial C} \right|_{P_0} = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

The Taylor series expansion becomes:

$$\mathbf{P} = \mathbf{I} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & A \\ 0 & -A & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & -B \\ 0 & 0 & 0 \\ B & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & C & 0 \\ -C & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Now that the point has been rotated and translated, we want to map it back into the image plane. The x and y coordinates in the image plane are obtained by simply dividing the X and Y world coordinates by the Z coordinate. Note that the principal distance has been normalized to 1. To obtain the actual x and y coordinates given the principal distance of the camera, the $(x, y, 1)$ point in the image plane is simply scaled by the principal distance.

The x and y coordinates are:

$$x = \frac{X}{Z}$$

$$y = \frac{Y}{Z}$$

The motion field is made up of the motion of points in the image plane as the camera is translated and rotated. The components of the motion vector for a point in the image plane are:

$$u = \dot{x} = \frac{\dot{X}}{Z} - \frac{X\dot{Z}}{Z^2}$$

$$v = \dot{y} = \frac{\dot{Y}}{Z} - \frac{Y\dot{Z}}{Z^2}$$

The dotted world coordinates, \dot{X} , \dot{Y} , \dot{Z} , can represent changes in the world coordinates as a function of time or images. The rotation and translation vectors can be rotational and translational velocities. Alternatively, the rotation and translation vectors can be the translation and rotation of the camera between images. The natural unit of measurement for most vision systems is "images" since processing is

done using images. Note that the conversion between units is easy since the time between images can be easily obtained.

The dotted world coordinates can be obtained from:

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{bmatrix} = \begin{bmatrix} X' \\ Y' \\ Z' \end{bmatrix} - \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = (\mathbf{P} - \mathbf{I}) \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} - \begin{bmatrix} U \\ V \\ W \end{bmatrix}$$

Using the Taylor series expansion, the dotted world coordinates are:

$$\dot{X} = CY - BZ - U$$

$$\dot{Y} = AZ - CX - V$$

$$\dot{Z} = BX - AY - W$$

Plugging in to find the motion vector yields:

$$u = \frac{CY - BZ - U}{Z} - \frac{X(BX - AY - W)}{Z^2}$$

$$v = \frac{AZ - CX - V}{Z} - \frac{Y(BX - AY - W)}{Z^2}$$

which simplifies to:

$$u = \frac{xW - U}{Z} + xyA - (x^2 + 1)B + yC$$

$$v = \frac{yW - V}{Z} + (y^2 + 1)A - xyB - xC$$

These equations were introduced by Longuet-Higgins and Prazdny [13], although they did not discuss the linearization. The derivation of the linearized equations was presented here so it was clear from where they came. Additionally, the original equations could be used with a non-linear iterative method if the rotation between images were not small.

Optical Flow

Relative motion between the camera and the scene can also be described by an optical flow, which represents how the image is changing through successive images. An object that moves relative to the camera causes its image to also move in the image plane. The movement of the image can be described by a flow in some direction. One motion field, however, can produce many optical flows or no optical flow. See *Robot Vision* by Horn [10] for more information on optical flow.

The assumption in this thesis is that the optical flow equals the motion field. In most cases, this assumption is valid. Only in special cases does this assumption break down. For example, when a perfectly reflecting sphere rotates, there is no optical flow, but there is definitely a motion field (It is rotating!). A sample optical flow is shown in Figure 2.5. For the purposes of this thesis, one arrow in the optical flow will be called a flow vector. This terminology is not standard.

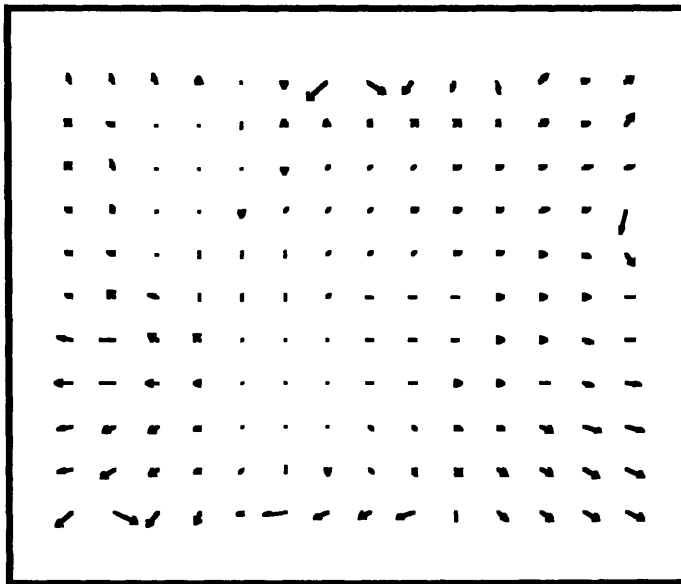


Figure 2.5 Sample Optical Flow

CCD Cameras for Imaging the Environment

The common imaging device used in machine vision systems is a CCD camera. A lens focuses the incoming light rays on a silicon array of photocells. Photons create free electrons that are captured and measured to determine the brightness at each pixel element.

A lens is used to capture more light emanating from a scene point and focus this light on a single point in the image plane. When parallel rays traveling along the optical axis of a lens pass through the lens, they are focused at a single point. The distance from the lens to this point is the lens' focal length. When light rays emanate from a point source and strike the lens, they are focused at a point which lies on the line formed by the point source and the center of projection of the lens. The center of projection is the optical center of the lens. In comparison with a pin-hole system, a lensing system captures more light from each point source and reduces the required sensitivity of the image plane.

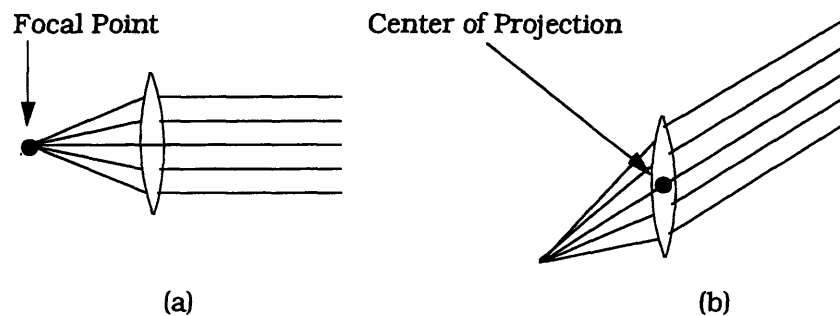


Figure 2.6 Function of a Lens. (a) Light rays traveling parallel to the optical axis are focused at the focal point. (b) Light rays coming from some point in the scene are focused at a point on the line formed by the scene point and the center of projection of the lens.

A CCD camera must perform 4 basic steps to generate an image. In the first step, free electrons are generated via the photoelectric effect when incoming light strikes the silicon. After a sufficient amount of exposure time, the electrons must be collected at the nearest gate, or pixel. The number of electrons collected will be a linear function of the light irradiating the area near the gate. In the third step, a row of pixels is

transferred to output registers from where the pixels will be output sequentially. The last step involves detecting the amount of charge stored for a particular pixel and generating an output voltage. Figure 2.7 shows how each row is selected and transferred to the output registers from where each pixel is read and amplified.

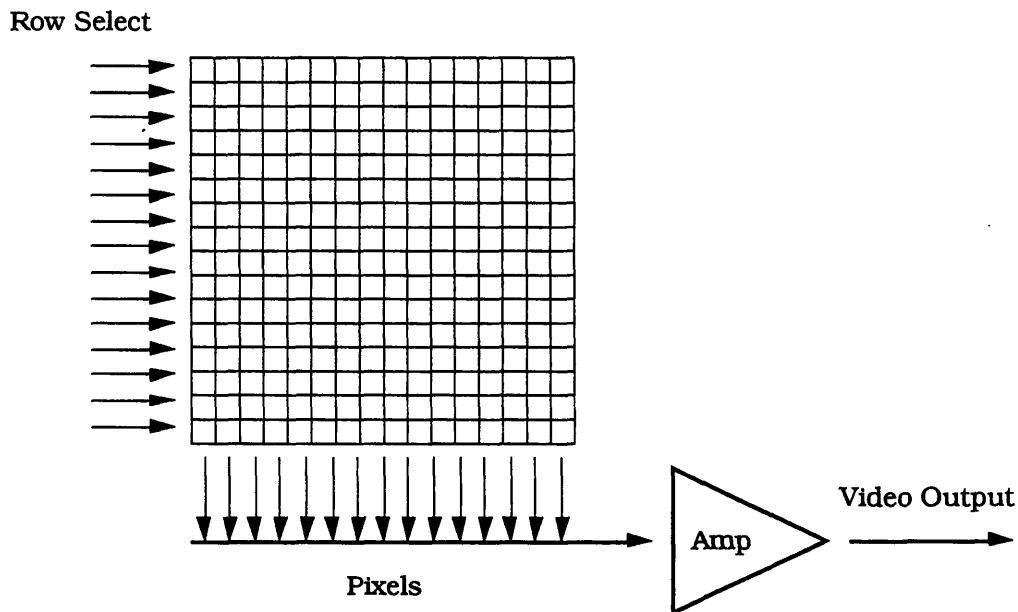


Figure 2.7 Sequential Output of CCD Array. Each row of pixels is selected and transferred to output registers from where the pixels are read and amplified sequentially.

Video Output Formats

There are three common video output formats. The standard format that is compatible with televisions and monitors is the RS-170, or NTSC format, which contains lines of video information separated by sync pulses. A slightly decoded version is the DC-restored video format, in which the sync pulses are separated from the video data. The format that is easiest to directly store in memory is the digital output format, which contains the digitized brightness, or pixel, value.

The RS-170 format is commonly used because it can be directly connected to most televisions and monitors. This format contains 30 images per second. Each image, or frame, is divided into 2 fields - one

field contains the odd lines, and the other field contains the even lines. There is a field sync pulse that signals the beginning of a field. Each field has a distinct field sync pulse. Each line begins with a line sync pulse, which is followed by the video data encoded as an analog signal. The brightness values are offset with a negative DC value, so that the maximum positive and negative excursions are the same.

Recovering the pixels is very difficult because the sampling rate and timing must be chosen correctly. First, the DC level must be restored so that the minimum brightness value is zero. Then, a sampling rate has to be chosen, and the analog video signal has to be sampled. Even if the sampling rate is the same as that used by the camera in encoding the pixels, it is improbable that the timing is correct. Therefore, the pixel values sampled will be some intermediate values between two pixels.

The DC restored format only requires an analog-to-digital converter to produce digital pixels that can be stored in memory. The analog video data is separated from the frame and line sync pulses, and the DC level has been restored so that the minimum brightness value is zero. The rising edge of the pixel clock signals that a pixel should be sampled from the analog video signal.

The digital output format contains the digitized pixels and a pixel clock. The video output can be directly connected to a memory chip. Frame and line sync pulses define the beginning of a frame or line. The pixel clock signals when a pixel can be loaded into memory. Figure 2.8 shows the 3 common output formats.

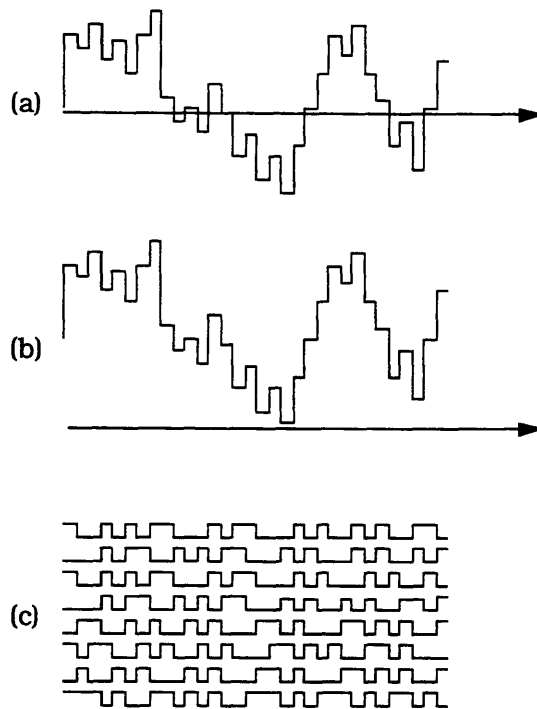


Figure 2.8 Video Output Formats. (a) Standard video format. (b) DC-restored video signal. (c) Digital output.

Grabbing Images from a CCD Camera

Implementing a motion vision system involves capturing images from the output of a CCD camera. If the output of the camera is in the digital format, all that is involved is some logic circuitry, such as counters, that will address the pixels. The output of the camera is connected to the data lines of the memory chip, and the pixel clock is connected to the write enable pin of the memory chip and the count pin of the counter. If the DC-restored format is used, an analog-to-digital converter is required between the camera and the data lines of the memory chip. A specialized chip, or frame grabbing board, is required for the RS-170, or NTSC, format since the sync pulses are embedded in the video signal and must be separated. When the images are stored in memory, they are ready to be processed by the vision system. Following will be a discussion of a common method of implementing a motion vision system.

A Popular Method - Constant Brightness Assumption

A popular way of determining camera motion between two successive images is to relate the camera motion to brightness derivatives in the image. The six parameters describing the translation and rotation of the camera can be related to the motion field. Similarly, the brightness derivatives in the image can be related to the motion field. Eliminating the motion field gives equations involving only the motion parameters, for which we want to solve, and the brightness derivatives, which we can compute from the images.

The motion parameters can be related to the motion field through the Longuet-Higgins and Prazdny equations. These equations describe the motion field in terms of the translation and rotation vectors. The translation vector $\mathbf{t} = [U \ V \ W]^T$ and the rotation vector $\mathbf{p} = [A \ B \ C]^T$ are related to a two-dimensional motion field in the image plane by:

$$u = \frac{-U + xW}{Z} + Axy - B(x^2 + 1) + Cy$$
$$v = \frac{-V + yW}{Z} - Bxy + A(y^2 + 1) - Cx$$

The variables u and v are the x and y components of the motion field.

The brightness constraint equation introduced by Horn and Schunck [8] relates brightness derivatives in the image to optical flow. They assume that the scene points are Lambertian reflectors that radiate light equally in all directions. Thus, any scene point appears equally bright from any viewing angle. As the camera moves, the image of the scene point will move, but its brightness remains unchanged.

Another assumption that is considered very minor, but in some cases may not be, is that objects do not become hidden by other objects as the camera moves. This assumption cannot be completely satisfied because some scene points will become obstructed. However, if the images are grabbed at a sufficient rate, the number of obstructed points will be small.

The brightness constraint equation is obtained by differentiating brightness with respect to time and setting the derivative equal to zero. The constant brightness assumption states that the brightness, E , of a scene point does not change, therefore the time derivative is zero. Brightness is a function of not only time, but also the x and y coordinates in the image plane, so partial derivatives are involved. The brightness constraint equation is:

$$\frac{dE}{dt} = \frac{\partial E}{\partial x} \frac{dx}{dt} + \frac{\partial E}{\partial y} \frac{dy}{dt} + \frac{\partial E}{\partial t} = E_x u + E_y v + E_t = 0$$

The variables u and v are the components of optical flow. The partial derivatives are the brightness derivatives in the image.

To relate the motion parameters to the brightness derivatives, it is necessary to assume that the optical flow in the brightness constraint equation represents the motion field. This assumption is valid for most cases. Only for unusual scenes is this assumption invalid.

The motion field can be eliminated by plugging the Longuet-Higgins and Prazdny equations into the brightness constraint equation. Horn and Weldon [9] showed that the motion parameters are related to the brightness derivatives by:

$$\frac{\mathbf{s} \cdot \mathbf{t}}{Z} + \mathbf{v} \cdot \mathbf{p} + E_t = 0$$

where \mathbf{s} and \mathbf{v} are given by:

$$\mathbf{s} = [-E_x \quad -E_y \quad xE_x + yE_y]^T$$

$$\mathbf{v} = [E_y + y(xE_x + yE_y) \quad -E_x - x(xE_x + yE_y) \quad yE_x - xE_y]^T$$

This equation will be called the brightness derivative motion equation (BDME) in this thesis.

Too Many Unknowns

The BDME contains many unknowns. The six parameters describing the translation and rotation of the camera are what we want. However, the scene depth value, Z , at each point in the image plane is also unknown. For a 256 x 256 pixel image, there are 256 times 256 plus 6, or 65,542 unknowns.

There are more unknowns than equations, so there are many solutions for the motion of the camera. The BDME is valid at each point in the image. Choosing an arbitrary camera motion allows the solution of the scene depth, Z , for each point in the image.

A method to arrive at the correct camera motion and scene depth was introduced by Heel [7]. It involves initially assuming a constant depth over the entire scene and iterating through the motion parameters and depth values until the motion parameters converge. A constant depth allows an average motion to be computed. Adjusting the depth values will move them in the correct direction, either nearer to or farther from the camera. A new calculation of the motion parameters will yield a better estimate, from which the depth values can be adjusted. Eventually, the motion parameters will settle and converge to a solution. Heel has shown convergence within ten iterations.

The iterations can be performed over a pair of images or a sequence of images. Using a pair of images will allow one to estimate the camera motion between 2 images. Using a sequence of images, one can estimate some average motion over the sequence.

When using a pair of images, some post-filtering will be necessary since noise can be expected in the motion estimates. The filter would average say the last 5 motion estimates to produce the current motion estimate that would be used by the control computer. Using a sequence of images incorporates the filtering operation within the motion estimation process at the expense of larger convergence times.

When iterating over a sequence of images, the current depth map must be "warped" by the current motion estimate before being used to estimate the motion in the next stage of the iteration. This introduces more calculations in the motion estimation process. Heel [7] proposed an interesting way of "warping" the current depth map. However, in the opinion of the author, the filtering could be delayed to post-processing, where the calculations would be simpler and less intensive than warping the depth map.

Assuming a constant scene depth removes the dependence of the motion estimate on the scene. Although some constant value is assumed, the scale factor ambiguity problem makes this constant irrelevant. Thus, the first estimate of the motion is independent of the scene.

Estimating Motion Using Assumed or Known Depth Values

Assuming some estimate of the scene depth at each point in the image allows the motion of the camera to be solved via a Least Squares minimization. The brightness constraint equation will not be exactly satisfied at each pixel because we are using estimates of the scene depth. Thus, each BDME will not equal zero, but some small error:

$$\frac{\mathbf{s} \cdot \mathbf{t}}{Z} + \mathbf{v} \cdot \mathbf{p} + E_t = \varepsilon$$

One way of determining the motion that best fits the data is by squaring the errors, which is equivalent to squaring the BDME's, and minimizing their sum:

$$SSE = \sum_x \sum_y \left(\frac{\mathbf{s} \cdot \mathbf{t}}{Z} + \mathbf{v} \cdot \mathbf{p} + E_t \right)^2$$

Differentiating with respect to the translation and rotation vectors yields a set of six linear, independent equations, three for translation and three for rotation:

$$\left(\sum_x \sum_y \frac{\mathbf{s}\mathbf{s}^T}{Z^2} \right) \mathbf{t} + \left(\sum_x \sum_y \frac{\mathbf{s}\mathbf{v}^T}{Z} \right) \mathbf{p} = - \sum_x \sum_y \frac{E_t \mathbf{s}}{Z}$$

$$\left(\sum_x \sum_y \frac{\mathbf{v}\mathbf{s}^T}{Z} \right) \mathbf{t} + \left(\sum_x \sum_y \mathbf{v}\mathbf{v}^T \right) \mathbf{p} = - \sum_x \sum_y E_t \mathbf{v}$$

Estimating Depth from Known Motion Values

The depth can be estimated at each pixel by squaring the BDME and differentiating it with respect to the depth, Z . The process minimizes the squared error in the BDME. Noise can be removed from the depth estimates by grouping neighboring pixels together and estimating a depth that minimizes the sum of squared errors. The depth estimate using only one BDME is:

$$Z = \frac{(\mathbf{s} \cdot \mathbf{t})^2}{-(\mathbf{v} \cdot \mathbf{p} + E_t) \mathbf{s} \cdot \mathbf{t}}$$

Computational Requirements of the Brightness Derivatives Method

The computational requirements of the brightness gradient method unfortunately prohibit its implementation on the Micro-Rover because about 3 million computations are required for one pair of images. The number of computations required to compute the derivatives, the \mathbf{s} and \mathbf{v} intermediate vectors, and the dyadic products for each pixel are shown below in Table 2.1. These computations must be performed for each new pair of images. Table 2.2 lists operations that must be performed at each step of the iteration. These involve incorporating the new depth values into the Least-Squares equations. For example, if the iteration were over a pair of images, the computations in Table 2.1 would be executed at the beginning, and the calculations in Table 2.2 would be performed at each step in the iteration. Alternately, if the iteration were over a sequence of images, the computations in Tables 2.1 and 2.2 would be performed for every new image, which is equivalent to every step of the iteration.

Table 2.1 Computations per Pixel to Initialize an Iteration

Calculation	\pmSign	Mult	Div	Add	Sub
derivatives					
Ex, Ey					2
Et					1
vectors					
s	2	2		1	
v	1	4		1	2
dyadics					
ss^T		6			
sv^T		9			
vs^T					
vv^T		6		9	
scaling					
-E _t s					
-E _t v		3		3	
Total	3	30		14	5

Table 2.2 Computations per Pixel at Each Step of the Iteration

Calculation	\pmSign	Mult	Div	Add	Sub
dyadics					
ss^T		1	6	9	
sv^T			9	9	
vs^T				9	
scaling					
-E _t s		3	1	3	
Total		4	16	30	

We must compute these items for every pixel in the image, so the total number of computations increases quadratically as the resolution increases. For example, 818,200 computations are required per step in the iteration for a 128 x 128 image, and 3,276,800 computations are required for a 256 x 256 image. The computations involved in solving the set of six linear equations are insignificant and were not considered.

The time required to process one pair of images is prohibitive to implementing the brightness derivative method on the Micro-Rover in real-time. If a 128 x 128 image is used, 0.27 of a second is required for processing one pair of images using a 30 MHz processor that requires 10 clock cycles per computation. If a 256 x 256 image is used, 1.09 seconds is required for processing. We would like to process about five to ten image pairs per second.

One must keep in mind that the size and power constraints for a small vehicle such as the Micro-Rover limit the complexity of the processing unit used to perform the vision analysis. Many vision systems implement this method using parallel workstations. However, the space required to house multiple processors and the power required to run them is obviously not realistic.

An Alternate Method - The Constant Pattern Assumption

If a vision system is to be implemented on the Micro-Rover, the computational complexity must be reduced without sacrificing accuracy. It is obvious that the Micro-Rover cannot perform the required processing of the brightness derivative method. However, the processing cannot be reduced by using less derivatives because this method depends on the overlap of the derivatives. Thus, a new method must be developed for the Micro-Rover.

A different approach that offers potential is finding matching image patches between two images to estimate the motion field and then performing a Least-Squares minimization to find the motion causing the motion field. When an image patch in one image is matched to an image

patch in another image, the offset in horizontal and vertical pixels can be used as an estimated flow vector. Matching an entire image will yield an estimated optical flow. We can then assume that the optical flow represents the motion field, as before. The estimated motion field can then be compared to the calculated motion field, obtained from the depth to the scene patch and the motion parameters of the Micro-Rover. A Least-Squares minimization can then be performed to solve for the motion parameters that minimize the sum of squared errors between estimated and calculated motion fields.

This method offers potential because the matching can be performed via integrated circuits, leaving much less computations for the processor. LSI Logic produces a motion estimation processor, designed for HDTV applications. It takes a section, called the data block, from one image and finds the best match in a larger section, called the search window, of another image. The best match is defined by the smallest match error, where a match error is the sum of absolute pixel differences.

A constant pattern assumption can be made about an image patch if the motion of the vehicle is small between two successive images. A scene patch will produce some image pattern on the imaging surface. The pattern in the image should be the same for two camera positions that aren't too different. For example, when the camera moves and rotates slightly between frames, most, if not all, of the scene points making up an image patch are still visible. Thus, the patch in the new image should be the same as that in the old, except for a slight translation and/or rotation.

The constant pattern assumption is similar to the constant brightness assumption, except that the constant pattern assumption imposes more constraints and thus can be solved locally. The constant brightness assumption relates the time derivative of brightness at a pixel to the spatial derivatives of brightness at that pixel and the motion of that pixel. There are many motions that will solve for the change in brightness. For example, two solutions are $(0, -E_t/E_y)$ and $(-E_t/E_x, 0)$. However, the constant pattern assumption requires the ordering of pixels to be the

same. One can isolate some image patch in one image and find the best match to it within some region of another image.

Determining Motion from Matching Data

The motion field obtained from the Longuet-Higgins and Prazdny equations can be thought of as the calculated motion field. Given the translation and rotation vectors describing the motion of the Micro-Rover and the depth to each scene patch, the motion field can be calculated from these equations.

The results of the matching algorithm can be thought of as the estimated motion field. The best match of each data block in the first image is found within a search window in the second image. The offset to the best match position is the estimated optical flow, which is assumed to equal the motion field.

An error between the calculated and estimated motion field can be defined at each point in the image. Letting the carets represent the calculated motion field, the squared error at a point in the image can be defined as:

$$\text{Error} = (\hat{u} - u)^2 + (\hat{v} - v)^2$$

The errors at each point in the image can be summed, and the aggregate error can be minimized to solve for the camera motion. The effect of noise in the estimated motion field would be reduced, thereby producing a better estimate of motion. The errors can also be weighted to place more weight on and hence forcing the solution closer to the scene points that are very distinctive.

Derivation of the Least-Squares Solution

The Longuet-Higgins and Prazdny equations for the calculated motion field can be written in vector form as:

$$\hat{u} = \frac{\mathbf{s}_u \cdot \mathbf{t}}{Z} + \mathbf{r}_u \cdot \mathbf{p}$$

$$\hat{v} = \frac{\mathbf{s}_v \cdot \mathbf{t}}{Z} + \mathbf{r}_v \cdot \mathbf{p}$$

where the vectors:

$$\mathbf{s}_u = [-1 \ 0 \ x]^T$$

$$\mathbf{s}_v = [0 \ -1 \ y]^T$$

$$\mathbf{r}_u = [xy \ -(x^2 + 1) \ y]^T$$

$$\mathbf{r}_v = [y^2 + 1 \ -xy \ -x]^T$$

are the coefficients of the motion parameters, and the vectors:

$$\mathbf{t} = [U \ V \ W]^T$$

$$\mathbf{p} = [A \ B \ C]^T$$

are the translation and rotation vectors.

The sum of squared errors can be written as:

$$SSE = \sum_x \sum_y (\hat{u} - u)^2 + (\hat{v} - v)^2$$

The Least-Squares solution is found by taking the partial derivatives with respect to the translation and rotation vectors and setting each of the two vector equations equal to the zero vector. The partial derivative of the aggregate error with respect to the translation vector is:

$$\left(\sum_x \sum_y \frac{\mathbf{s}_u \mathbf{s}_u^T + \mathbf{s}_v \mathbf{s}_v^T}{Z^2} \right) \mathbf{t} + \left(\sum_x \sum_y \frac{\mathbf{s}_u \mathbf{r}_u^T + \mathbf{s}_v \mathbf{r}_v^T}{Z} \right) \mathbf{p} = \sum_x \sum_y \frac{u \mathbf{s}_u + v \mathbf{s}_v}{Z}$$

and the partial derivative with respect to the rotation vector is:

$$\left(\sum_x \sum_y \frac{\mathbf{r}_u \mathbf{s}_u^T + \mathbf{r}_v \mathbf{s}_v^T}{Z} \right) \mathbf{t} + \left(\sum_x \sum_y \mathbf{r}_u \mathbf{r}_u^T + \mathbf{r}_v \mathbf{r}_v^T \right) \mathbf{p} = \sum_x \sum_y \mathbf{u} \mathbf{r}_u + \mathbf{v} \mathbf{r}_v$$

These vector equations give six linear equations in six unknowns. Note that the 6 x 6 matrix is symmetric:

$$\begin{bmatrix} \sum_x \sum_y \mathbf{S} / Z^2 & \sum_x \sum_y \mathbf{G} / Z \\ \sum_x \sum_y \mathbf{G}^T / Z & \sum_x \sum_y \mathbf{R} \end{bmatrix} \begin{bmatrix} \mathbf{t} \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \sum_x \sum_y \mathbf{D} / Z \\ \sum_x \sum_y \mathbf{E} \end{bmatrix}$$

where the following matrices have been substituted:

$$\begin{aligned} \mathbf{S} &= \mathbf{s}_u \cdot \mathbf{s}_u^T + \mathbf{s}_v \cdot \mathbf{s}_v^T \\ \mathbf{G} &= \mathbf{s}_u \cdot \mathbf{r}_u^T + \mathbf{s}_v \cdot \mathbf{r}_v^T \\ \mathbf{G}^T &= \mathbf{r}_u \cdot \mathbf{s}_u^T + \mathbf{r}_v \cdot \mathbf{s}_v^T \\ \mathbf{R} &= \mathbf{r}_u \cdot \mathbf{r}_u^T + \mathbf{r}_v \cdot \mathbf{r}_v^T \end{aligned}$$

as well as the following vectors:

$$\begin{aligned} \mathbf{D} &= \mathbf{u} \mathbf{s}_u + \mathbf{v} \mathbf{s}_v \\ \mathbf{E} &= \mathbf{u} \mathbf{r}_u + \mathbf{v} \mathbf{r}_v \end{aligned}$$

The \mathbf{S} , \mathbf{G} , and \mathbf{R} matrices can be described in terms of their elements:

$$\mathbf{S} = \begin{bmatrix} 1 & 0 & -x \\ 0 & 1 & -y \\ -x & -y & x^2 + y^2 \end{bmatrix}$$

$$\mathbf{G} = \begin{bmatrix} -xy & x^2 + 1 & -y \\ -y^2 - 1 & xy & x \\ y^3 + (x^2 + 1)y & -x^3 - (y^2 + 1)x & 0 \end{bmatrix}$$

$$\mathbf{R} = \begin{bmatrix} y^4 + 2y^2 + x^2y^2 + 1 & -x^3y - 2xy - xy^3 & -x \\ -x^3y - 2xy - xy^3 & x^4 + 2x^2 + x^2y^2 + 1 & -y \\ -x & -y & x^2 + y^2 \end{bmatrix}$$

The **S**, **Q**, and **R** matrices do not contain the estimated motion field, i.e. u and v , and thus can be computed for each image patch and stored in memory. The matrices are simply scaled by some function of the depth to each scene patch and summed to obtain the 6 x 6 matrix.

The **D** and **E** vectors contain the estimated motion field, and thus cannot be pre-calculated. The u and v components of the estimated motion field multiply the **s** and **r** vectors. The **s** vectors are then divided by the depth to the scene patch.

Another Way of Obtaining the Least-Squares Equations

The Least-Squares equations can also be obtained from a matrix-form derivation that is similar to Strang's presentation in *Introduction to Applied Mathematics*. Each of the flow vectors imposes 2 constraints on the vehicle motion:

$$\begin{bmatrix} 0 & \frac{-1}{Z} & \frac{y}{Z} & (y^2 + 1) & -xy & -x \\ \frac{-1}{Z} & 0 & \frac{x}{Z} & xy & -(x^2 + 1) & y \end{bmatrix} \mathbf{m} = \begin{bmatrix} u \\ v \end{bmatrix}$$

where

$$\mathbf{m} = \begin{bmatrix} U \\ V \\ W \\ A \\ B \\ C \end{bmatrix}$$

However, there is one unknown for each flow vector, namely the scene depth, Z . So, each flow vector actually imposes only one constraint. Note that if the scene depth is known for each flow vector, only 3 flow vectors are needed to solve for the motion because each flow vector will impose 2 constraints on the motion.

Over an entire image, there are more constraints than variables so the system becomes over-determined. We get a $2n \times 6$ matrix, where "n" is the number of flow vectors:

$$\begin{bmatrix}
 0 & \frac{-1}{Z_1} & \frac{y_1}{Z_1} & (y_1^2 + 1) & -x_1 y_1 & -x_1 \\
 \frac{-1}{Z_1} & 0 & \frac{x_1}{Z_1} & x_1 y_1 & -(x_1^2 + 1) & y_1 \\
 0 & \frac{-1}{Z_2} & \frac{y_2}{Z_2} & (y_2^2 + 1) & -x_2 y_2 & -x_2 \\
 \frac{-1}{Z_2} & 0 & \frac{x_2}{Z_2} & x_2 y_2 & -(x_2^2 + 1) & y_2 \\
 \cdot & & & & & \\
 \cdot & & & & & \\
 0 & \frac{-1}{Z_n} & \frac{y_n}{Z_n} & (y_n^2 + 1) & -x_n y_n & -x_n \\
 \frac{-1}{Z_n} & 0 & \frac{x_n}{Z_n} & x_n y_n & -(x_n^2 + 1) & y_n
 \end{bmatrix} \mathbf{m} = \mathbf{A} \mathbf{m} = \begin{bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ \cdot \\ \cdot \\ u_n \\ v_n \end{bmatrix} = \mathbf{b}$$

Since the system is over-determined, we want to choose a solution, \mathbf{m} , that minimizes the squared Euclidean distance:

$$\|\mathbf{A} \mathbf{m} - \mathbf{b}\|^2$$

Strang showed that the solution can be found by multiplying both sides by \mathbf{A}^T . The result is a 6×6 matrix representing 6 linear equations:

$$\mathbf{A}^T \mathbf{A} \mathbf{m} = \mathbf{A}^T \mathbf{b}$$

The result is the same as found previously by differentiating the sum of squared errors and setting the result equal to zero. This method was presented because it provides more insight. Namely, if the scene depth for each flow vector is unknown, 6 flow vectors are required since each imposes 1 net constraint on the motion. However, if the scene depth is known, only 3 flow vectors are required since each imposes 2 constraints.

Implementation of the Matching Algorithm

The memory requirement to store the pre-calculated **S**, **g**, and **R** matrices for each data block is very modest. The **S** matrix, which like the other two is 3 x 3, requires 5 different terms to be stored since it is symmetric and contains one zero in the upper triangular. The **g** matrix requires 8 terms to be stored since it is not symmetric, but contains one zero. The **R** matrix requires no terms to be stored for each scene patch since it is independent of any of the data. The individual **R** matrices can be pre-calculated, summed, and stored in the lower-right 3 x 3 matrix of the 6 x 6 matrix. Therefore, 13 terms need to be stored for each data block. There are 14 times 14, or 196 data blocks in a 256 x 256 image using 16 x 16 data blocks. Thus, a total of 13 times 196, or 2,548 terms need to be stored.

The number of calculations required to compute the 6x6 matrix are also modest. As shown in Table 2.3, 55 computations are required for each data block. For a 256 x 256 image, a total of 10,780 calculations are required.

Table 2.3 Matrix Calculations for Each Data Block

Calculation	Mult	Div	Add	Sub
S matrix	1	4	7	
g matrix		7	7	1
g^T matrix			8	
R matrix				
D vector	4	4	4	
E vector	4		4	
Total	9	15	30	1

Solving the Least-Squares Minimization Equations

The 6 linear equations in 6 unknowns can be solved using Gaussian elimination to find the motion, namely the translation and rotation, of the Micro-Rover. Gaussian elimination involves multiplying and subtracting equations until the 6 x 6 matrix is upper triangular. The solutions can then be picked off quickly.

The method of Gaussian elimination can be performed very systematically. Eliminating the first element in each row involves multiplying the first equation by the ratio of the first element in the desired row by the first element in the first row and subtracting the scaled first equation from the desired equation. The first element in the desired row is eliminated since from it is subtracted the product of the first element in the first row and the ratio of the first element in the desired row to the first element in the first row.

The total number of calculations to get the 6 x 6 matrix in upper-triangular form is 170 as shown in Table 2.4. Eliminating the first element requires 7 multiplications and 7 subtractions for each of 5 equations. The second element requires 6 multiplications and 6 subtractions to eliminate it from 4 equations. The last elimination involves 3 multiplications and 3 subtractions for 1 equation.

Table 2.4 Rearranging Matrix into Upper Triangular Form

Row	Mult	Div	Add	Sub
first	7x5	5		7x5
second	6x4	4		6x4
third	5x3	3		5x3
fourth	4x2	2		4x2
fifth	3x1	1		3x1
Total	85	15		85

To solve for the 6 variables given an upper-triangular 6 x 6 matrix, 36 computations are required as shown in Table 2.5. The variables that have been solved must be multiplied by their appropriate factors and then subtracted from the right-hand side of the equation. The variable which is being solved is then the right-hand side divided by its factor. The sixth equation requires 1 division. The fifth equation requires 1 multiplication, 1 subtraction, and 1 division. The fourth equation requires 2 multiplications, 2 subtractions, and 1 division.

Table 2.5 Calculations to Solve Upper Triangular Matrix

Equation	Mult	Div	Add	Sub
first	5	1		5
second	4	1		4
third	3	1		3
fourth	2	1		2
fifth	1	1		1
sixth		1		
Total	15	6		15

Updating the Depth Map Using the New Motion Parameters

After the motion parameters have been estimated from the Least-Squares minimization, the depth map must be updated. Solving for the motion parameters is an iterative process that requires the depth map to be found. When the depth map converges, the motion parameters can be faithfully accepted.

Updating the depth map requires plugging the motion parameters into the motion field error equation and setting the error equal to zero. The error involves two squared terms, so each term can be independently set equal to zero. However, it is unlikely that both solutions for the depth value will be the same since the Least Squares solution does not make all the errors equal to zero, but minimizes the aggregate error. Therefore, the error cannot be made zero, but only minimized.

A Least Squares minimization will find the depth value that minimizes the error. The error for one data block is:

$$\text{Error} = (\hat{u} - u)^2 + (\hat{v} - v)^2$$

Differentiating this with respect to the depth yields:

$$2 \left(\frac{\mathbf{s}_u \cdot \mathbf{t}}{Z} + \mathbf{r}_u \cdot \mathbf{p} - u \right) \left(-\frac{\mathbf{s}_u \cdot \mathbf{t}}{Z^2} \right) + 2 \left(\frac{\mathbf{s}_v \cdot \mathbf{t}}{Z} + \mathbf{r}_v \cdot \mathbf{p} - v \right) \left(-\frac{\mathbf{s}_v \cdot \mathbf{t}}{Z^2} \right) = 0$$

Simplifying the equation gives:

$$\left(\frac{\mathbf{s}_u \cdot \mathbf{t}}{Z} + \mathbf{r}_u \cdot \mathbf{p} - u \right) \mathbf{s}_u \cdot \mathbf{t} + \left(\frac{\mathbf{s}_v \cdot \mathbf{t}}{Z} + \mathbf{r}_v \cdot \mathbf{p} - v \right) \mathbf{s}_v \cdot \mathbf{t} = 0$$

which can be manipulated into:

$$(\mathbf{s}_u \cdot \mathbf{t})^2 + Z (\mathbf{r}_u \cdot \mathbf{p} - u) \mathbf{s}_u \cdot \mathbf{t} + (\mathbf{s}_v \cdot \mathbf{t})^2 + Z (\mathbf{r}_v \cdot \mathbf{p} - v) \mathbf{s}_v \cdot \mathbf{t} = 0$$

Finally, the solution for the depth is:

$$Z = \frac{(s_u \cdot t)^2 + (s_v \cdot t)^2}{(u - r_u \cdot p)s_u \cdot t + (v - r_v \cdot p)s_v \cdot t}$$

The Least Squares minimization adds an additional 5,096 computations to the number of computations performed per image pair as shown in Table 2.6.

Table 2.6 Calculations to Update a Depth Value

Calculation	Mult	Div	Add	Sub
numerator	6		3	
denominator	8	1	5	2
Total	14	1	8	2

Implementing the Algorithm on a Microprocessor

Implementing the algorithm on a microprocessor requires a consideration of the number of computations performed per second and the speed of the processor. There are 15,886 computations to be performed per step of the iteration. Allowing a safety margin of 50% adds an additional 7,943 computations for a total of 23,829 computations per step as shown in Table 2.7. The vision system is expected to produce 10 motion estimates per second, i.e. 10 iterations or image pairs. Assuming that each iteration requires 10 steps to converge, the total number of computations per second is 2,382,900. This is within the range of a '486-based single board computer.

Table 2.7 Computations Required per Step of the Iteration

Type of Computation	Number of Computations
Compute 6x6 Matrix	6,860
Compute 6-D Vector	3,920
Solve Linear Equations	206
Update Depth Map	4,900
Safety Margin	7,943
Total	23,829

The microprocessor used for the final processing will be expected to run at 30 MHz. Assuming a 50% duty ratio, the effective rate would be 15 MHz to be used for processing. To perform 2,382,900 computations per second, the microprocessor could allow 6.2 cycles per computation as shown in Table 2.8. This seems especially reasonable assuming that the microprocessor contains an internal coprocessor to handle the floating-point arithmetic and that it will be programmed in assembly language with a lot of thought going into the memory layout. For example, it would make sense to store the pre-calculated matrices in memory so that an address register need only be incremented, rather than an effective address be calculated.

Table 2.8 Microprocessor Considerations

Clock Speed	30 MHz
Duty Ratio	50%
Effective Clock Speed	15 MHz
Number of Computations	2,382,900
Cycles per Computation	6.2

Chapter Summary

This chapter discussed two methods of implementing a motion vision system. The first method, which uses brightness gradients, cannot be implemented in real-time on a Micro-Rover because its computational requirements are too high. The second method, which uses pattern matching, relies upon an integrated circuit chip to perform a bulk of the processing. The remaining calculations to convert the matching results into motion estimates is significantly less than the number of calculations to compute the brightness gradients. As a result, the pattern matching approach to a motion vision system can be implemented on a Micro-Rover in real-time.

Chapter 3 Testing the Pattern Matching

This chapter discusses the initial testing of the matching algorithm on a Macintosh computer. A brief description of the camera set-up as well as the program used for testing is presented. Initial tests showing the ability of the matching algorithm to find the correct match even when the camera has undergone significant translation and rotation are discussed. The matching algorithm is then applied to entire images to obtain optical flows. The optical flows seem to be accurate representations of the motion field, except for a few incorrect flow vectors. A size of data block test was performed to determine which size - 8 x 8 or 16 x 16 - was more accurate and more useful. Finally, various weighting schemes to highlight correct flow vectors and attenuate incorrect flow vectors are discussed and tested.

Simulation of Matching Algorithm on a Macintosh

The matching algorithm was simulated in software on a Macintosh Ix computer. A Pulnix 7-CN CCD camera with a Computar 8.5 mm auto-iris lens was used. A DigiVideo frame grabbing board was plugged into a NuBus slot on the Macintosh and used to digitize the NTSC video signal from the Pulnix camera. The program to test the matching algorithm on the images grabbed from the DigiVideo board was written in Think C version 6.0. The matching functions are described in Appendix A.

In order to implement the matching algorithm and the subsequent Least-Squares motion estimation equations, an entire Macintosh application was developed. Functions to save images and match results to disk, as well as routines to update the windows, were some of the utilities that were programmed. The full-sized Mac application is named TwinScreen™.

Single Match Tests

The first tests concentrated on determining whether the matching algorithm could find the correct match of a small part of one image

within a larger part of another image. The two images were slightly different views of the same scene. The small part of the first image is called the data block and measures 16 x 16 pixels. The larger part of the second image is called the search window and measures 32 x 32 pixels.

The metric that is used to judge the quality of a match is the sum of absolute differences. Suppose the data block is placed at some position within the search window. Each pixel in the data block can be subtracted from the underlying pixel in the search window. The absolute value of this difference is called the absolute difference. The 256 absolute differences that can be calculated from a 16 x 16 data block are added together and called the sum of absolute differences, or the match error.

The data block can be placed in 289 different positions within the search window. The data block measures 16 pixels in each direction, and the search window measures 32 pixels in each direction. Thus, there are 17 different positions in each direction. The total number of positions is 17 squared, or 289. The position of the data block is chosen to be defined by its top-left corner.

The best match criterion is the smallest match error among the 289 match errors. There will be some 16 x 16 section of the search window whose match error is less than or equal to all the other match errors. The position of the top-left corner of the 16 x 16 pixel section is called the best match position since we have chosen to define the position of the data block by its top-left corner.

A small match error indicates that the data block pattern closely matches the pattern in the search window. For the match error to be small, the search window pattern must have pixels with high brightness values in the same positions as the bright pixels in the data block. Conversely, the search window pattern must have pixels with low brightness values in the same positions as the dim pixels in the data block.

The single match testing was performed on two pairs of images obtained in an office at Draper Laboratory. The first image pair was two different views of a bookcase. The second image pair was two different views of a plastic cup with the Red Sox logo. The translation and rotation between camera positions in a pair of images was not measured, but was more than would be expected between two successive images. The intent was to magnify and twist the data block, and then see if the correct position could be found. The search windows were selected so that they contained the corresponding data block.

The first pair of images tested was two different views of a bookcase. The data block was taken from the label on a binder just slightly right of center in the first image. There are 4 corners in the data block, so the pattern was very distinct. The search window was obtained from the same label in the second image. The two images as well as the data block, search window, and matching results are shown in Figures 3.1-3.5.

The matching results show the errors at each of the positions at which the data block can be placed in the search window. The bright spot in the center of the matching results indicates the best match, i.e. where the sum of absolute differences is minimum. The dark areas represent bad matches, or positions where the sum of absolute differences is maximum. The match errors were scaled to the range 0-127 because the screen's color table contained only 128 shades of gray. The resolution was further reduced by the printer because it could print about 8 different gray halftones.

The matching results for the bookcase images show that the best match for the data block is 9 pixels to the right and 10 pixels down from the top left corner of the search window. In other words, when the top left pixel of the data block is positioned at (9,10) in the search window, the match error is minimum. Since the data block pattern does not exactly match the pattern in the search window, the expected best match is somewhat arbitrary. However, the best match position of (9,10) is one pixel from

any reasonably expected best match that one could estimate by placing the data block in the search window.

Bookcase Images

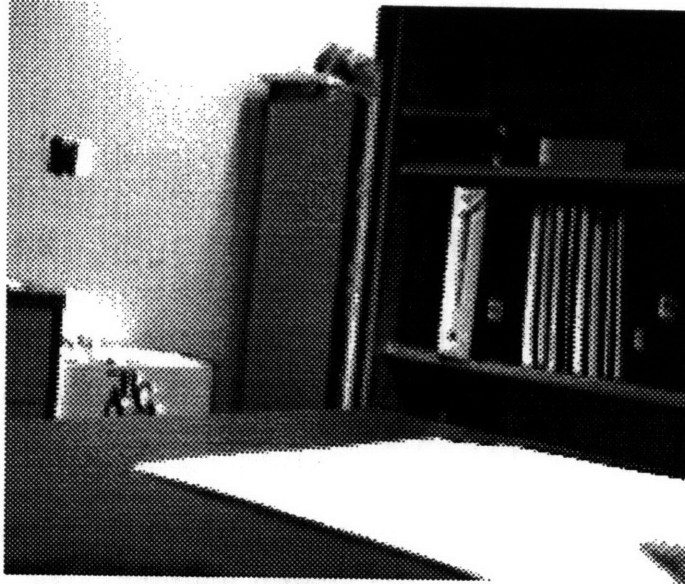


Figure 3.1 First Image of Bookcase

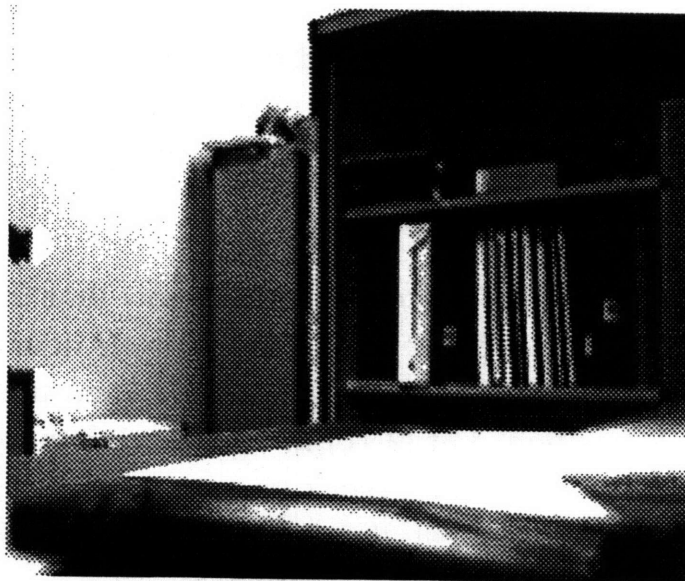


Figure 3.2 Second Image of Bookcase

Data Block, Search Window, and Matching Results

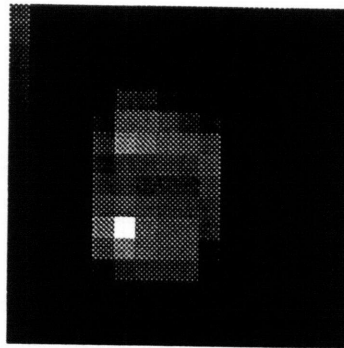


Figure 3.3 Data Block from First Image

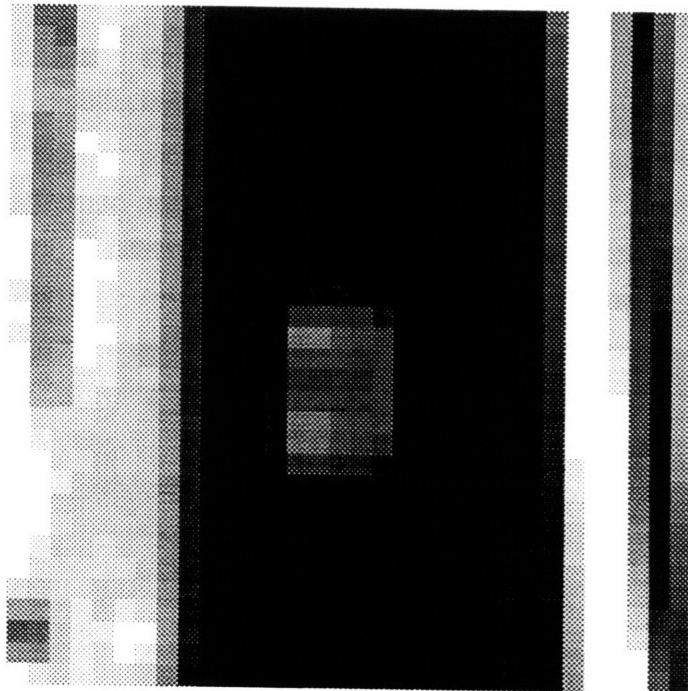


Figure 3.4 Search Block from Second Image

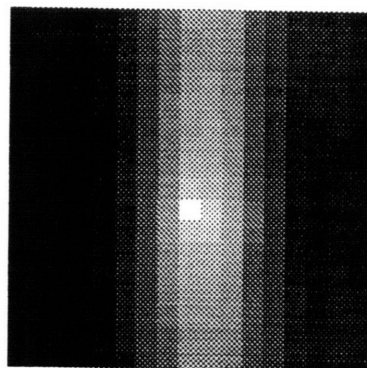


Figure 3.5 Matching Results

The second pair of images was taken of a plastic cup with the Red Sox logo. The data block was taken from the area below the left shoulder of the batter and just under the Red Sox logo. The search window includes the same area. The two images, as well as the data block, search window, and matching results are shown in Figures 3.6-3.10.

Similar to the bookcase results, the bright spot in the Red Sox Cup matching results represents the best match. The match results from the Red Sox Cup image pair show that the best match of the data block is (6,8). Again, this is close to any reasonable interpretation of the data block in the search window.

The main purpose of this first series of tests was to get an idea of what kind of results we should expect for each of the matches. The 2 data blocks were very distinct, so we expected (and obtained) very good matches. If the matches were poor, we would have to reconsider whether this is the correct approach - what would be the results for less distinct data blocks? Also, compared to what we would expect from images on the Micro-Rover, there was significant rotation and some magnification of the data block pattern in the search window. Lesser-quality match results would have required us to determine how much translation and rotation was allowable to produce good match results.

Red Sox Cup Images



Figure 3.6 First Image of Red Sox Cup



Figure 3.7 Second Image of Red Sox Cup

Data Block, Search Window, and Matching Results

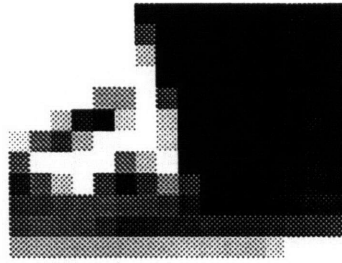


Figure 3.8 Data Block from First Image

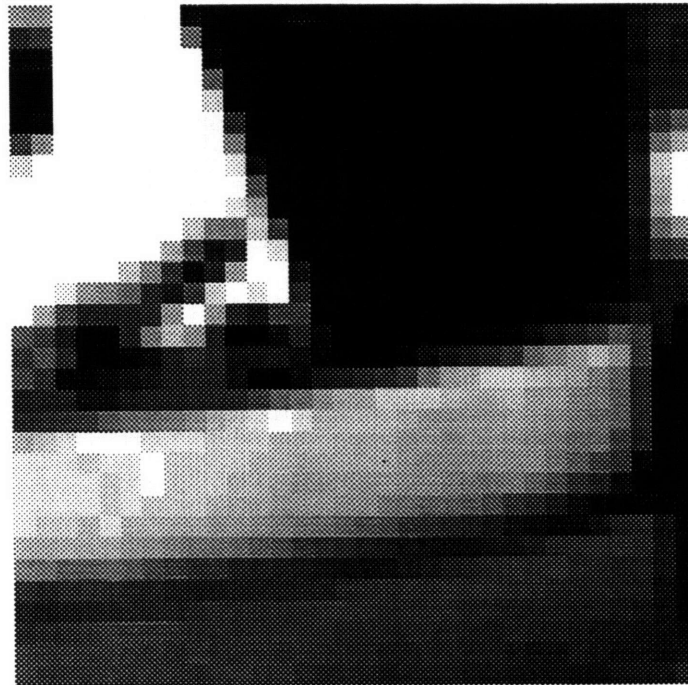


Figure 3.9 Search Window from Second Image

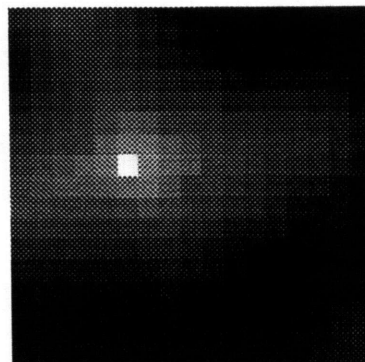


Figure 3.10 Matching Results

Optical Flow Tests

The next series of tests was performed to determine whether the matching algorithm could be applied to an entire image to create an optical flow. It was already determined that the matching algorithm can be used to find the best match of a data block in a search window. If the best match can be calculated for every data block in an image, the best match positions could be compared to the original data block positions. The original position and the best match position define the starting and ending points of a flow vector.

The optical flow was calculated by finding the best match of a data block in a search window and determining how many pixels the data block moved in the image coordinate system. The data block was chosen from the first image and measured 16 x 16 pixels. The search window was taken from the second image and measured 32 x 32 pixels. The search window was chosen by extending the data block position outward by 8 pixels in each direction. Thus, if there were no motion between the images, the best match position would be in the center of the search window.

The flow vector was calculated from the best match position by subtracting from it the zero position. The zero position of the data block was the center of the search window, or (8,8) in the matching results coordinate system, in which each axis ranges from 0 to 16. This position was subtracted from the best match position to produce a two-dimensional vector which represents the flow of the data block from the first image to the second. The x axis points to the right, and the y axis points down. For example, a best match position of (3,4) would yield a flow vector (-5,-4), pointing up and to the left.

The first image was divided into 168 non-overlapping data blocks. The images grabbed from the DigiVideo board measure 256 pixels horizontally by 216 pixels vertically. The data blocks measure 16 x 16 pixels. The image was divided into 14 data blocks horizontally by 12 data blocks vertically, for a total of 168 data blocks. The coordinates of

the data blocks range from (16,12) for the top-left corner of the top-left data block to (224,188) for the top-left corner of the bottom-right data block.

The optical flows for two pairs of images as well as each image pair are shown in Figures 3.11-3.16. The results show that the matching algorithm can produce reasonable optical flows, except for a few incorrect flow vectors. The optical flows were overlaid on the first image in each pair to produce an "overlay flow." The overlay flow allows one to judge the quality of a flow vector by the distinctness of the underlying data block.

Checking the Optical Flow

The optical flow was not checked against the motion field because to do so requires calibrating the camera. The Longuet-Higgins and Prazdny equations can be used with a geometric description of the scene to produce the motion field. However, as a minimum, the center of projection and the principal distance must be determined. It was not desirable to calibrate the camera because it is a separate thesis topic in and of itself. The parameters can be estimated to solve for the motion field, however, the results that would be obtained wouldn't warrant the work involved in computing the motion field. Therefore, a visual check of the optical flows is all that will be used to judge whether they are correct.

Images of Portable Radio



Figure 3.11 First Image of Radio



Figure 3.12 Second Image of Radio

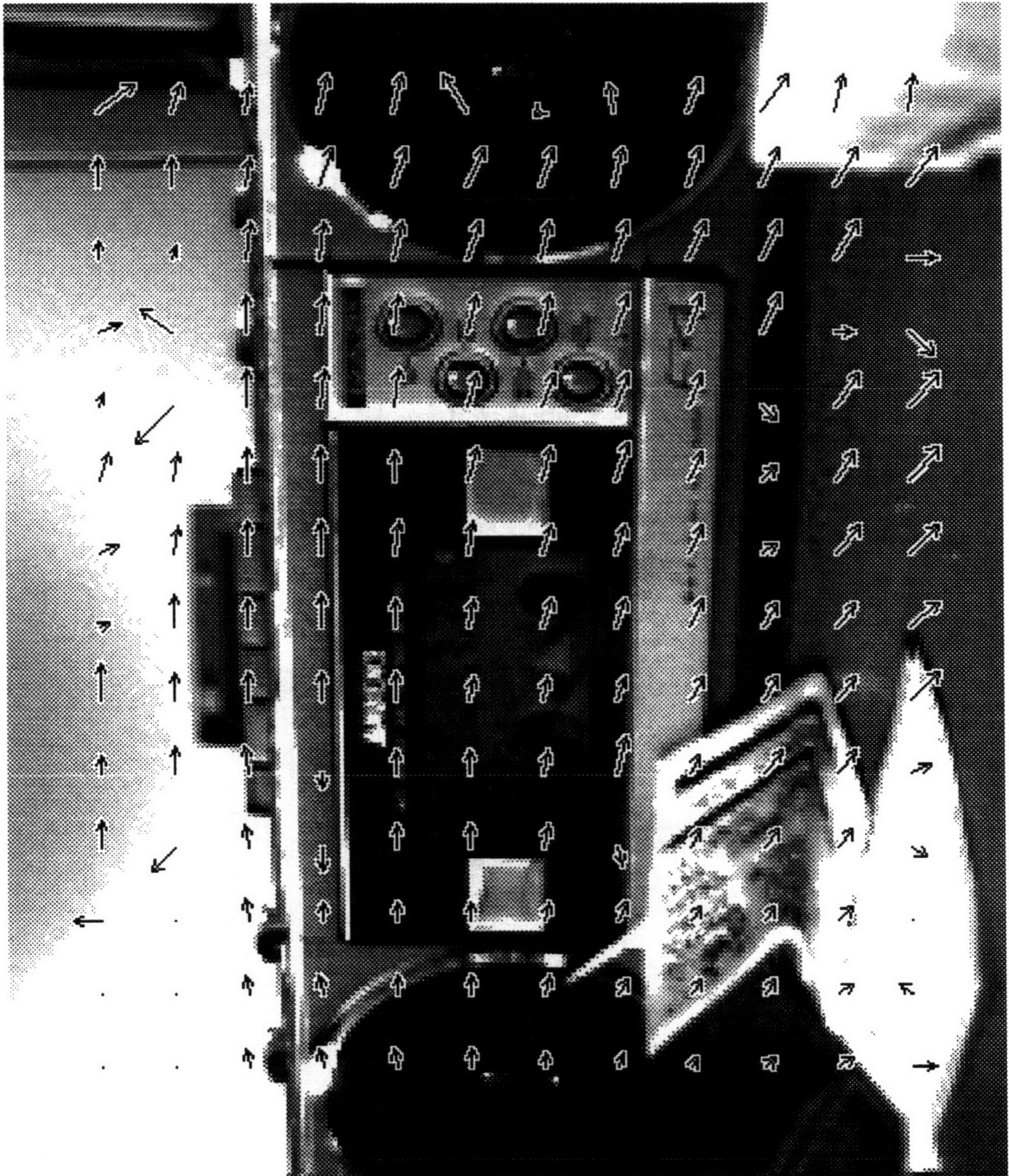


Figure 3.13 Overlay Flow for Radio Images

Images of Fred's Chair



Figure 3.14 First Image of Fred's Chair



Figure 3.15 Second Image of Fred's Chair

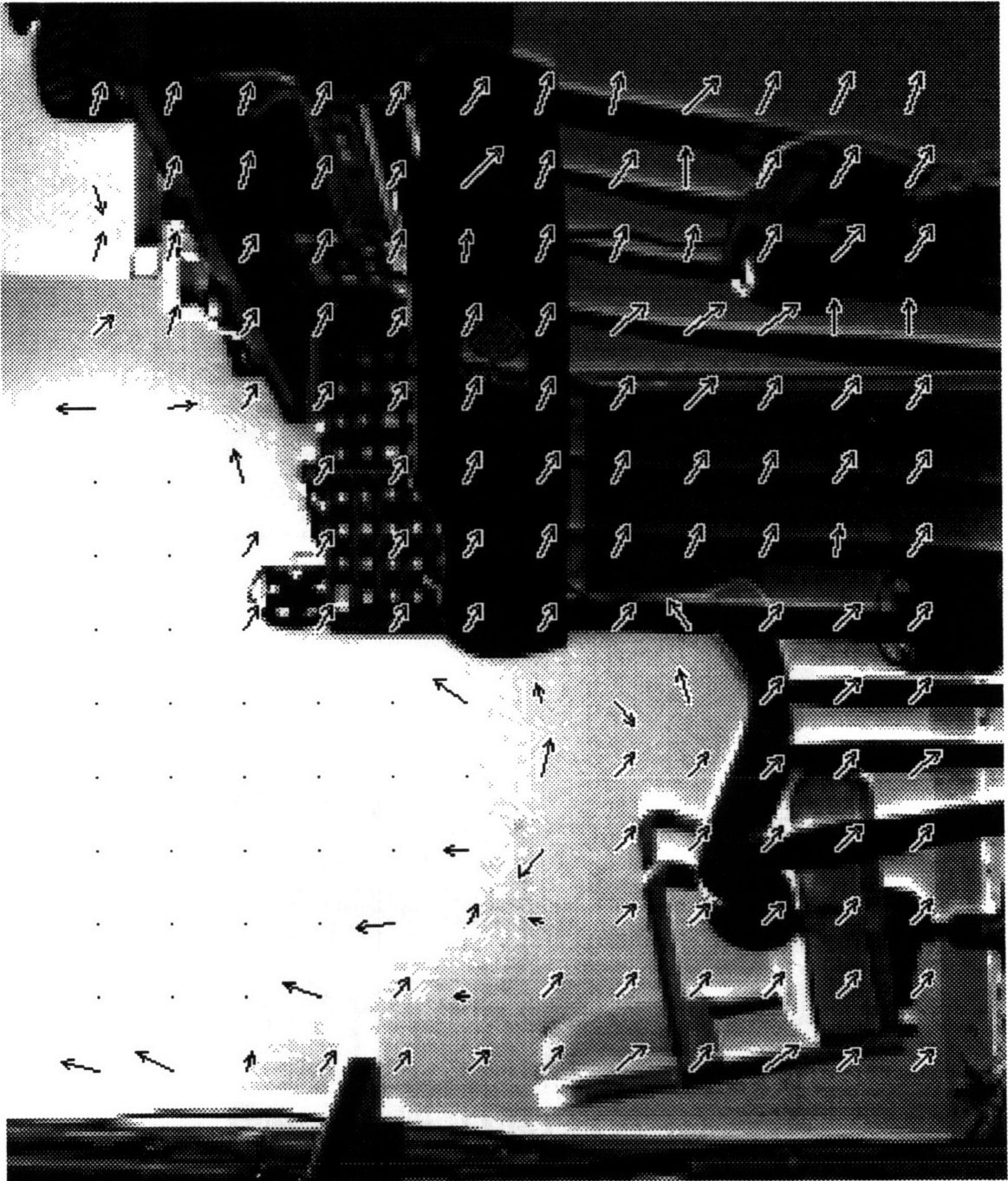


Figure 3.16 Overlay Flow for Fred's Chair Images

Size of Data Block Tests

The next tests dealt with choosing the size of the data block. The choice of size for the data block is completely arbitrary, as is the size of the search window. The data block and search window sizes used previously are one of the two combinations that can be implemented on the LSI Logic Motion Estimation Processor. Since the matching algorithm needs to be performed in hardware if the vision system is to work in real-time, the choice of data block and search window sizes were constrained to the two that can be implemented on this particular integrated circuit chip. The LSI Logic Motion Estimation Processor can perform the matching algorithm on 16 x 16 data blocks using 32 x 32 search windows, or on 8 x 8 data blocks using 16 x 16 search windows.

The optical flow for the two different size data blocks was calculated from a pair of images taken in a computer room at Draper Laboratory. The motion of the camera between images was a translation along the optical axis. The camera was not calibrated so the optical flow that was obtained was not checked against the motion field. The two images and the optical flow for the two different sizes of data blocks are shown in Figures 3.17-3.20.

The results show no advantage in computing more flow vectors with smaller data blocks because the percentage of incorrect flow vectors is not reduced. The optical flow from the 16 x 16 data block matching is as good if not better than the optical flow from the 8 x 8 data block matching. Although the 8 x 8 data block matching yields more flow vectors, it also produces more erroneous flow vectors which will have to be eliminated.

The time constraint also influences the decision to choose the larger data blocks. Although the LSI Logic Motion Estimation Processor can process the smaller data blocks faster than the larger data blocks by a factor of about ten, there are four times as many flow vectors to process. This increases the number of computations that must be performed by the

final microprocessor that finds the motion parameters via the Least-Squares minimization.

TwoTires Images



Figure 3.17 First TwoTires Image



Figure 3.18 Second TwoTires Image

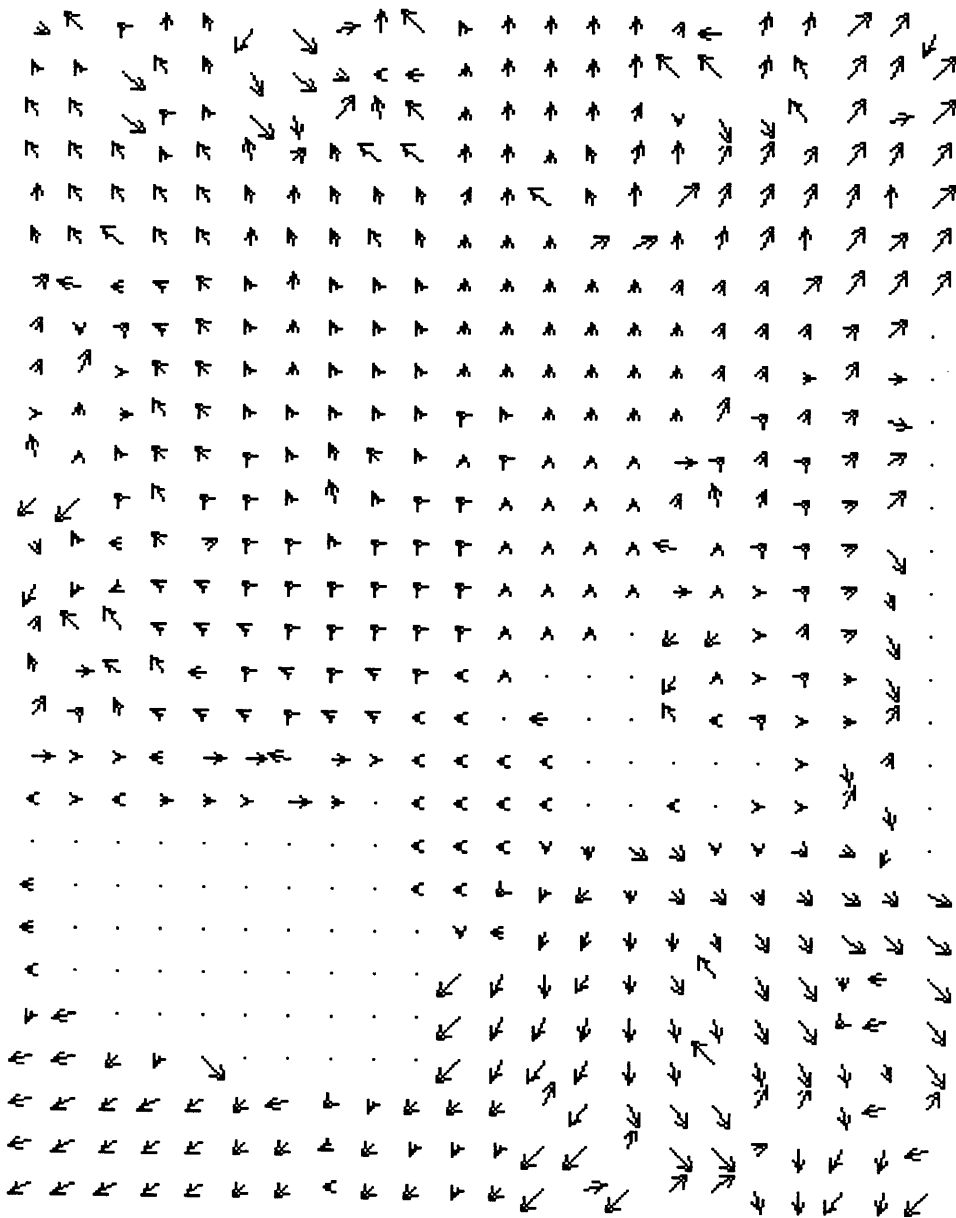


Figure 3.19 Optical Flow for 8 x 8 Data Blocks

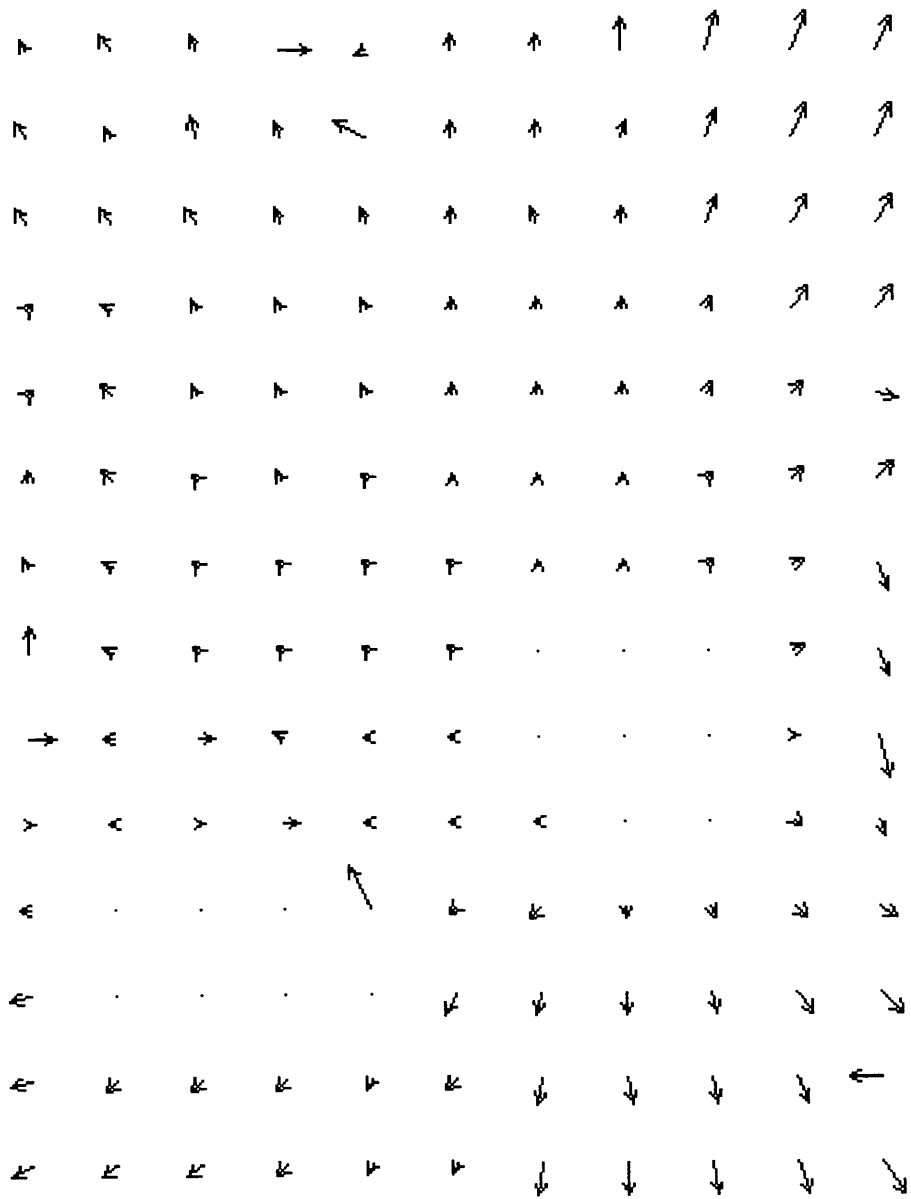


Figure 3.20 Optical Flow for 16 x 16 Data Blocks

Erroneous Flow Vectors

In the optical flows obtained from performing the matching algorithm over an entire image, there will be some incorrect flow vectors because some data blocks will match some other position better than the correct position. In the TwoTires image pair, there are incorrect flow vectors on the floor, on the wall, and in the black area on the right.

Incorrect flow vectors usually indicate that the data block being matched has no distinct pattern and thus can be placed at several positions within the search window with similar matching errors. The incorrect flow vectors occur when the data block is nearly all one shade of gray with no distinct bright spots. The incorrect flow vectors in the TwoTires optical flow occur in the all white and all black areas of the image.

Since the number of these incorrect flow vectors is usually small compared to the total number of flow vectors, they will have little effect on the final motion estimation because the Least-Squares minimization finds the best estimate over all the flow vectors. An incorrect vector is determined by its orientation compared to the general optical flow pattern. In the TwoTires matching, 26 of the 154 flow vectors are incorrect. The 128 correct flow vectors will have a greater influence on the motion estimate since they have a larger effect on the sum of squared errors than the 26 incorrect flow vectors.

However, in some environments, the number of incorrect flow vectors may be large compared to the total number of flow vectors. For example, the walls in a hallway may be painted a solid color and thus have very few distinct patterns. The flow vectors obtained from the walls will be very error-prone. Since the walls represent a large portion of the image, the number of correct flow vectors may not be much greater than the number of incorrect flow vectors.

Therefore, some weighting scheme should be used to attenuate the incorrect flow vectors. The results of the Least Squares minimization can be improved by reducing the effect of the incorrect vectors when they

represent a large portion of the total flow vectors. Placing more emphasis on the correct flow vectors relative to the incorrect flow vectors will force the Least-Squares solution closer to the motion described by the correct flow vectors.

The Least Squares Solution with Weights

The Least Squares solution presented in Chapter 2 is easily extended to allow for weighting the errors between the estimated and calculated motion fields. Each of the errors is weighted to reflect the quality of the flow vector obtained from the matching algorithm. The weighted sum of squared errors is:

$$\text{SSE} = \sum_x \sum_y W_{xy} \left((\hat{u} - u)^2 + (\hat{v} - v)^2 \right)$$

After differentiating the sum of squared errors with respect to the translation and rotation vectors, the Least Squares solution becomes:

$$\begin{bmatrix} \sum_x \sum_y W_{xy} \frac{\mathbf{S}}{Z^2} & \sum_x \sum_y W_{xy} \frac{\mathbf{Q}}{Z} \\ \sum_x \sum_y W_{xy} \frac{\mathbf{Q}^T}{Z} & \sum_x \sum_y W_{xy} \mathbf{R} \end{bmatrix} \begin{bmatrix} \mathbf{t} \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \sum_x \sum_y W_{xy} \frac{\mathbf{D}}{Z} \\ \sum_x \sum_y W_{xy} \mathbf{E} \end{bmatrix}$$

For a few extra calculations, the camera motion estimates can be improved because the weights force the estimates closer to the higher-quality flow vectors. An error in the flow vector error equation for a flow vector with a large weight will significantly affect the sum of squared errors. However, an error in the flow vector error equation for a flow vector with a small weight will have little effect on the sum of squared errors.

Error Weights

The error weights were the first weighting scheme considered because the LSI Logic Motion Estimation Processor (MEP) outputs the match errors after a matching calculation is completed. The offset to the best match

position, the best match error, and the zero offset error are output immediately from the MEP after the matching calculation is completed. In addition, all the errors from the other offset positions can be accessed after a calculation is completed by addressing the MEP.

The first error weight considered was the ratio of the zero offset error to best match error. The zero offset position is the position of the data block in the first image. The "zero-to-best" ratio highlights flow vectors whose best match error is much less than the zero offset error. The zero-to-best weighting results are shown in Figure 3.21 for the TwoTires images.

The results from the zero-to-best weights can be interpreted as follows. The number beside the flow vector is the ratio of the zero offset error to the best error. Larger numbers mean that the match error has been significantly reduced by moving the data block from its original position to the best match position. As can be seen in Figure 3.21, the weights for the tires are around 3-6. However, the weights for the incorrect flow vectors on the floor vary from 14 to 174. This shows that this weighting scheme highlights the incorrect flow vectors over the correct flow vectors. Thus, the zero-to-best weights actually do the opposite of what we want!

The next weight considered was the "average-to-best" ratio, which highlights flow vectors whose best match error is much less than the average match error. The average-to-best scheme was developed to eliminate the randomness in the zero position error, which could inadvertently enhance or reduce the zero-to-best error. The average-to-best results are shown in Figure 3.22. Here, larger numbers mean that the best match position is significantly better than other positions.

The results for this weighting scheme are even worse because the incorrect flow vectors are even further enhanced. The weights for the tires and spools of wire are around 3-6. However, the weights for the floor are in the thousands! Additionally, the weights for the white wall in the background are now very significant since 2 of the weights are over 100.

Neither the zero-to-best weighting scheme nor the average-to-best weighting scheme was successful because each highlighted incorrect flow vectors instead of attenuating them. For very indistinct areas of an image such as walls or floors, a data block can have a best match error near zero. However, if this indistinct area borders an area with some distinction and this distinct area is included in the search window, the zero error or average error may not be near zero. As a result, the zero-to-best and average-to-best ratios will be large. Thus, the zero-to-best and average-to-best ratios cannot be used to weight the flow vectors because they highlight the indistinct areas of an image - the very regions that we want to attenuate!

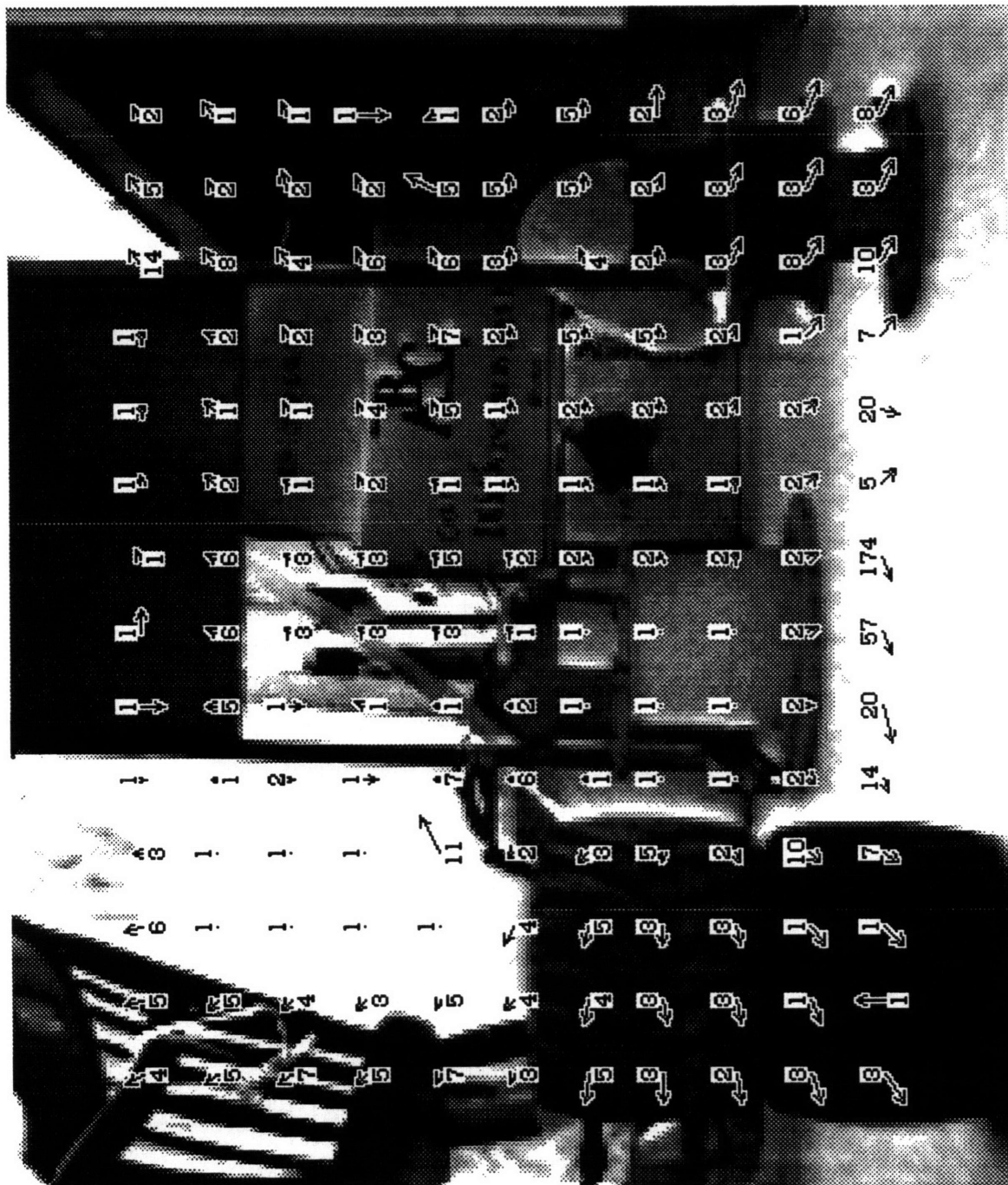


Figure 3.21 Zero-to-Best Weights



Figure 3.22 Average-to-Best Weights

Variance Weights

The variance weights, as well as the subsequent cosine weights, depend upon neighboring flow vectors. The nearest neighbors to each flow vector are 16 pixels away, either up, down, left, or right. These were determined to be too far away to be considered a neighbor. Therefore, another series of matchings was performed to calculate flow vectors for data blocks that are staggered 8 pixels up or down, and 8 pixels left or right of the original data block. The flow vectors obtained from the original matching are called original flow vectors, and the flow vectors obtained from the new matching are called staggered flow vectors. The four staggered data blocks overlap the original data block and meet at its center as shown in Figure 3.23. Figure 3.24 shows the optical flow for the data block flow vectors, and Figure 3.25 shows the optical flow for the staggered flow vectors.

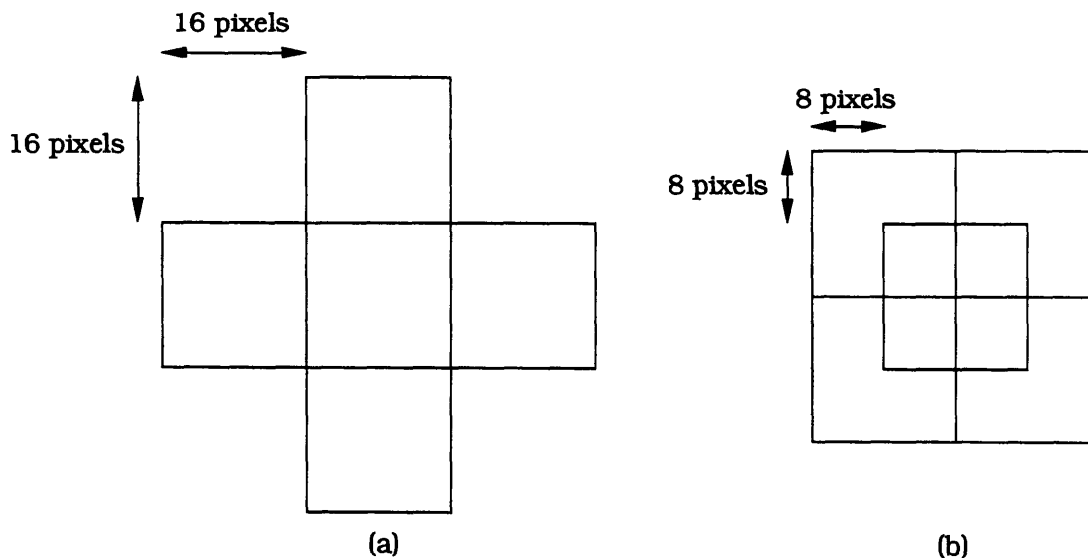


Figure 3.23 (a) Neighbor data blocks are too far away and don't overlap. (b) Staggered data blocks are closer and overlap the original data block.

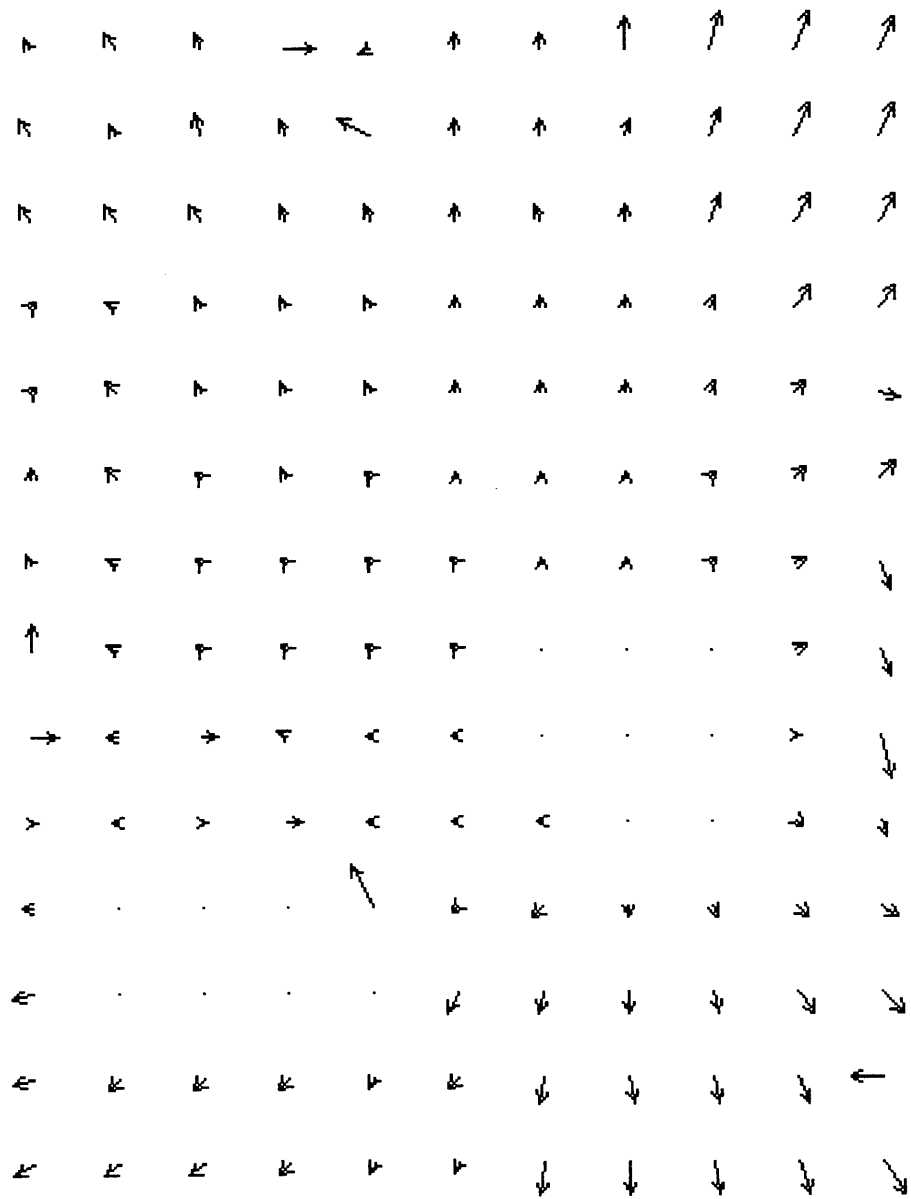


Figure 3.24 Optical Flow for Original Data Blocks

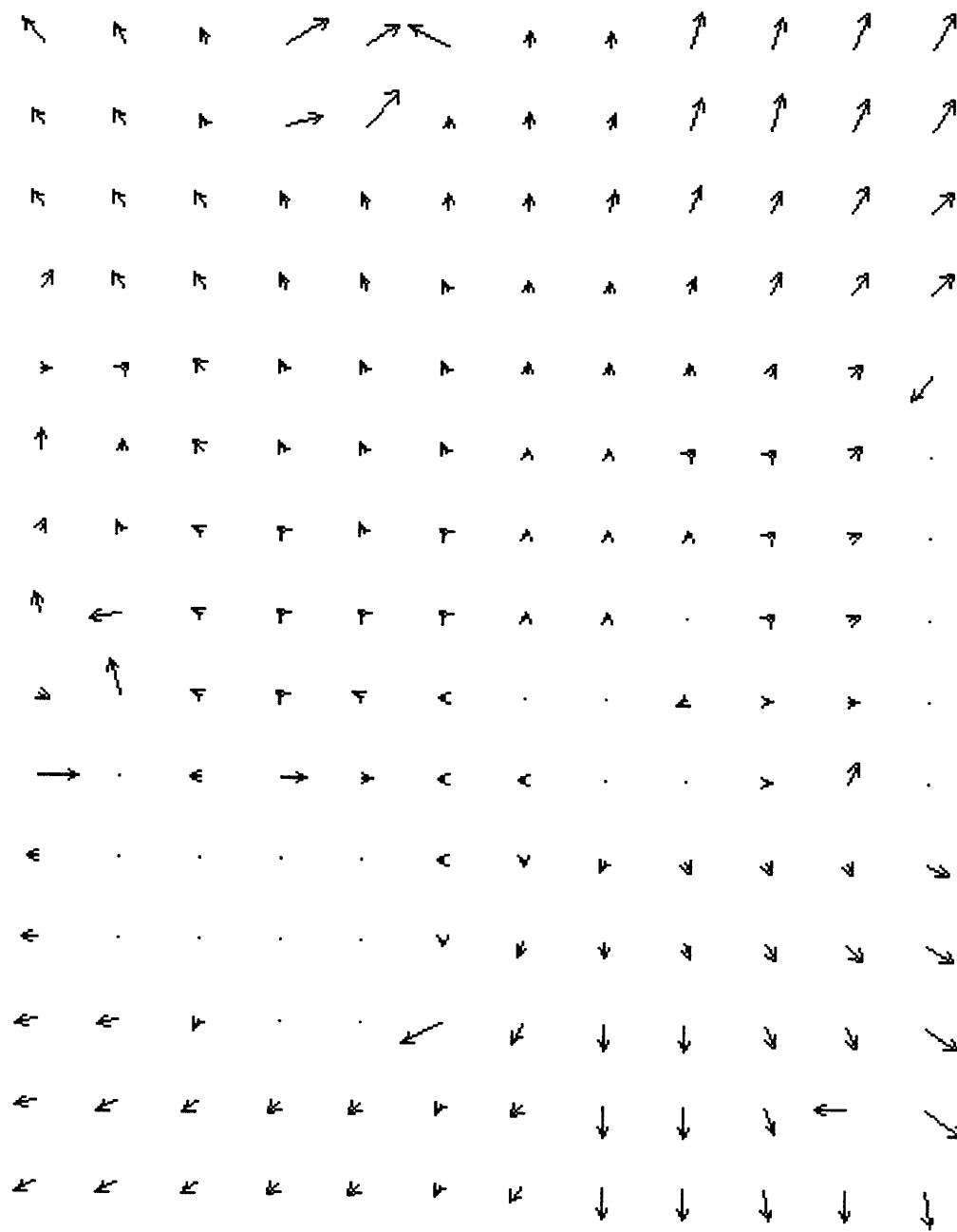


Figure 3.25 Optical Flow for Staggered Data Blocks

An average flow vector and a variance can be calculated from the original flow vector and the 4 staggered flow vectors around it. The average flow vector is simply the average of the five flow vectors. The sample variance can be easily computed from the u and v components of the five flow vectors. The sample variance is:

$$\text{Sample Variance} = \frac{\sum u^2}{5} - \bar{u}^2 + \frac{\sum v^2}{5} - \bar{v}^2$$

An "inverse variance" weight can be calculated for each data block by taking the inverse of the sample variance. This weight will highlight those areas where the variance is small and attenuate those areas where the variance is large. Results from applying the inverse variance weights are shown in Figure 3.26, where the weights have been scaled by 100.

Although they attenuate some of the incorrect flow vectors, the inverse variance weights do not work well because they tend to highlight short flow vectors. The incorrect flow vectors in the floor are weighted much less than most of the surrounding correct flow vectors. However, the short flow vectors on the white wall in the background, as well as other short, correct flow vectors, are weighted much higher than the weights on the tires.

The reason for this is that the short flow vectors tend to have smaller variances than the longer flow vectors. The sample variance of short flow vectors will usually be small, unless the data block is indistinct. Large flow vectors are more likely to have larger sample variances because a small angle difference between two flow vectors translates into a large vector difference.

The "flow-to-variance" weight tries to counteract this problem by using the length of the flow vector to normalize the weight. The flow-to-variance weight is the ratio of the squared length of the original flow vector to the squared length of the original flow vector plus the sample variance. The squared length of the original flow vector can be viewed as

signal power, and the sample variance is similar to noise. The flow-to-variance weights range from 0 to 100 and attempt to highlight flow vectors whose length is large compared to the variance. The results from the flow-to-variance weighting method are shown in Figure 3.27, where the ratios have been scaled by 100.

These weights are the first "good" set of weights that we have seen because they highlight correct flow vectors and attenuate incorrect flow vectors. The weights on the tires and spools are greater than those of the floor and background wall. The problem of short flow vectors being artificially weighted too high, as with the inverse variance weights, has been eliminated. However, a better weighting scheme is the "average flow-to-variance" weighting scheme.

The average flow-to-variance weight is the ratio of the squared length of the average flow vector to the squared length of the average flow vector plus the sample variance. The squared length of the average offset vector represents the average signal power in a small region. These weights were created because incorrect original flow vectors can sometimes be quite large, although the average flow vector is much smaller. These weights also range from 0 to 100 and attempt to highlight flow vectors around which the average flow vector is large compared to the variance. The results of this weighting scheme are shown in Figure 3.28.

These weights improve upon the results of the flow-to-variance weights because incorrect flow vectors are further attenuated without affecting the enhancement of the correct flow vectors. The weights on the floor and the wall have been reduced. However, the weights on the tires and spools as well as the other correct weights in the background have not been significantly affected. This method is the best of the variance weights because it eliminates the artificially high weighting of the short flow vectors found with the inverse variance weights. Additionally, the sensitivity to the original flow vector length found with the flow-to-variance weights has been reduced.

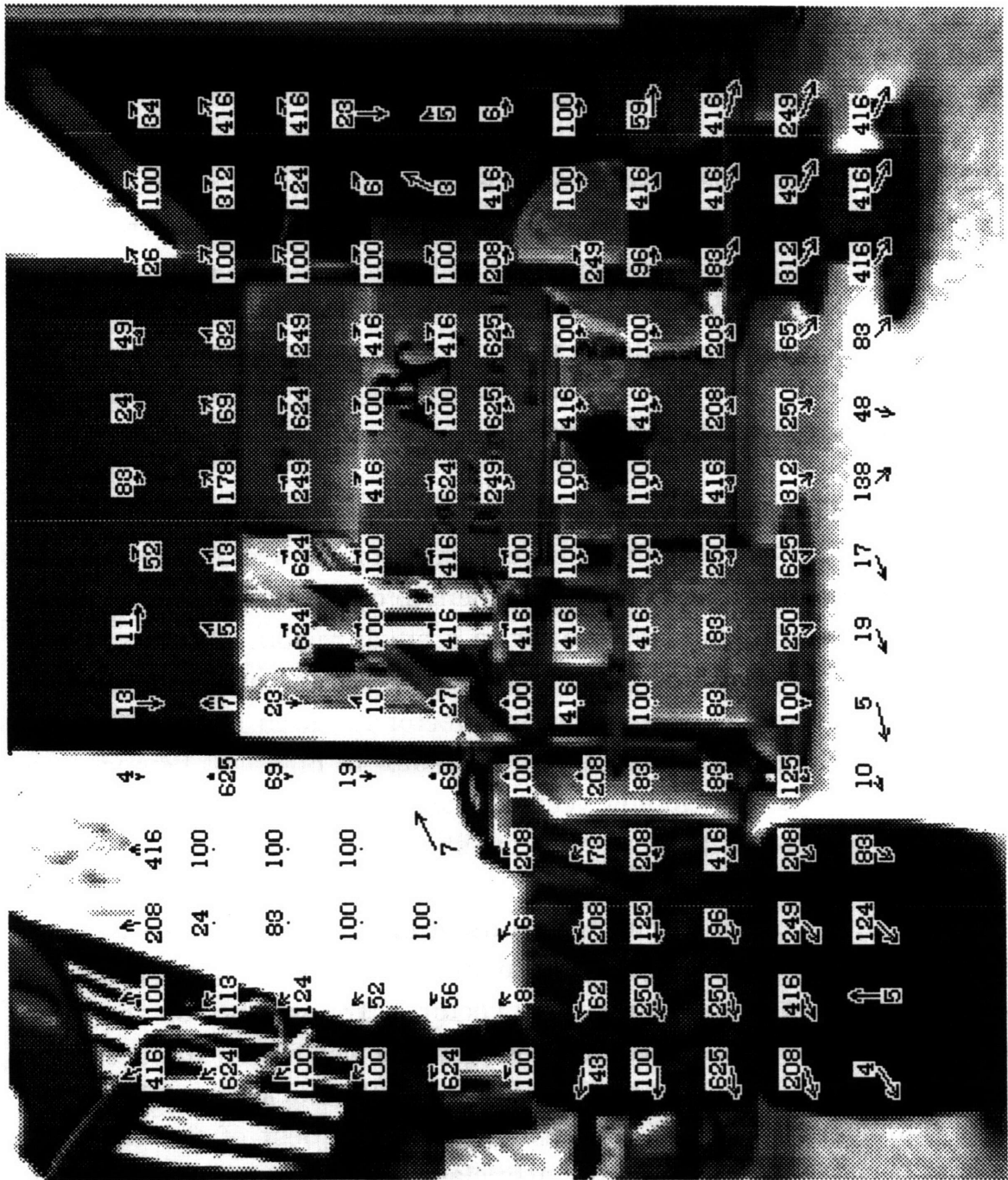


Figure 3.26 Inverse Variance Weights

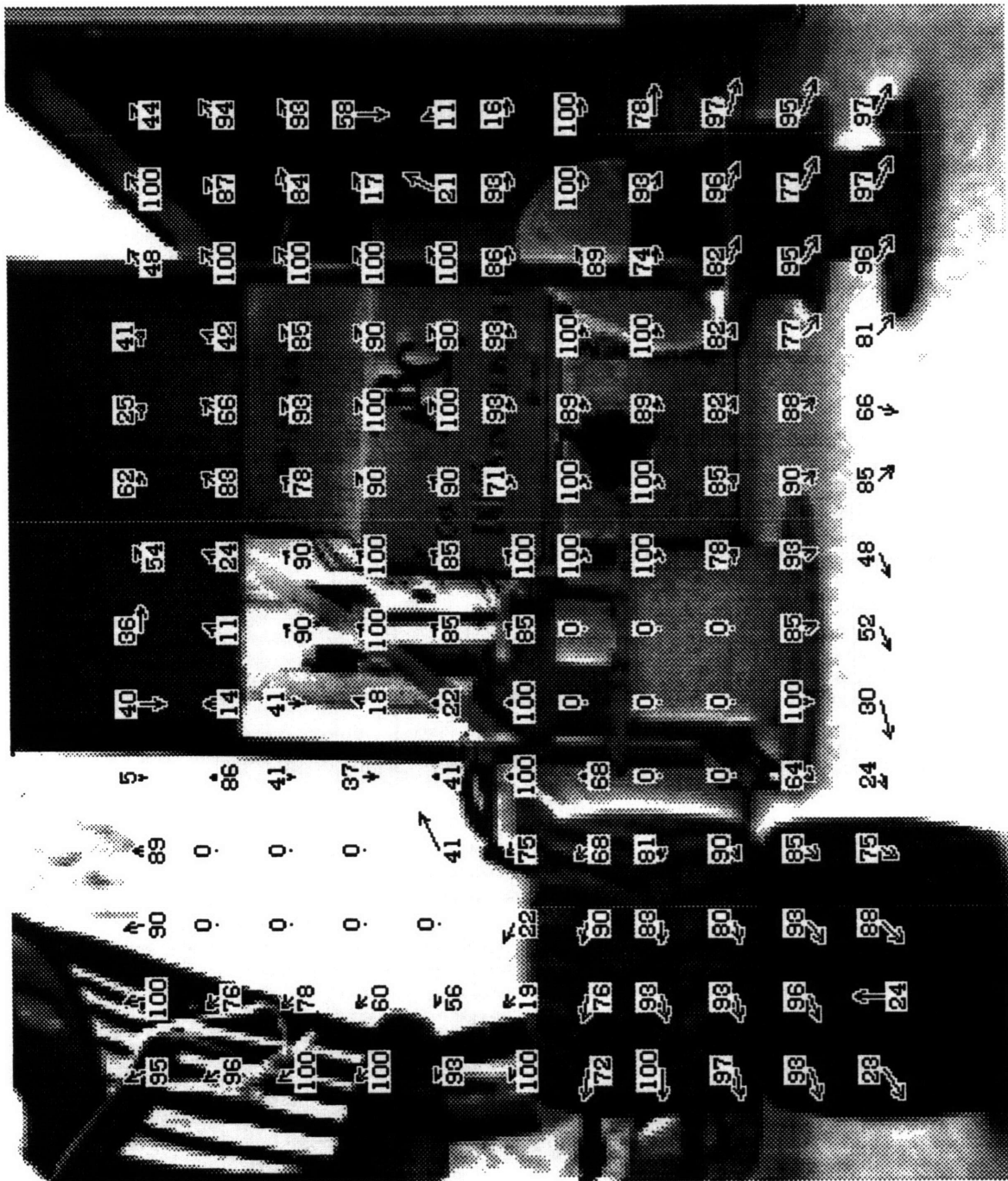


Figure 3.27 Flow-to-Variance Weights

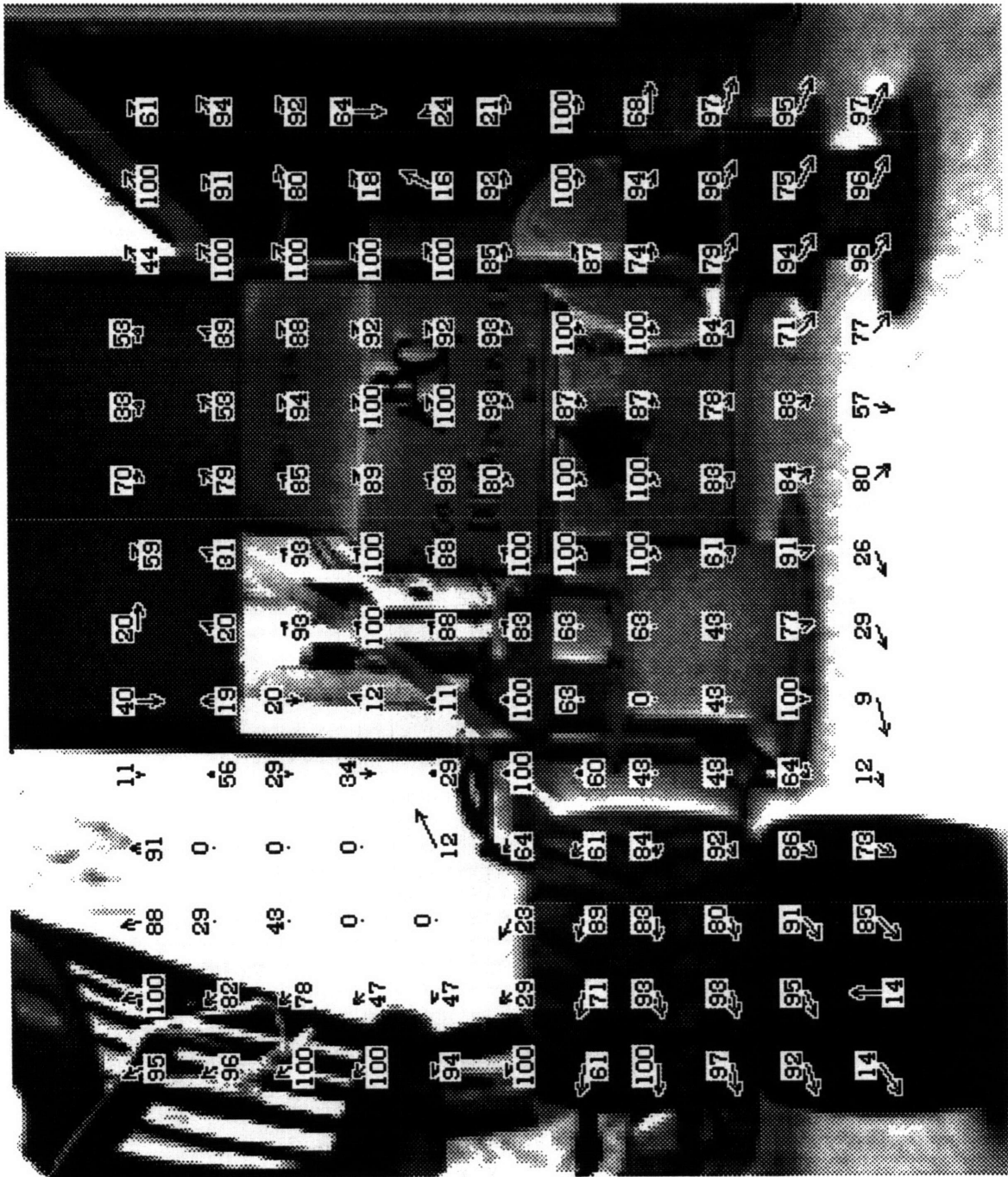


Figure 3.28 Average Flow-to-Variance Weights

Cosine Weights

The cosine weights were designed to exploit the alignment between an original flow vector and the 4 staggered flow vectors around it. Two flow vectors that point in the same direction are well aligned. It was found by looking at the images that the alignment between an incorrect original flow vector and any one of the staggered flow vectors around it is very poor. Thus, some method that brings out this characteristic seems appropriate.

The dot product operation was used as a measure of alignment because it ranges from one for perfect alignment to zero for orthogonal alignment. The dot product of unit normalized vectors is simply the cosine of the angle between them. For two unit vectors, the dot product is:

$$\hat{\mathbf{f}}_1 \cdot \hat{\mathbf{f}}_2 = \frac{u_1 u_2 + v_1 v_2}{\sqrt{u_1^2 + v_1^2} \sqrt{u_2^2 + v_2^2}} = \cos(\theta)$$

The dot product operation can also be interpreted as the projection of one of the vectors on the other.

The "sum of cosines" weighting scheme attempts to represent the average projection of the four staggered unit flow vectors onto the original unit flow vector. The projection of one of the staggered flow vectors onto the original flow vector is simply the dot product of the two vectors. This amounts to the cosine of the angle between the vectors. The average projection is calculated by taking the average of the four projections of the staggered unit flow vectors onto the original unit flow vector. This is the sum of the dot products of the original flow vector with each of the four staggered flow vectors around it divided by 4:

$$\text{Sum of Cosines} = (\hat{\mathbf{f}}_O \cdot \hat{\mathbf{f}}_{S1} + \hat{\mathbf{f}}_O \cdot \hat{\mathbf{f}}_{S2} + \hat{\mathbf{f}}_O \cdot \hat{\mathbf{f}}_{S3} + \hat{\mathbf{f}}_O \cdot \hat{\mathbf{f}}_{S4}) / 4$$

The results of this weighting scheme are shown in Figure 3.29 where the weights have been scaled by 100. This weighting scheme represents a major improvement over previous methods because almost all incorrect flow vectors have a zero weight. However, a few incorrect vectors still

remain in the white area at the bottom of the image. The correct flow vectors which are located on the tires, the spools of wire, and in the background have weights that are close to 100.

The "product of cosines" weighting scheme requires that all the unit flow vectors be consistent rather than just the average. Instead of adding the four cosines, the four cosines are multiplied:

$$\text{Product of Cosines} = \hat{\mathbf{f}}_O \cdot \hat{\mathbf{f}}_{S1} \times \hat{\mathbf{f}}_O \cdot \hat{\mathbf{f}}_{S2} \times \hat{\mathbf{f}}_O \cdot \hat{\mathbf{f}}_{S3} \times \hat{\mathbf{f}}_O \cdot \hat{\mathbf{f}}_{S4}$$

If one of the staggered flow vectors is orthogonal to the original flow vector, the weight is zero regardless of the other staggered flow vectors. Therefore, there must be a higher degree of alignment between the original flow vector and the four staggered flow vectors around it. A weight near one means the all five flow vectors are pointing in the same direction, while a weight near zero means that one or more of the staggered flow vectors are orthogonal to the original flow vector. The results of this weighting method are shown in Figure 3.30, where the weights have been scaled by 100. The cosine in this method is bounded between zero and one, so cosines less than zero are set equal to zero.

This method produces the best weights since the correct flow vectors are highlighted and almost all the incorrect flow vectors are attenuated. The only incorrect flow vector with a significant contribution is the "66" on the right hand side of the image. The incorrect flow vectors on the floor and on the wall have been assigned a zero weight.

The product of cosines weights have been applied to two other pairs of images - the Radio images and Fred's Chair images. The overlay flows for these can be found in Figures 3.31 and 3.32. Similar to the TwoTires results, the product of cosines weights attenuate the incorrect flow vectors.

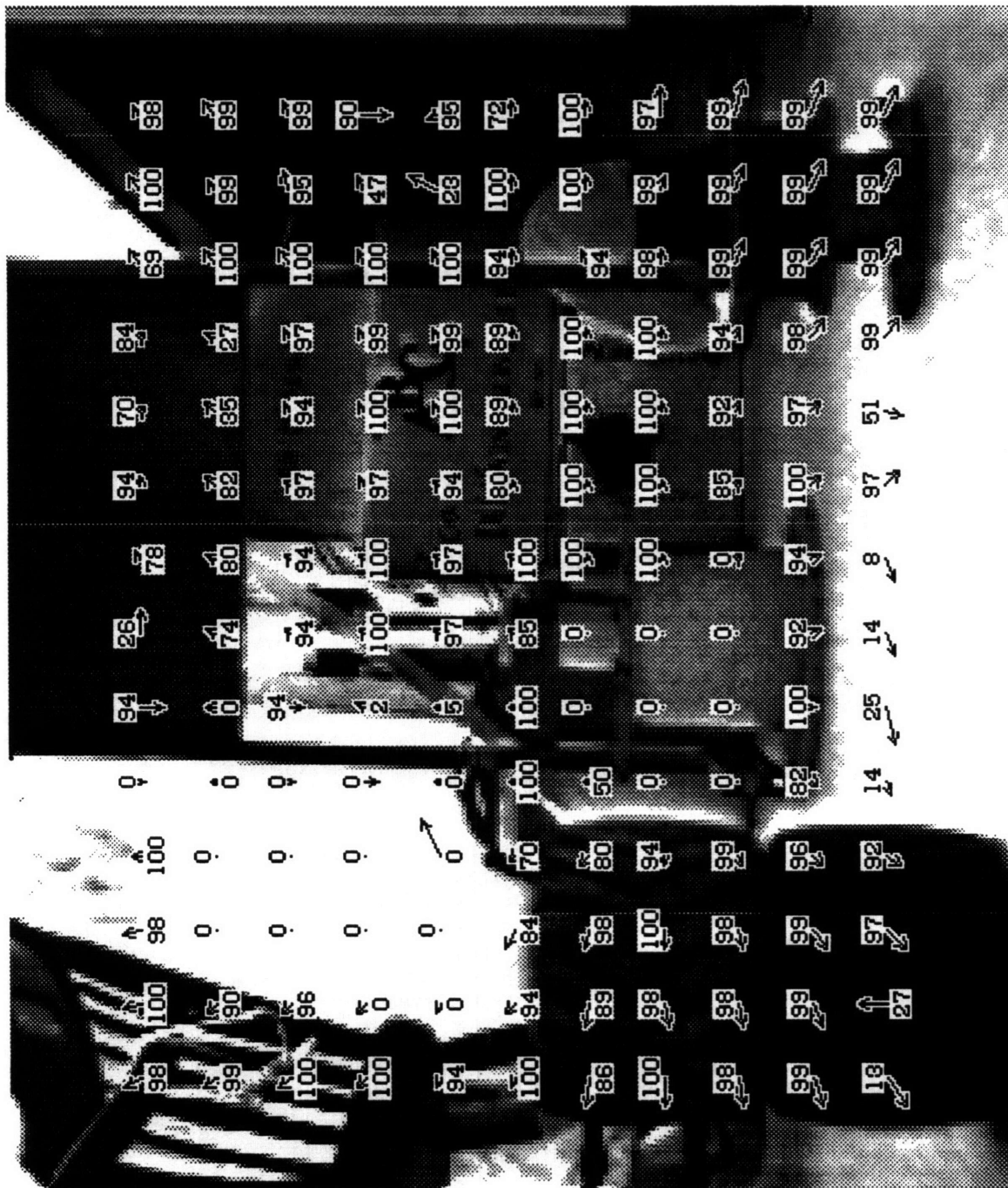


Figure 3.29 Sum of Cosines Weights

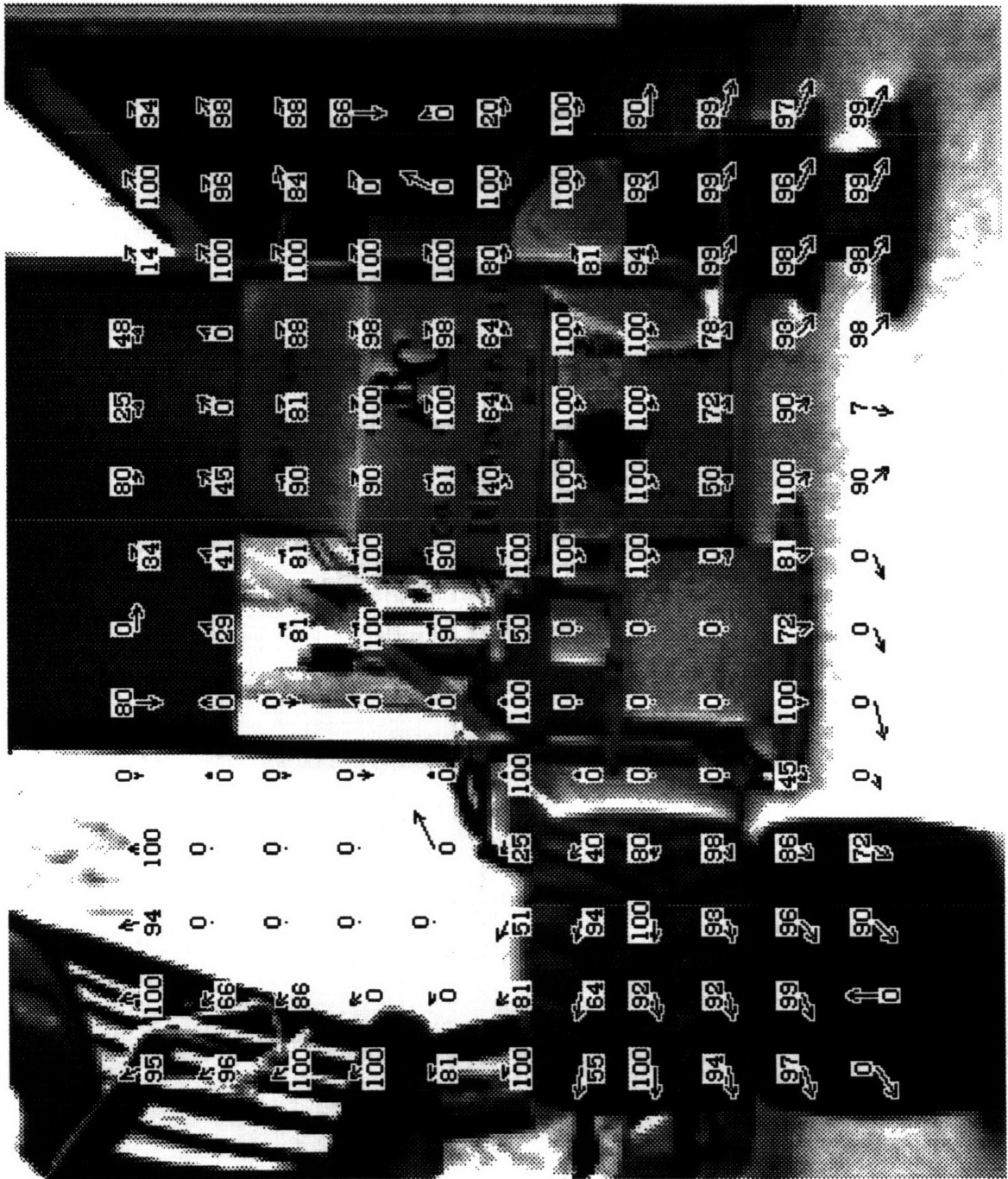


Figure 3.30 Product of Cosines Weights

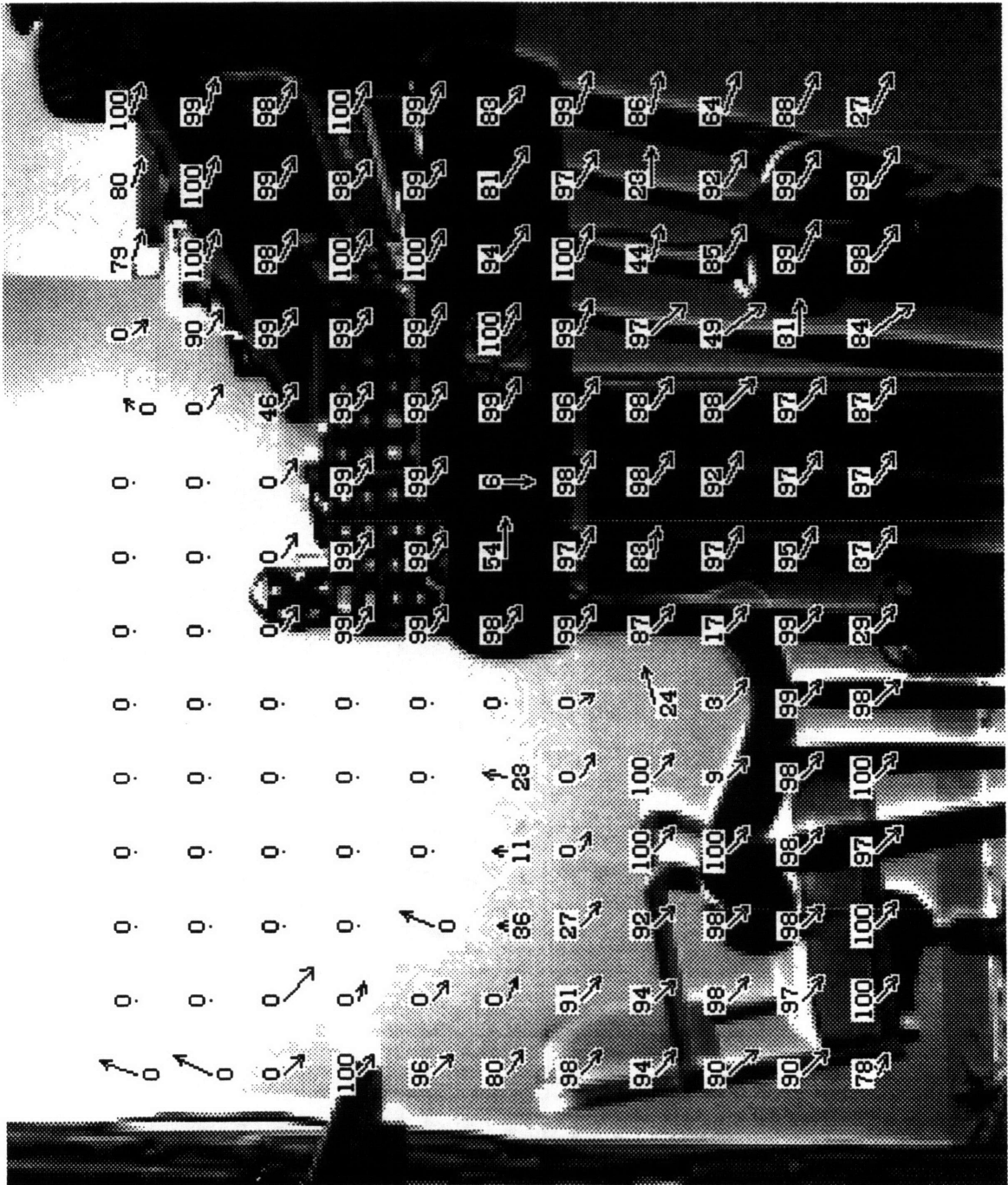


Figure 3.32 Product of Cosines Weights for Fred's Chair Images

Implementing the Product of Cosines Weights

Implementing the product of cosines weights in the vision system requires additional processing to be performed by the LSI Logic Motion Estimation Processor and the microprocessor that converts the match results into motion estimates. The MEP must calculate approximately twice as many best matches. The microprocessor must calculate weights to apply to the Least Squares equations.

Calculating Neighbor Flow Vectors

The staggered flow vectors can be calculated in the same manner as the original flow vectors. The staggered data blocks are offset from the original data blocks by 8 pixels up or down and 8 pixels left or right. If the original flow vectors were obtained by performing the matching algorithm from (16h,20v) to (224h,180v), the staggered flow vectors can be calculated by performing the same matching algorithm from (8h,12v) to (232h,188v).

The LSI Logic Motion Estimation Processor must now perform twice as many matching calculations. The entire image will have to be processed once to obtain the original flow vectors and a second time to obtain the staggered flow vectors. The MEP can process about 52 images/sec. Doubling the amount of matchings amounts to an effective rate of 20 images/sec, which is still well within range. Additionally, the processing rate of 10 images/sec is expected to be reduced because it was found that a slower rate is adequate.

Calculating the Weights

Calculating the weights to apply to the Least Squares equations imposes an additional burden of 8,428 computations per image pair on the microprocessor. Each weight requires 43 computations as shown in Table 3.1. For a 256 x 256 image, there are 196 data blocks, and hence 196 weights that must be calculated.

Table 3.1 Computations to Calculate a Product of Cosines Weight

Calculations	Mult	Div	Add	Sub	Sqrt
5 Vector lengths	10		5		5
4 Unit dot products	8	8	4		
Product of Cosines	3				
Total	21	8	9		5

Adding these calculations to those required to generate the 6 x 6 matrix, solve the matrix, and update the depth values, one can revise the microprocessor considerations discussed in Chapter 2. Each pair of images now requires 24,510 computations to estimate the motion from the matching results. Adding a safety margin of 50% brings the total number of computations to 36,765. Assuming that we still want to produce 10 motion estimates per second and that each iteration requires 10 steps to converge, the microprocessor will have 4.0 clock cycles per computation.

Table 3.2 Microprocessor Reconsiderations

Clock Speed	30 MHz
Duty Ratio	50%
Effective Clock Speed	15 MHz
Number of Computations	3,676,500
Cycles per Computation	4.0

Although this seems to be pushing the microprocessor, the estimate is very conservative. There are two margins for safety, namely the extra 50% to the number of computations and the duty ratio of the microprocessor. Additionally, we will see later that the number of motion estimates per second is closer to 3 or 4, and the number of steps for convergence of the iteration is closer to 5.

Chapter Summary

This chapter has presented the initial testing of the Constant Pattern Assumption via the matching algorithm. The single data block tests have shown that the matching algorithm can find the correct match even when the camera has undergone significant translation and rotation, i.e. when the pattern has been magnified and twisted. The optical flows found by applying the matching algorithm to entire images show that the Constant Pattern Assumption is valid. Finally, no advantage was found in using smaller data blocks to obtain more flow vectors.

Various weighting schemes to attenuate incorrect flow vectors and highlight correct flow vectors were explored in this chapter. Error weights were tested and found to incorrectly highlight data blocks that had very indistinct, if any, patterns because the best match error could be close to zero while the zero position or average error was non-zero. Variance weights were tested and found to be better than error weights because they did not incorrectly highlight indistinct patterns. However, the product of cosines weight was found to be better than variance weights because more incorrect flow vectors were eliminated. Table 3.3 summarizes the various weights tested in this chapter.

Table 3.3 Summary of Flow Vector Weights

<i>Weight</i>	<i>Comments</i>
Zero-to-Best	Enhances Incorrect FV's Over Correct FV's
Average-to-Best	Worse than Zero-to-Best
Inverse Variance	Some Incorrect FV's Attenuated, but Short FV's Highlighted
Flow-to-Variance	Eliminates Short FV Problem, but Incorrect FV's Still Present
Ave Flow-to-Var	Attenuates Incorrect FV's Better than Flow-to-Variance
Sum of Cosines	Correct FV's Enhanced and Most Incorrect FV's Attenuated
Product of Cosines	Correct FV's Highlighted and All Incorrect FV's Attenuated

The final tasks to complete the testing of this vision system are to test the motion estimation equations and the use of the depth maps in obstacle avoidance. The Least-Squares equations for motion should produce reasonable estimates of vehicle motion that can be used for dead-reckoning navigation. Additionally, one should be able to interpret the depth values in a way that obstacles can be recognized and avoided.

Chapter 4 Motion Estimation

In this chapter, the Least-Squares motion estimation equations are tested. These equations were implemented in TwinScreen™, and the results from several motion estimations are presented. Both a planar and a contoured surface were analyzed, and it was discovered that the 6 degree of freedom model for the Micro-Rover's motion cannot be used because it is impossible to distinguish lateral motion from rotation. A 4 degree of freedom model for the Micro-Rover's motion is suggested as a good approximation. The new Least-Squares equations were tested on the same 2 pairs of images, and the results indicate that this model will adequately represent the Micro-Rover's motion.

Implementing the Least-Squares Equations in TwinScreen™

Implementing the motion estimation equations requires some knowledge of the camera characteristics since they are used to obtain the normalized coordinates of the pixels. The Longuet-Higgins and Prazdny equations contain x , y , u , and v that are normalized to the principal distance. Thus, the principal distance is required. Additionally, the principal point is required because the image is simply an array of pixels, where the center is not necessarily the principal point. The total area of the CCD array that the image represents must also be found to obtain the absolute coordinates of each pixel. The absolute pixels are then divided by the principal point to normalize them.

The camera characteristics were estimated rather than measured because the camera calibration process is quite extensive. The principal distance was estimated to be the focal length of the Computar 8.5 mm auto-iris lens. The principal point was assumed to be the center of the image grabbed from the camera by the DigiVideo frame grabbing board. The image was 256 x 216, so the principal point was estimated to be (128, 108). The area of the CCD array was about 7 mm x 5 mm. This is close to the 4:3 aspect ratio common to most video applications. It was assumed that the DigiVideo board "grabbed" the center 5 mm of the 7

mm width so that "square" pixels were obtained. Thus, the area of the CCD array was estimated to be 5 mm x 5 mm.

If the estimated values of the camera characteristics differ significantly from the actual values, the motion estimates and the depth map will contain errors. One type of error occurs when either the principal distance or the area of the CCD array is incorrect. An error in either estimate affects the angle that a ray to a scene point makes with the optical axis. This will either compress or expand the scene, and the resulting motion estimate will be greater or less than the actual value. Another type of error occurs when the principal point is incorrect. This will affect the motion estimate because the various scene points will appear to be farther or closer to the optical axis.

The TwinScreen™ code that was used to test the motion estimation equations is reproduced in Appendix B. The iteration for the motion is performed in steps. At each step, the user may choose to update the depth map from the current motion estimate and then compute a new motion estimate from the updated depth map. Alternatively, the user may opt to scale the depth map to simulate the vision system's response to a sensor, such as a laser range-finder, that makes some measure on the environment. Additionally, the user may enter an estimate of the vehicle's motion and then continue the iteration based upon this starting value.

Testing the Motion Estimation Equations

Along the same lines as Heel's work [7] the motion estimation equations were tested against both a planar and a contoured background. The planar background consisted of 2 posters from the Micro-Rover team's presentation at the Rover Expo in Washington, D.C. in September '92. The posters were 5 feet away from the camera and chosen because of their strong and numerous patterns. The contoured scene consisted of 2 stacked tires about 3 feet out and 1/2 foot to the left of the optical axis, and 3 spools of wire about 4 feet out and 1 foot to the right of the optical

axis. The background was the rear of a computer/ storage room at Draper Laboratory.

The camera was translated along the optical axis in both scenes. The camera moved about 1.75 inches in the planar scene. In the contoured scene, the camera was translated 2.25 inches. Before presenting the results of the testing on the 2 scenes, we must discuss how the scale factor ambiguity problem was solved.

The scale factor ambiguity problem, which is inherent in monocular vision applications, can be solved in 2 ways - either by scaling the depth map or by scaling the motion. By using a sensor that can measure the distance to various points in the environment, one can scale the depth map. For instance, on the Micro-Rover, the laser range-finder can be used to obtain the depth to various points. The resulting motion estimate will then be appropriately scaled. Alternately, one can measure the distance traveled by the Micro-Rover and scale the motion estimate. The drag wheel on the Micro-Rover can be used to measure the distance traveled. The depth map will then reflect actual distances to objects in the scene.

The results from the planar scene are pretty good, and the motion estimates are shown in Table 4.1. The translation along the optical axis was scaled to 4.4 cm, or 1.75 inches, which is the actual translation of the camera. There is a significant lateral translation in the x direction of 2.1 cm, or .83 inches. Also, there is a rotation about the y axis of -0.013 radians, or about .74 degree. These two errors are interrelated as will be shown shortly. The depth map from this motion estimate is a fairly decent representation of the planar surface and is shown in Figure 4.1.

The results from the contoured scene are not good at all. In this case, the depth map was scaled so that the 2 tires were about 3 feet, or 1 meter away. The translation was estimated at 3.5 cm, or about 1.38 inches. This is an error of about -39%. However, the lateral translation in the y direction was estimated at 12 cm, or 4.72 inches, and the rotation about the x axis was 0.118 radians, or 6.8 degrees. These

results, as shown in Table 4.2, clearly indicate that something is wrong with the Least-Squares equations.

The depth map from this contoured scene, as shown in Figure 4.2, is surprisingly flat. In fact, it is the initial depth map assumed by the motion estimation algorithm. Coincidentally, the depth to the tires and the initial constant depth were both 1 meter. Some of the values have been adjusted slightly up or down from the original 1.00 value, but not by much.

The motion estimation equations need to be examined since intuition suggests that the motion and depth map should be obtained at least to a scale factor. When the camera moves forward, the image will "flow" outward. Closer objects will move faster than farther objects, so their flow vectors will be longer. However, an average forward motion can be estimated and then used to determine whether points in the scene are far or close. Close objects will have longer flow vectors, and the corresponding depth values will be less. The new "contoured" depth map can be used to generate a "better" motion estimate. Then, the new motion estimate will be used to produce a new depth map. This iterative solution should converge to a reasonable estimate for the motion and the depth.

Table 4.1 Motion Estimates of Planar Surface Using 6 DOF's

Parameter	Estimated	Actual	Parameter	Estimated	Actual
U (m)	0.021	0.000	A (rad)	-0.004	0.000
V (m)	-0.005	0.000	B (rad)	-0.013	0.000
W (m)	0.044	0.044	C (rad)	-0.003	0.000

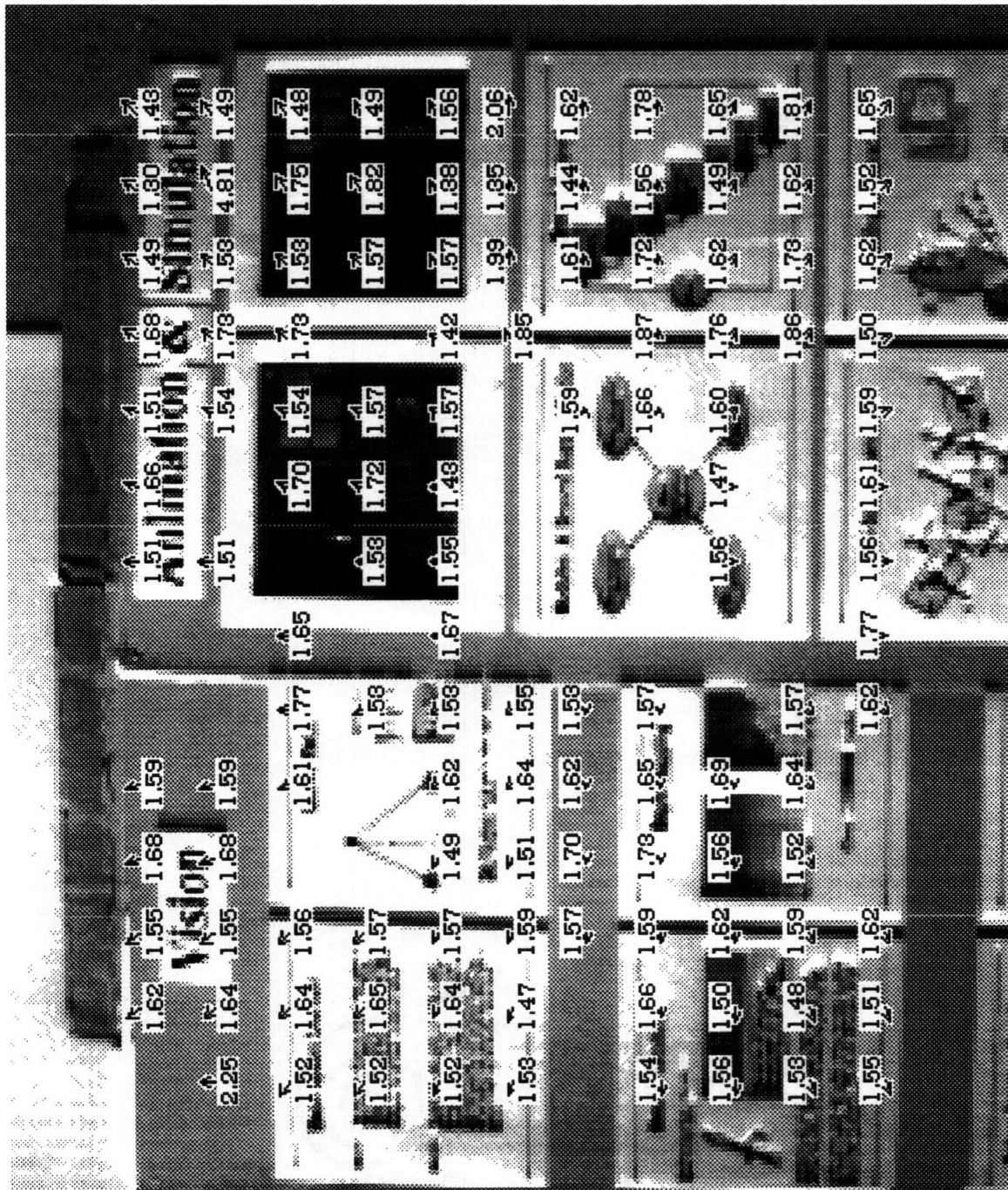


Figure 4.1 Depth Map After Solving for Motion with 6 Degrees of Freedom

Table 4.2 Motion Estimates of Contoured Surface Using 6 DOF's

Parameter	Estimated	Actual	Parameter	Estimated	Actual
U (m)	-0.021	0.000	A (rad)	0.118	0.000
V (m)	0.120	0.000	B (rad)	0.020	0.000
W (m)	0.035	0.053	C (rad)	-0.006	0.000

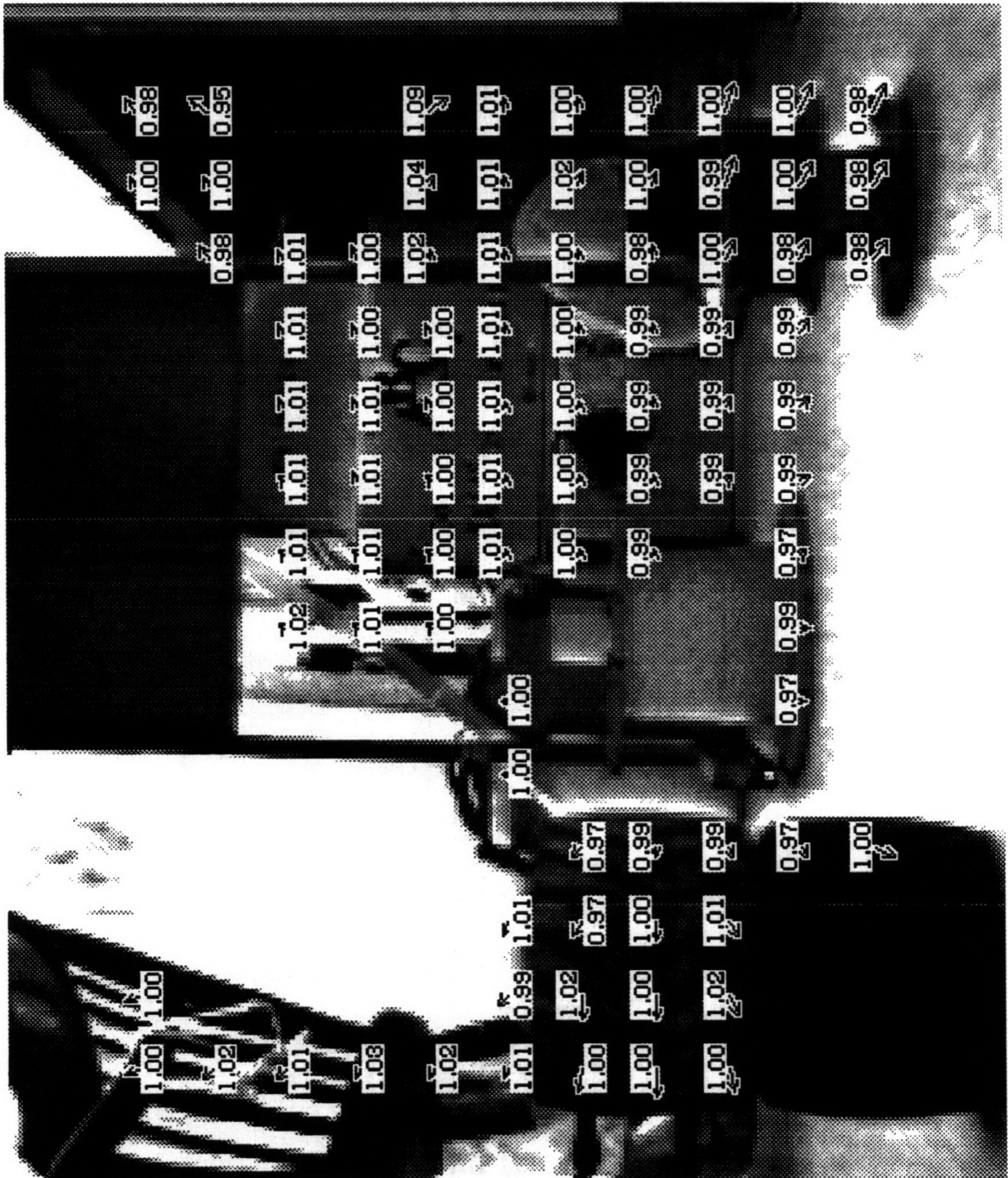


Figure 4.2 Depth Map After Solving for Motion with 6 Degrees of Freedom

To examine the Least-Squares equations, we will look at the 6 x 6 matrix generated from the data blocks of the contoured scene. The matrix is shown in Table 4.3. There are only four independent columns in this matrix. The first and fifth columns are nearly identical, as well as the second and fourth columns. The first column is related to the x component of translation, and the fifth column is related to the rotation about the y axis. The second column represents translation in the y direction, and the rotation about the x axis is represented in the fourth column.

Table 4.3 Matrix from First Motion Estimation of TwoTires Images

89.090	0	-3.2478	-0.11065	91.339	-1.6283
0	89.090	-1.6756	-89.943	0.11065	3.2311
-3.2478	-1.6756	3.2224	1.6798	-3.3529	0
-0.11065	-89.943	1.6798	90.854	-0.23155	-3.2169
91.339	0.11065	-3.3529	-0.23155	93.742	-1.5809
-1.6283	3.2311	0	-3.2169	-1.5809	3.2132

Calculating the dot products between the nearly identical columns will show how close they really are. Two vectors are identical when the cosine of the angle between them is one. The cosine of the angle between two vectors is their dot product divided by the length of each vector. When the cosine was calculated for the first and fifth columns, the result was 0.999989, which is equal to 1 when the significant digits are considered. The cosine between the second and fourth vectors is -0.999999, which equals -1.

The problem here is the common field of view (FOV) problem as discussed in Horn [10]. Basically, when the FOV is small, as is the case for most cameras, lateral translation and rotation cannot be distinguished. Second-order terms in the Longuet-Higgins and Prazdny equations can then be dropped. Thus, the motion field for rotation about the y axis becomes $(-B, 0)$. The motion field for lateral translation in the x direction doesn't change and is $(-U/Z_{xy}, 0)$. If the depth is constant over the entire scene, the $-U/Z_{xy}$ term can be replaced by $-U'$. Thus,

rotation can be accounted for in U' , and translation can be incorporated in B . Testing the 6 degree of freedom equations, we found that sometimes the motion estimate would converge to the correct solution, and sometimes the incorrect (constant depth) solution would prevail.

The field of view of our camera is small at 32.8 degrees. The edge of the image is 2.5 mm since we have a 5 mm x 5 mm square CCD array. The principal distance is 8.5 mm. The angle defined by $\arctan(2.5/8.5)$ is 16.4 degrees. The FOV is twice this angle, or 32.8 degrees.

As a result, second order terms can be dropped from the Longuet-Higgins and Prazdny equations, and hence the Least-Squares motion estimation equations. The edge of the image plane is $2.5/8.5$, or 0.294, in normalized units. This value squared is 0.087, which is small compared to one. When we consider the fact that this is the largest second-order term possible and that most second-order terms are much less, we can see that the second-order terms have negligible effect and can be ignored.

Reducing the Degrees of Freedom to Four

A way around this problem is to reduce the degrees of freedom from six to four. The motion can be modeled as some rotation about each of the three axes plus a translation along the optical axis. This eliminates the translations in the x and y directions from the motion estimate and forces their values to be zero. The resulting motion should closely model the motion of the Micro-Rover since it is not supposed to slide laterally to the left or right, nor jump up and down. For example, the Micro-Rover will turn in a circle of some radius. The motion can be approximated by a rotation and a translation forward as shown in Figure 4.3.

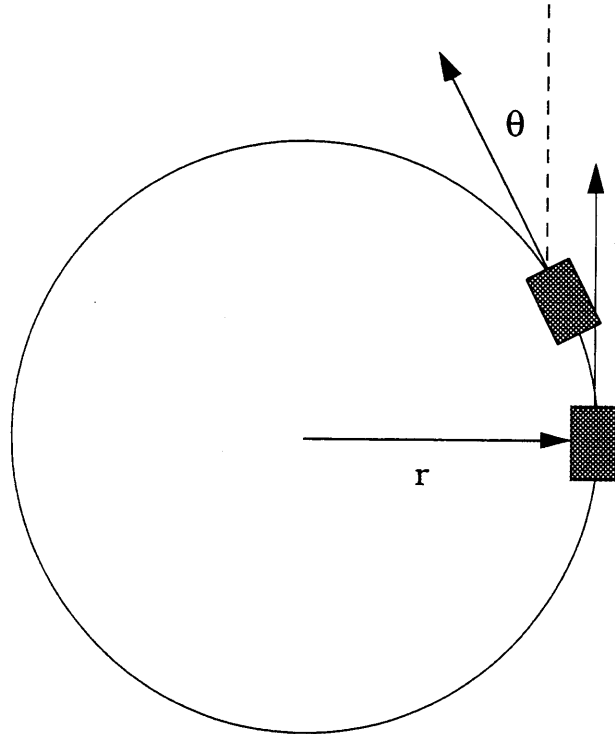


Figure 4.3 Micro-Rover Turning in a Circle

This model for the vehicle motion will yield a non-singular, 4 x 4 matrix which can be solved uniquely. In effect, we are removing the first 2 columns of the 6 x 6 matrix, which represent the U and V components of translation, and the first 2 rows, which are affected by these components. This can easily be seen from the Least-Squares equations in matrix form.

The new set of over-determined equations is:

$$\begin{bmatrix} \frac{y_1}{Z_1} & (y_1^{2+1}) & -x_1 y_1 & -x_1 \\ \frac{x_1}{Z_1} & x_1 y_1 & -(x_1^{2+1}) & y_1 \\ \frac{y_2}{Z_2} & (y_2^{2+1}) & -x_2 y_2 & -x_2 \\ \frac{x_2}{Z_2} & x_2 y_2 & -(x_2^{2+1}) & y_2 \\ \dots & & & \\ \frac{y_n}{Z_n} & (y_n^{2+1}) & -x_n y_n & -x_n \\ \frac{x_n}{Z_n} & x_n y_n & -(x_n^{2+1}) & y_n \end{bmatrix} \mathbf{m} = \mathbf{A} \mathbf{m} = \begin{bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ \dots \\ u_n \\ v_n \end{bmatrix} = \mathbf{b}$$

where the effects of the U and V translation components on the motion field have been removed. The motion vector, \mathbf{m} , is now:

$$\mathbf{m} = \begin{bmatrix} W \\ A \\ B \\ C \end{bmatrix}$$

Solving these equations in the same way as before, we get a 4 x 4 matrix, which is the original 6 x 6 with the first 2 columns and first 2 rows removed.

Testing the New Motion Equations

The two scenes, the planar and contoured, that were originally tested with the 6 degrees of freedom (DOF) model for the Micro-Rover's motion were tested again using the 4 DOF model. The new model will produce 4 independent equations, from which the camera motion can be determined uniquely.

The motion estimation for the planar scene produced estimates that were very close to the true values. The forward translation was again scaled to 4.4 cm, the actual translation. The 3 rotation estimates were very close to zero. The estimates are shown in Table 4.4.

The depth map produced by the motion estimation iteration looks pretty good. The values average around 1.6 m, or 5 feet. There is some variation about this value, but it is within the expected range of depth values as will be seen shortly. Figure 4.4 shows the depth map for this planar case.

The real test of the 4 DOF model is the motion estimation for the contoured scene. For this case, a point on the tires was scaled to 1 m, which was the tires' distance from the camera along the optical axis. The forward translation was found to be 5.3 cm, or 2.09 inches, which is close to 2.25 inches. The 3 rotation estimates are very close to zero. Table 4.5 lists the motion estimates for the contoured scene.

The depth map, as shown in Figure 4.5, also resembles the depths to the various points in the scene. The depth values of the 2 tires are very close to 1 m. The spools of wire are about 1.1 m away. The background is about 2-3 m away.

The results look pretty good for both cases when we consider that the camera characteristics were estimated and not measured. Errors in any of the parameters defining the camera would affect the motion estimate and the depth map. For example, if the principal point were assumed to be closer to the left side of the image than it actually was, then the depth values on the left side would be closer and the depth values on the right side would be farther than they actually were.

The next step is to look at the sensitivity of the motion estimates and depth maps to errors in the flow vectors. This will give us insight on what kind of results we should expect.

Table 4.4 Motion Estimate from Planar Surface Using 4 DOF's

Parameter	Estimated	Actual	Parameter	Estimated	Actual
U (m)	0.000	0.000	A (rad)	-0.002	0.000
V (m)	0.000	0.000	B (rad)	0.000	0.000
W (m)	0.044	0.044	C (rad)	-0.003	0.000

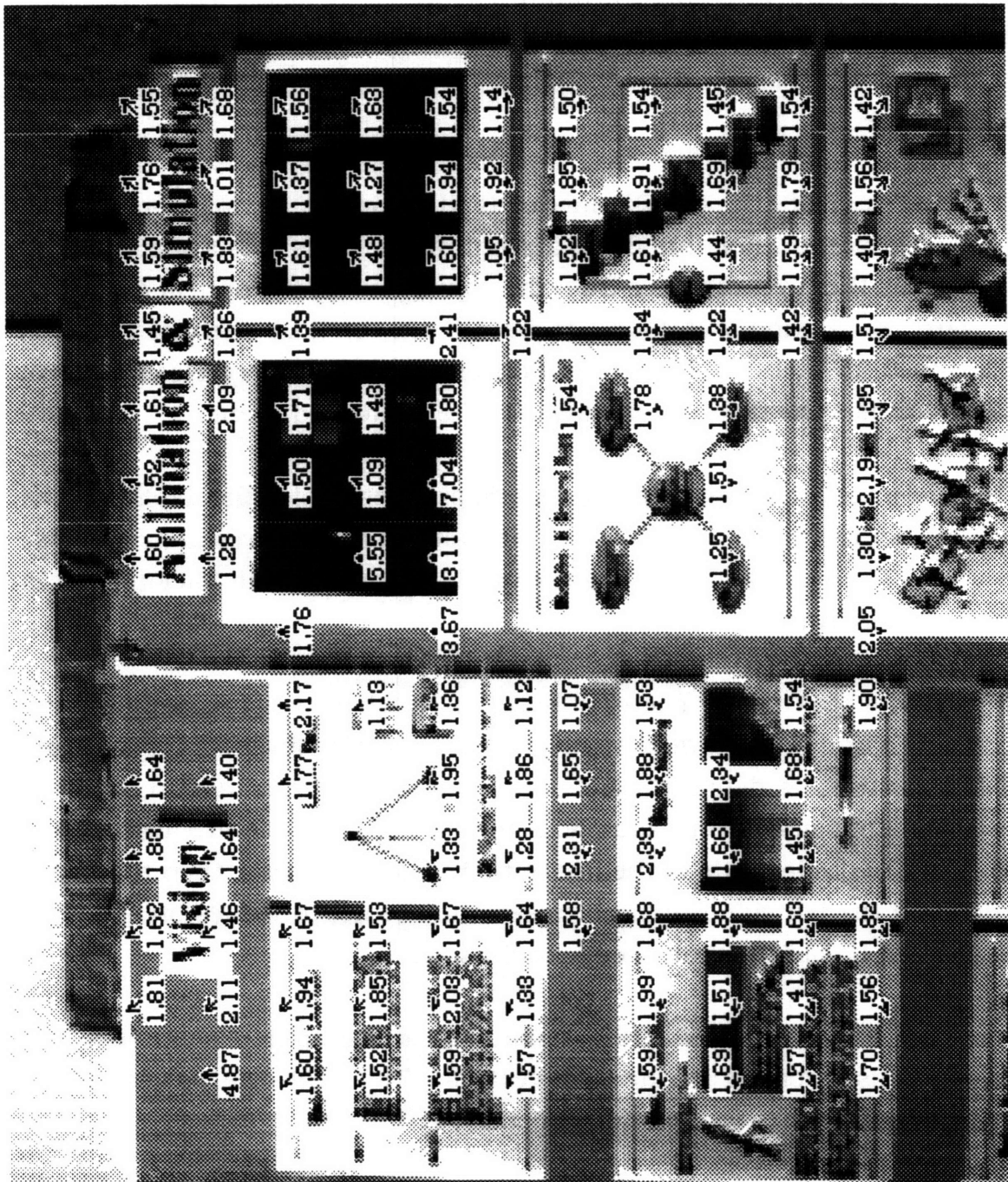


Figure 4.4 Depth Map After Solving for Motion with 4 Degrees of Freedom

Table 4.5 Motion Estimate of Contoured Surface Using 4 DOF's

Parameter	Estimated	Actual	Parameter	Estimated	Actual
U (m)	0.000	0.000	A (rad)	-0.002	0.000
V (m)	0.000	0.000	B (rad)	-0.002	0.000
W (m)	0.053	0.057	C (rad)	-0.005	0.000

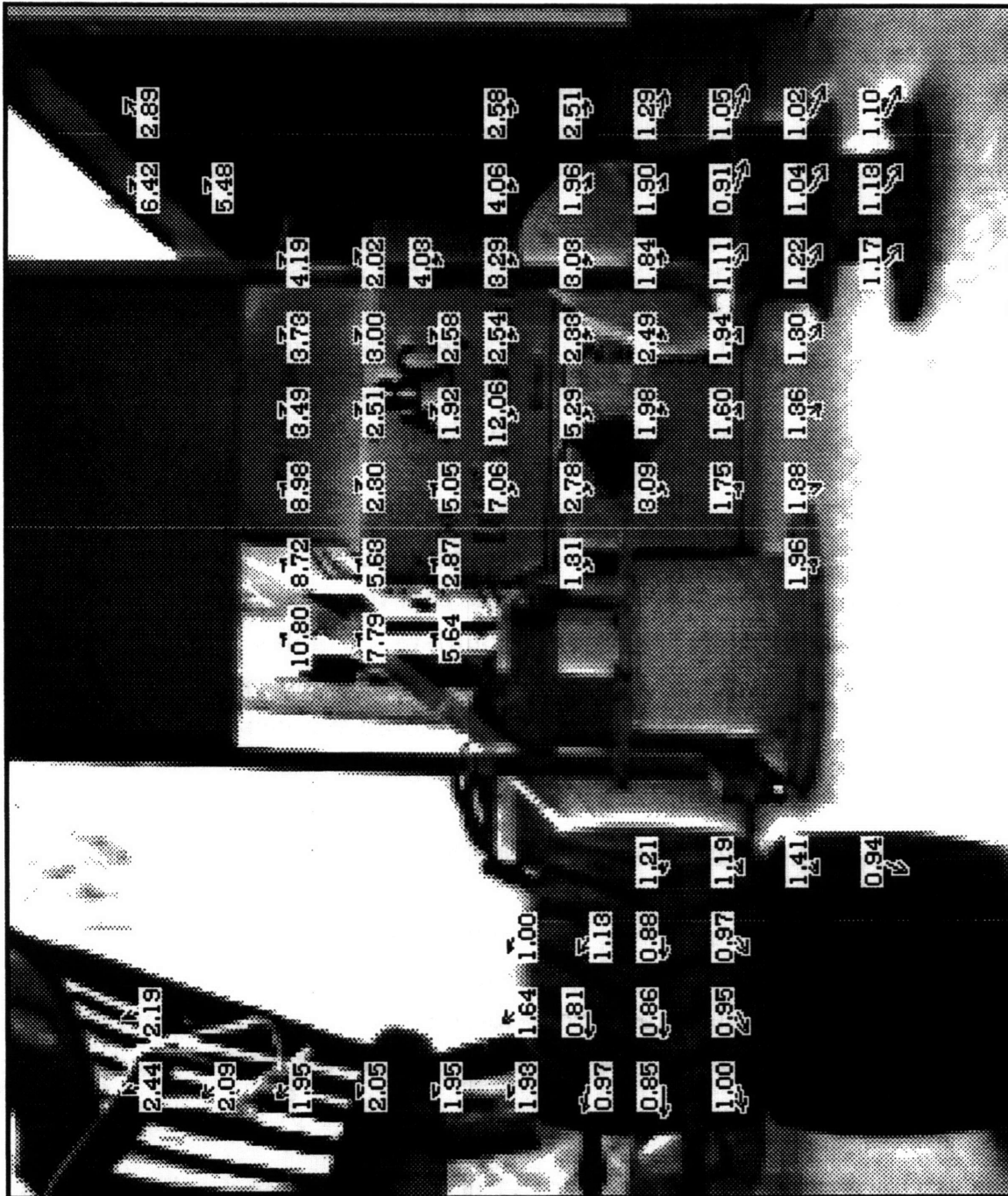


Figure 4.5 Depth Map After Solving for Motion with 4 Degrees of Freedom

Sensitivity of the Motion Estimate to Errors in the Flow Vectors

There are two types of errors that can affect the flow vectors, and hence the motion estimate, namely incorrect match errors and quantization errors. Incorrect match errors result when the new position of the data block in the search window is not the best match. Incorrect match errors can also result when there is too much motion between images, in which case the new position of the data block is outside the search window or only partially inside. Incorrect match errors should be rare, and the weights discussed in the previous chapters should eliminate most of them.

Quantization errors result because the image plane is sampled by the CCD array. Given an absolute position of a point in the image plane, the point will be assigned to the nearest pixel. Thus, for a small range of motions, the sampled image will look the same. This can lead to approximately a half pixel error in each component of a flow vector.

The quantization errors should not affect the motion estimate since they are uncorrelated and will be averaged out in the Least-Squares minimization. It is reasonable to assume that the quantization errors are independent, identically distributed random variables with a mean of zero. The motion estimate for a 256 x 256 image incorporates 196 flow vectors. Any errors in the flow vectors should be averaged out in the final motion estimate.

Sensitivity of the Depth Estimate to Errors in the Flow Vectors

The depth estimates are more sensitive to quantization errors since a depth estimate is calculated from a single flow vector. For each data block, a depth is calculated from its flow vector and the estimated motion. Any quantization error is not averaged out, but reflected in the depth estimate.

A good approximation for the error in the depth estimate can be obtained by looking at the Least-Squares equation for the depth. When the

motion is assumed to be a translation along the optical axis, the Least-Squares estimate for the depth to a scene point, as presented in Chapter 2, is:

$$Z = \frac{(xW)^2 + (yW)^2}{uxW + vyW} = W \frac{x^2 + y^2}{ux + vy}$$

The sensitivity of the depth estimate to the u -component of the flow vector is:

$$\frac{\partial Z}{\partial u} = -xW \frac{x^2 + y^2}{(ux + vy)^2} = -Z \frac{x}{ux + vy}$$

The maximum sensitivity occurs when y equals zero:

$$\left. \frac{\partial Z}{\partial u} \right|_{\max} = -\frac{Z}{u}$$

Note that for this case the motion is assumed to be a translation along the optical axis, so the denominator is always positive except at $(0,0)$, which is the focus of expansion.

The maximum percentage error in the depth estimate is simply 100 times the maximum sensitivity divided by the actual depth:

$$\% \Delta Z \Big|_{\max} = -100 \frac{1}{u}$$

Thus, the accuracy of the depth estimate is inversely proportional to the length of the flow vector. Suppose there is a pixel error in the length of the flow vector due to the image quantization. If the length of the flow vector is 10 pixels, the accuracy of the depth estimate is within 10%.

Errors could be reduced by using a higher-resolution camera and performing a Least-Squares fit to the depth estimates within a region of the image. A higher resolution camera would allow more data blocks to

be matched in the image. A Least-Squares estimate could be calculated that minimizes the sum of squared errors, where an error is defined as the difference between the depth estimate calculated from a flow vector and the Least-Squares estimate.

Estimating the Motion from a Sequence of Images

The last part of evaluating the motion estimation is to test the equations over a sequence of images. Two sequences of images were tested. The first was a sequence of images from the TwoTires scene. The second was a sequence of images taken in the same computer/storage room using stacked books as the obstacle.

The TwoTires image sequence contains 11 images. Images 1 and 10 were used in the previous contoured scene test. In this test, image 1 was compared against images 4, 6, 8, and 10. The camera was translated forward 0.25 inches between images, so the motion between these image pairs are 0.75", 1.25", 1.75", and 2.25", respectively.

The 4 DOF motion model was used to determine the motion between images. The product of cosines weight was applied to the optical flow obtained from each pair of images. The allowable values for the average cosine weight are between 0 and 100. The threshold was set at 75 since it seemed to eliminate most of the incorrect flow vectors.

The depth maps, as shown in Figures 4.6-4.9, were scaled uniformly so that comparisons could be made between them. A point on the tires was selected as the point whose depth would be scaled to the same value on all depth maps. This point is represented by the flow vector that is located on the left-hand side, third up from the bottom.

The motion estimates are affected by the flow vector at this point and any errors in it. As mentioned earlier, the sensitivity of the depth estimate is related to the inverse of the length of the flow vector. If there are any errors in it, the depth map will be scaled either too far or too close, resulting in motion estimates that are too large or too small. A better

way of scaling the depth map would have been to set the average depth of the tires equal to 1 m via a Least-Squares fit.

The motion estimates are very close to the actual values. Table 4.6 shows the motion estimates for the 4 pairs of images. The motion was found to be a translation along the optical axis with very little rotation. The largest value of rotation was 0.006 radians, or about a third of a degree.

Table 4.6 Motion Estimates for Various Pairs of TwoTires Images

Image Pair	U*	V*	W	A	B	C
Images 1 & 4	0.000	0.000	0.015	-0.000	-0.000	-0.001
Images 1 & 6	0.000	0.000	0.031	-0.002	-0.001	-0.001
Images 1 & 8	0.000	0.000	0.046	0.002	-0.001	-0.006
Images 1 & 10	0.000	0.000	0.053	-0.002	-0.002	-0.005

**These parameters were set identically equal to zero*

Table 4.7 shows the translation estimates, the actual translations, and the percentage errors. The first optical flow was scaled to a 3 pixel flow vector, so the error in the motion estimate scaling should be about 33%. In fact, the motion estimate is -21% off the actual motion. The second and third motion estimates are surprisingly close to their actual values. The last motion estimate is off by -7.3%. However, its optical flow was scaled by a 10 pixel flow vector. Thus, it can be expected to have an error of 10%.

Table 4.7 Comparison of Translation Estimate to Actual

Image Pair	W (m)	W (in)	Actual (in)	Error %
Images 1 & 4	0.015	0.59	0.75	-21.3
Images 1 & 6	0.031	1.22	1.25	-2.4
Images 1 & 8	0.046	1.81	1.75	+3.4
Images 1 & 10	0.053	2.09	2.25	-7.3

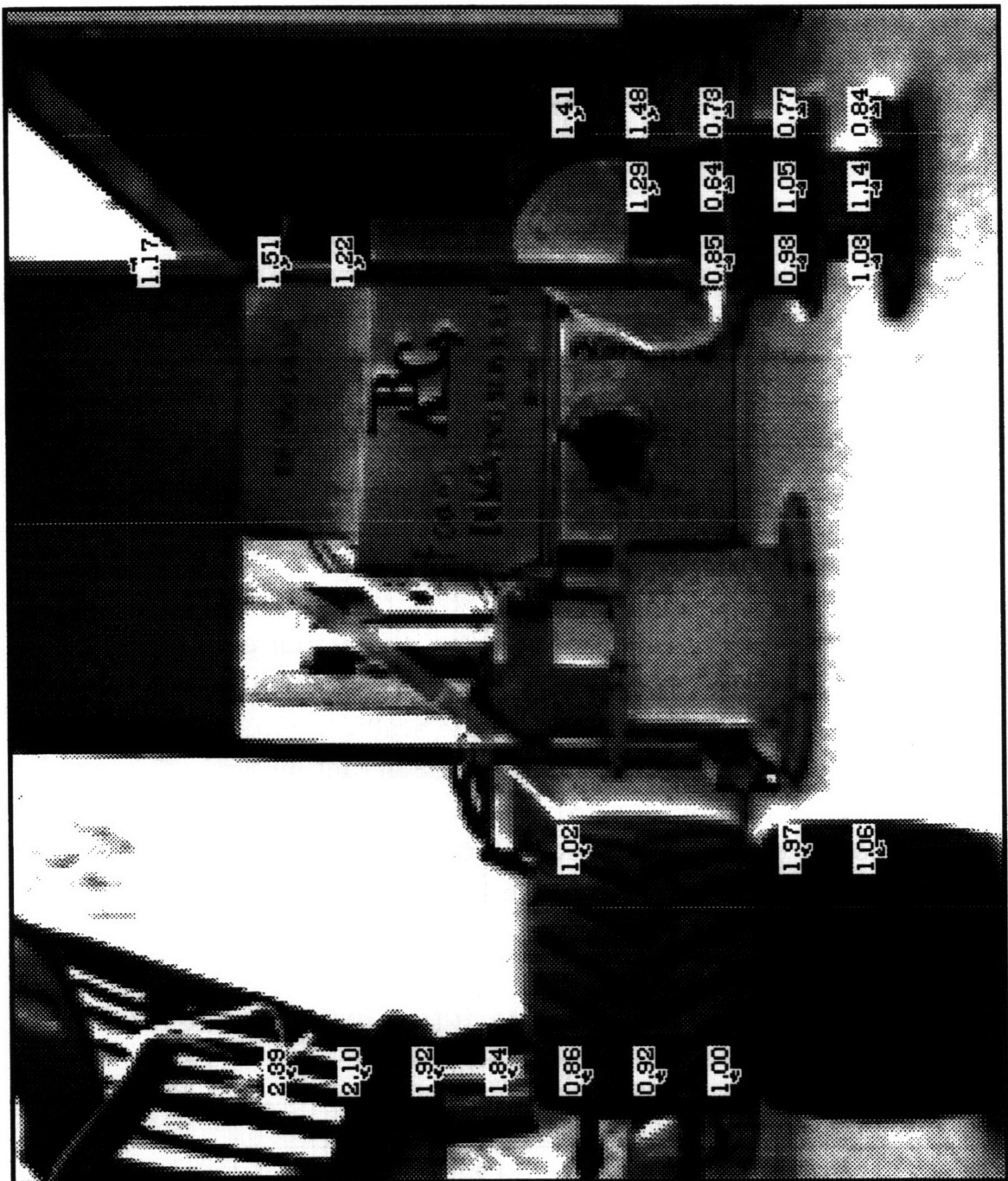


Figure 4.6 Depth Map from Motion Estimation of TwoTires Images 1 & 4

The second sequence of images used to test the motion estimation is the FineBooks sequence. Various books, for which the author was fined for returning late to Barker Library, were stacked in the computer/storage room about 4 feet in front of the camera. The sequence contains 4 images, and the camera was translated 4 inches between successive images.

The motion estimates, as shown in Table 4.8, are very close to the actual values. All the W components of translation are close to the correct value of 10 cm, and the rotation components are all near zero.

The depth maps, shown in Figures 4.10-4.12, show that the books stand out very distinctly from the background. The depth to the books is around 1.4 m, and the depth to the background is about 2.5 m. The depth maps could aid in hazard avoidance if they could be fused together into some obstacle map that showed the positions of obstacles on the ground plane.

Table 4.8 Motion Estimates for Various Pairs of FineBooks Images

Image Pair	U*	V*	W	A	B	C
Images 1 & 2	0.000	0.000	0.092	-0.004	-0.001	0.003
Images 2 & 3	0.000	0.000	0.099	-0.004	-0.001	0.000
Images 3 & 4	0.000	0.000	0.107	-0.004	-0.001	0.005

**These parameters were set identically equal to zero*

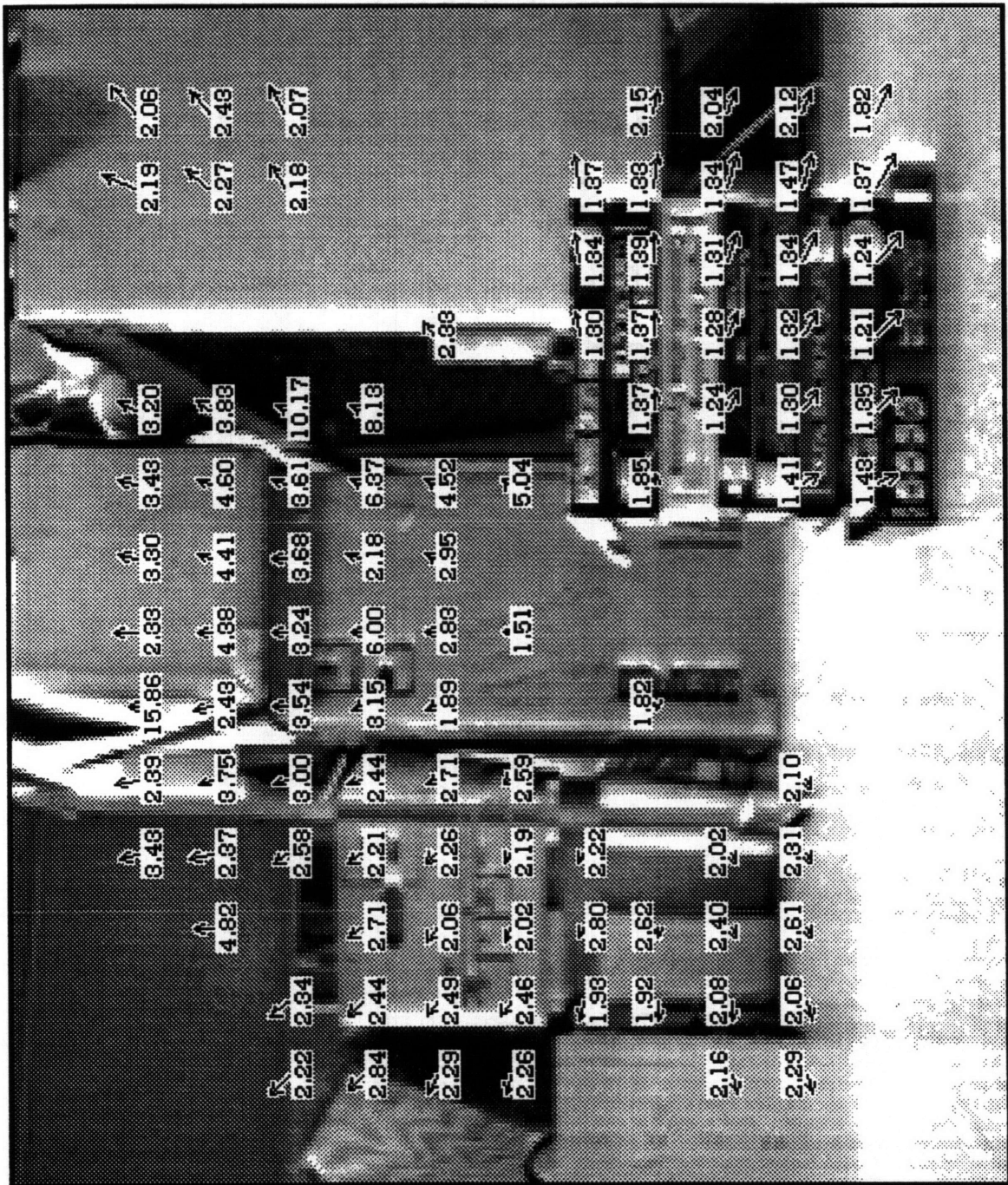


Figure 4.11 Depth Map from FineBooks Images 2 & 3

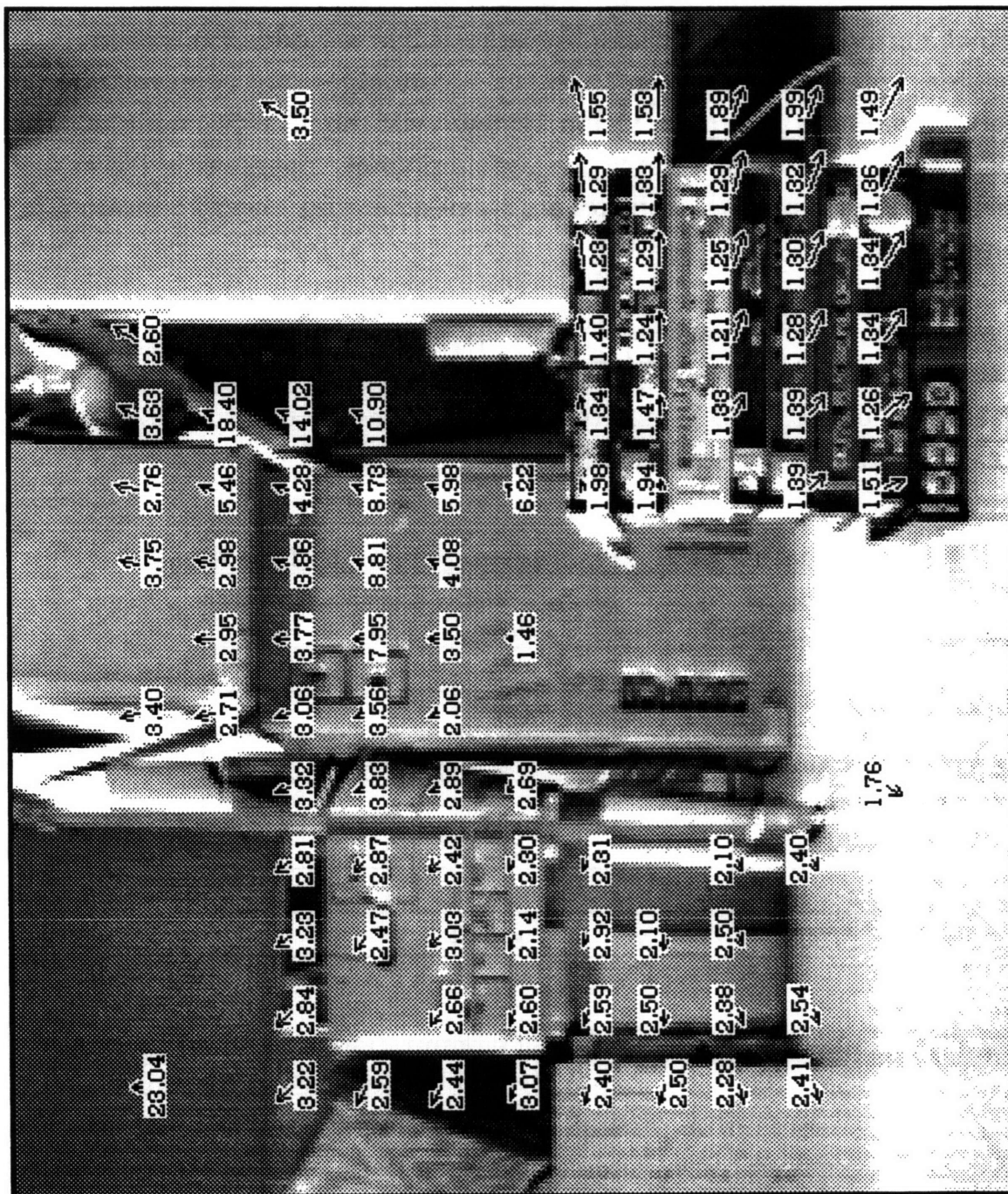


Figure 4.12 Depth Map from FineBooks Images 3 & 4

Microprocessor Re-Reconsiderations

Just as we reconsidered the number of computations involved in implementing the motion vision system when we added the product of cosines weights, we must now re-reconsider the computational requirements of the motion vision system with the 4 DOF model for the Micro-Rover's motion. Since we are implementing fewer degrees of freedom, there are less coefficients to calculate and fewer equations to solve.

The number of computations per data block have been significantly reduced from 55 to 23. Only one element of the 3 x 3 **S** matrix and only 2 elements of the 3 x 3 **g** matrix need to be computed. Table 4.9 shows the computations per data block.

Table 4.9 Matrix Calculations for Each Data Block

Calculation	Mult	Div	Add	Sub
S matrix	1	1	1	
g matrix		2	2	
g^T matrix			2	
R matrix				
D vector	2	2	2	
E vector	4		4	
Total	7	5	11	

Table 4.10 shows that the number of computations required to update a depth value has also been reduced. The savings result because the U and V components of the translation were eliminated. This changes the dot product between the **s_u** and **s_v** vectors and the translation vector to a simple scalar multiplication.

Table 4.10 Calculations to Update a Depth Value

Calculation	Mult	Div	Add	Sub
numerator	4		1	
denominator	8	1	5	2
Total	12	1	6	2

The number of calculations per step of the iteration has thus been reduced. The fewer computations required to generate the 4 x 4 array and the 4-D vector are due to the fewer number of computations per data block. The number of calculations to update a depth value also contributes to the lower number of computations per step of the iteration. Additionally, the number of computations to solve the 4 linear equations is reduced, although its influence is negligible. The number of computations to calculate the product of cosines weights is not affected since the weights are generated from the flow vectors (which still have 2 components - u and v!). Table 4.11 shows the number of computations per step of the iteration.

Table 4.11 Computations Required per Step of the Iteration

Type of Computation	Number of Computations
Compute 6x6 Matrix	1,568
Compute 6-D Vector	2,744
Solve Linear Equations	74
Update Depth Map	4,116
Calculate Weights	8,428
Safety Margin	8,465
Total	25,395

The final microprocessor re-reconsiderations are shown in Table 4.12. When comparing these results to those in Table 3.2, one notices that the number of cycles per computation has increased by almost 50% from 4.0 to 5.9. This is significant, but not as dramatic as may have been

anticipated. The computations required to calculate the product of cosines weights have become the most significant computation sink.

Table 4.12 Microprocessor Re-Reconsiderations

Clock Speed	30 MHz
Duty Ratio	50%
Effective Clock Speed	15 MHz
Number of Computations	2,539,500
Cycles per Computation	5.9

Chapter Summary

In this chapter, the Least-Squares equations for the motion of the Micro-Rover were tested. The narrow field of view of the camera makes it impossible to distinguish rotation from lateral translation. Since the Micro-Rover isn't expected to move laterally, a 4 degree of freedom model, which eliminates the lateral translations, was developed in place of the 6 degree of freedom model for motion. The motion estimates using this model are close to the actual values. Sensitivity analyses were performed to examine the motion estimates from sequences of images. They showed that the estimates were reliable and could be used in a dead-reckoning navigation system.

Chapter 5 Obstacle Recognition

In this chapter, the use of depth maps to aid in hazard avoidance is explored. A single depth map can generate an obstacle map, which is a top-down view of the ground plane with potential hazards marked. The locations of obstacles are placed on the obstacle map using uncertainty regions which represent our best knowledge about the obstacles' locations. Due to the uncertainty regions and some incorrect flow vectors that manage to get through the product of cosines filter, there will be noise in an obstacle map generated from a single depth map. Therefore, obstacle maps from a sequence of images are combined together to enhance the locations of obstacles and attenuate the noise. The final test applied to an obstacle map should be a binary threshold filter, which determines whether or not an obstacle is present in a small region of the map.

Obstacle Recognition from the Depth Maps

The first step in generating an obstacle map from a single depth map is translating the scene points from the camera coordinate system to the world coordinate system. The depth values in a depth map mark the distances to various scene points. These scene points represent various structures or obstacles in the environment. The coordinates of these scene points can be translated from the camera coordinate system to the world coordinate system by using the depth values. The depth values are the Z coordinates of the scene points.

The next step is to take the world coordinates and drop them straight down onto the ground plane. This requires knowledge of the Micro-Rover's attitude. In this thesis, we will assume that both the pitch and roll of the Micro-Rover are zero, i.e. the Y axis is perpendicular to the ground plane. The positions of the various scene points on this ground plane comprise an obstacle map. The Micro-Rover can then navigate around obstacles by noting their positions relative to its own.

To be more explicit, the depth value is used with the x and y normalized coordinates to find the position of the obstacle and its height. The depth value is the distance of the obstacle along the optical axis, or the Z coordinate of the obstacle. The x coordinate of the data block and the depth value are used to find the X coordinate of the obstacle by multiplying them together. Since we are projecting the world coordinates onto a ground plane, the Y coordinate is not needed. However, the height can be used to differentiate a match on an obstacle from a match on the ground plane. Thus, the y coordinate and the depth value are multiplied to obtain the height. Figure 5.1 shows the translation of a scene point onto the ground plane.

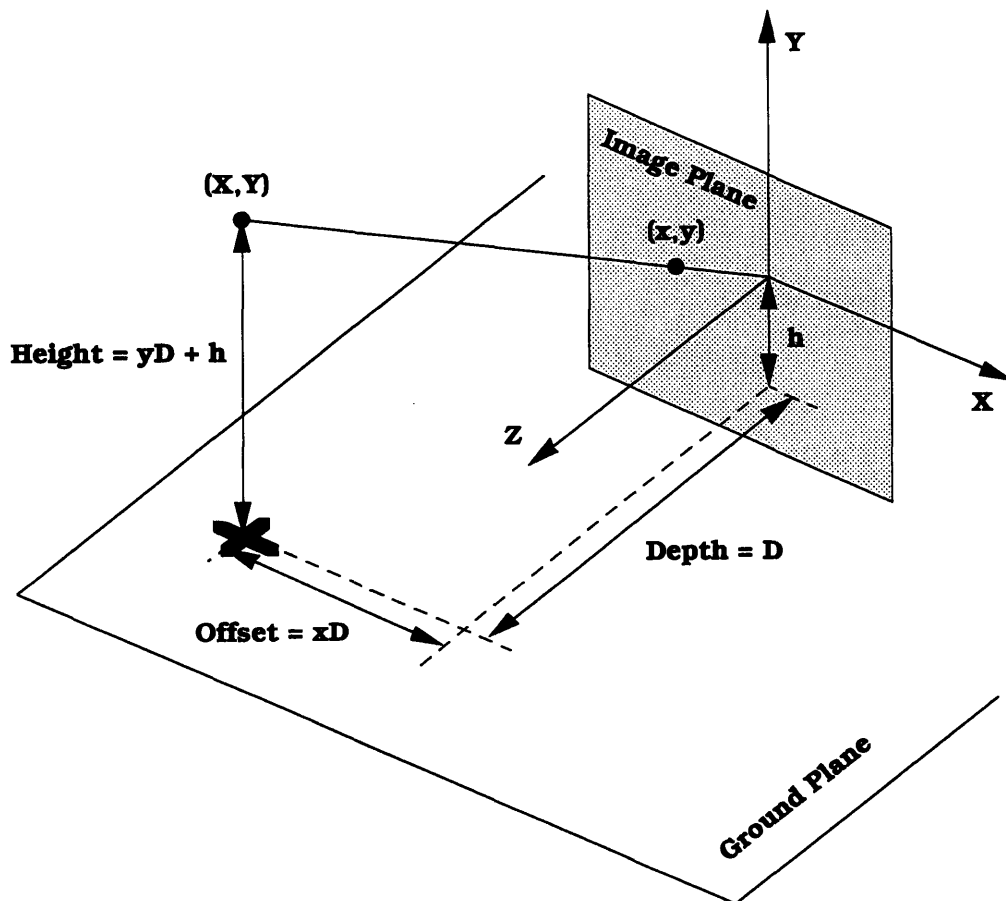


Figure 5.1 Translation of Scene Point onto the Ground Plane

An obstacle map is quantized by dividing it into many, small regions. The dimensions of these increments are based upon the resolution required by the navigation computer and those obtainable by the vision system. The obstacle maps that were generated by TwinScreen™ used square increments that were 5 cm on a side.

A scene point cannot be dropped onto the ground plane and placed in a single increment because there is some uncertainty to the actual position of the point. As discussed in Chapter 4, there is an uncertainty associated with each depth value that is inversely proportional to its length. Additionally, a depth value corresponds to a data block which has a finite width and height in the image plane. This width and height in the image plane corresponds to an uncertainty in the width and height of the world coordinate.

In our testing, a depth uncertainty, or uncertainty in the Z coordinate, is assigned to each world coordinate and based upon the length of the associated flow vector. Our sensitivity analysis, in which the derivative of the Least-Squares estimate for the depth with respect to a change in one coordinate of the flow vector was found, showed that the depth estimate has an uncertainty, or error, that is inversely proportional to the length of the flow vector. Thus, the boundaries for the uncertainty region on the Z axis are the Z coordinate of the scene point \pm the inverse of the length of the flow vector.

An additional uncertainty in the X component is assigned to each scene point and is based upon the width of the data block. Each data block is 16 pixels wide. The center of the data block determines the x and y coordinates in the image plane. An uncertainty of 4 pixels to either side of center is assigned to the x component. This gives us a minimum and maximum value for the x coordinate, and hence a minimum and maximum value for the X coordinate.

The extent of the X component uncertainty is dependent upon the particular depth value since the minimum and maximum x coordinates actually define angles. The minimum and maximum x components are

divided by the principal distance to normalize them. These normalized values are actually tangents of the minimum and maximum angles between a ray to the scene point and the optical axis. The X components are found by multiplying these tangents by a depth value. Thus, the X component uncertainty gets larger as the depth increases.

The uncertainty region that we are using is a trapezoid since we have two sides that are parallel and two sides that are not. The uncertainty in the depth creates 2 parallel boundaries that are perpendicular to the Z axis. The X component uncertainty creates 2 boundaries that meet at the center of projection.

The process of generating an obstacle map begins by translating each data block onto the ground plane as an uncertainty region. The size of the uncertainty region reflects the uncertainty in both the depth estimate and the X component. Since there are many depth values to translate onto the ground plane, there is a possibility that 2 or more uncertainty regions could overlap on the ground plane.

To represent overlaps of the uncertainty regions, the obstacle map must display several shades of gray. At each increment in the obstacle map, a count is kept on the number of uncertainty regions that overlap that increment. This count is then converted into a grayscale color, with a darker shade of gray representing many overlapping uncertainty regions. The obstacle map can then be viewed and interpreted easily.

The 4 depth maps from the TwoTires sequence that was analyzed in Chapter 4 were converted into obstacle maps. They are shown in Figures 5.2 - 5.5. The TwinScreen™ functions used to generate the obstacle maps are presented in Appendix C.

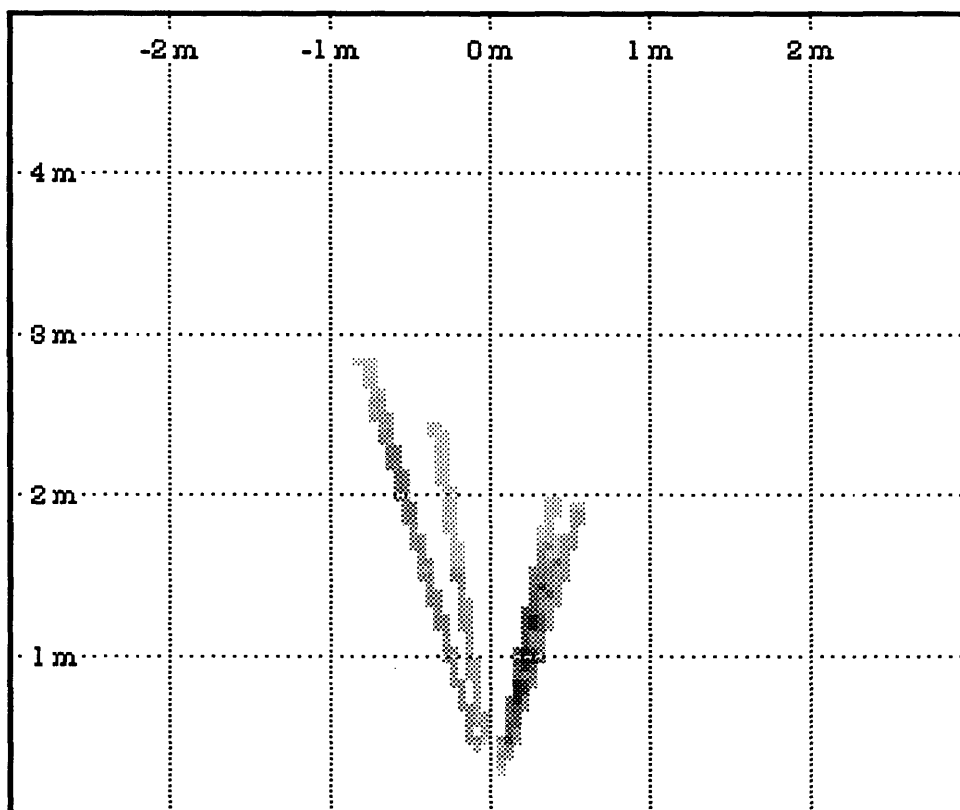


Figure 5.2 Obstacle Map from Depth Map for TwoTires Images 1 & 4

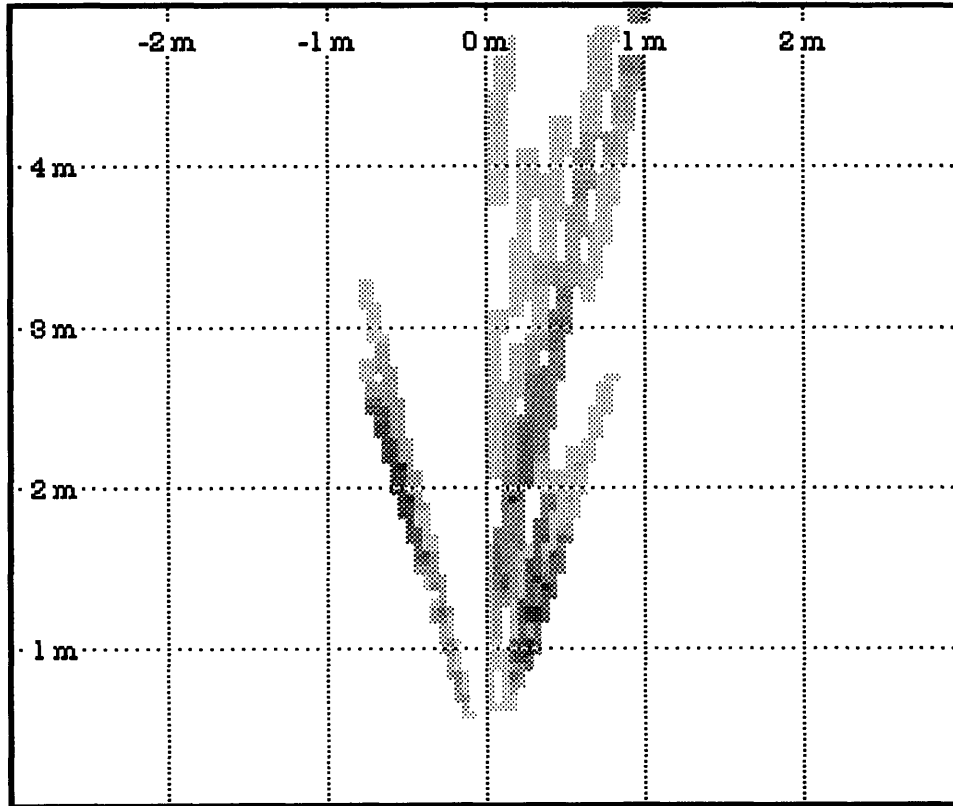


Figure 5.3 Obstacle Map from Depth Map for TwoTires Images 1 & 6

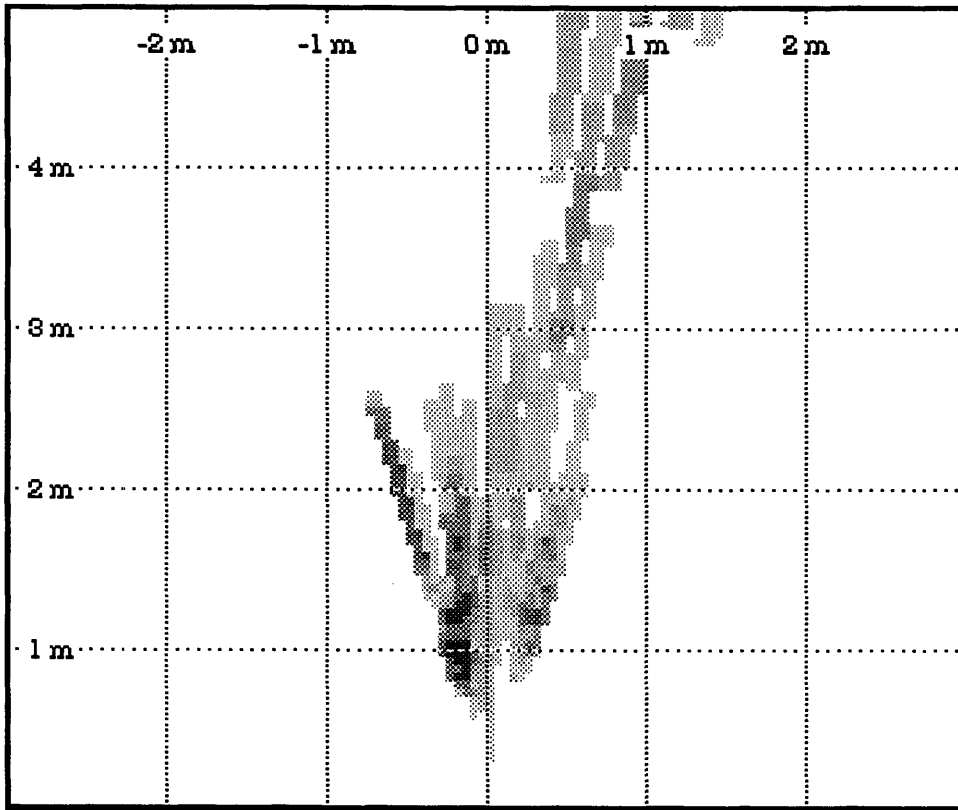


Figure 5.4 Obstacle Map from Depth Map for TwoTires Images 1 & 8

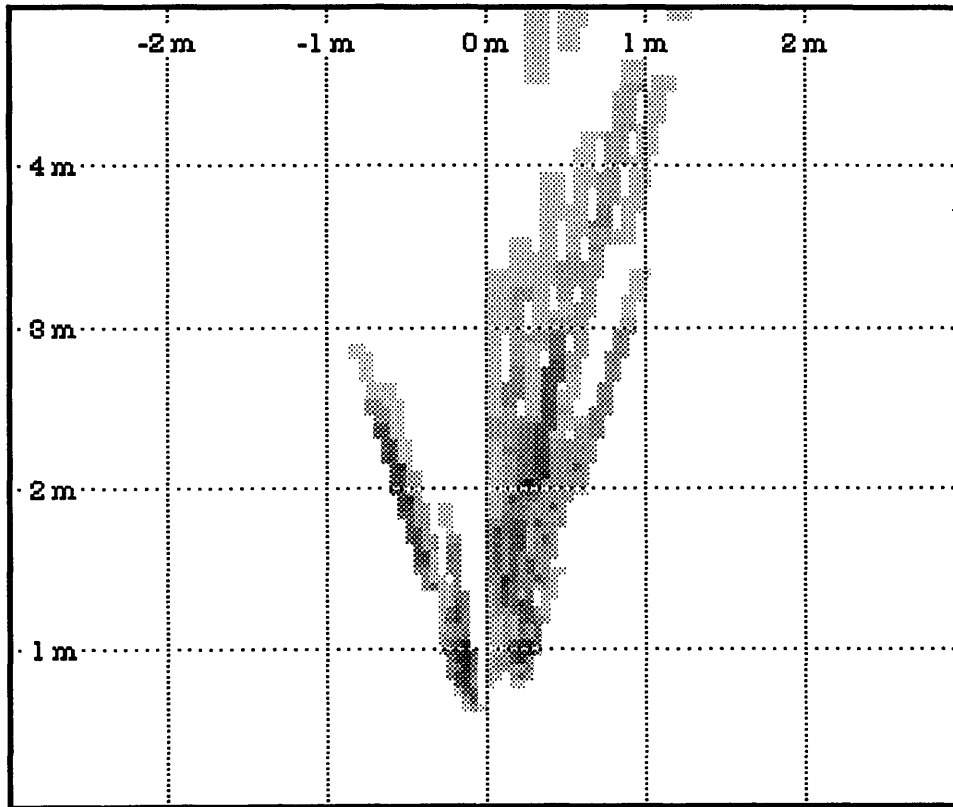


Figure 5.5 Obstacle Map from Depth Map for TwoTires Images 1 & 10

An obstacle map generated from a single depth map, as these 4 obstacle maps are, may be noisy due to errors in the flow vectors that produce errors in the depth estimates. Quantization errors can produce errors in the flow vectors. As previously discussed, the accuracy of the depth estimate is inversely proportional to the length of the underlying flow vector. Additionally, some incorrect matches may get through the product of cosines filter. These incorrect matches produce erroneous flow vectors, which lead to noise in the obstacle map.

An obstacle map can be improved by combining the depth maps from a sequence of images because the noise will be reduced. An obstacle map generated from a single depth map contains a lot of noise as seen in the previous obstacle maps. However, this noise should be uncorrelated between obstacle maps generated from different depth maps. Therefore, using a sequence of depth maps to generate an obstacle map should reinforce the points that are obstacles and attenuate points that are noise.

The process of fusing the depth maps is performed as follows. The first depth map from a sequence of images is used to generate an initial depth map. Then, the next image is used to generate an intermediate obstacle map. For every increment in the intermediate map where one or more uncertainty regions overlap, the count at the corresponding increment in the first obstacle map is increased by this value. For every increment in the intermediate map where no uncertainty regions overlap, the corresponding increment in the first obstacle map is checked and decremented by one if its value is not zero.

This method of combining the depth maps will tend to enhance any obstacles and attenuate any noise. The count, or grayscale, at increments which continually have one or more uncertainty regions overlap it will grow, or get darker. At increments where a few uncertainty regions from a single depth map overlapped, the count, or grayscale, will decrease, or become lighter.

There is a simple rationale behind this method of filtering. If several depth maps report this location as an obstacle, the value at the increment will be high, and there is probably an obstacle there. However, if this location is reported as an obstacle only a few times (most likely because of noise) then the value will be low, and this location is probably clear.

An obstacle map was generated for the 4 TwoTires depth maps and is shown in Figure 5.6. The tires, spools, and chair are distinctly shown in black. There is some noise in the obstacle map, but it is much less than in the 4 individual obstacle maps.

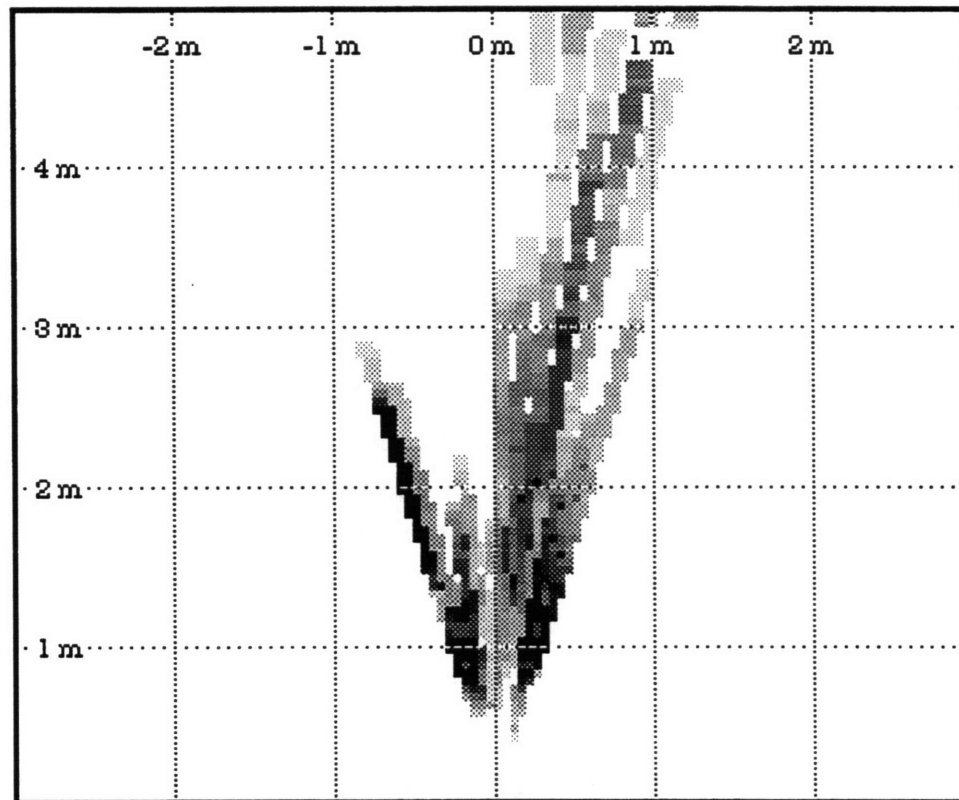


Figure 5.6 Obstacle Map from the 4 TwoTires Depth Maps

The last step required to use this obstacle map to avoid hazards is to make a decision at each increment whether or not an obstacle is present. There is still some noise in the fused obstacle map, so not every increment which has a non-zero count can be accepted as an obstacle.

Some count threshold must be established, above which it is very likely that an obstacle is present and below which it is very improbable that an obstacle is present. The exact threshold depends upon the relative importance of Type I errors (failing to detect an obstacle) to Type II errors (incorrectly flagging an obstacle) in a particular application. The fused obstacle map in Figure 5.6 was thresholded and appears in Figure 5.7.

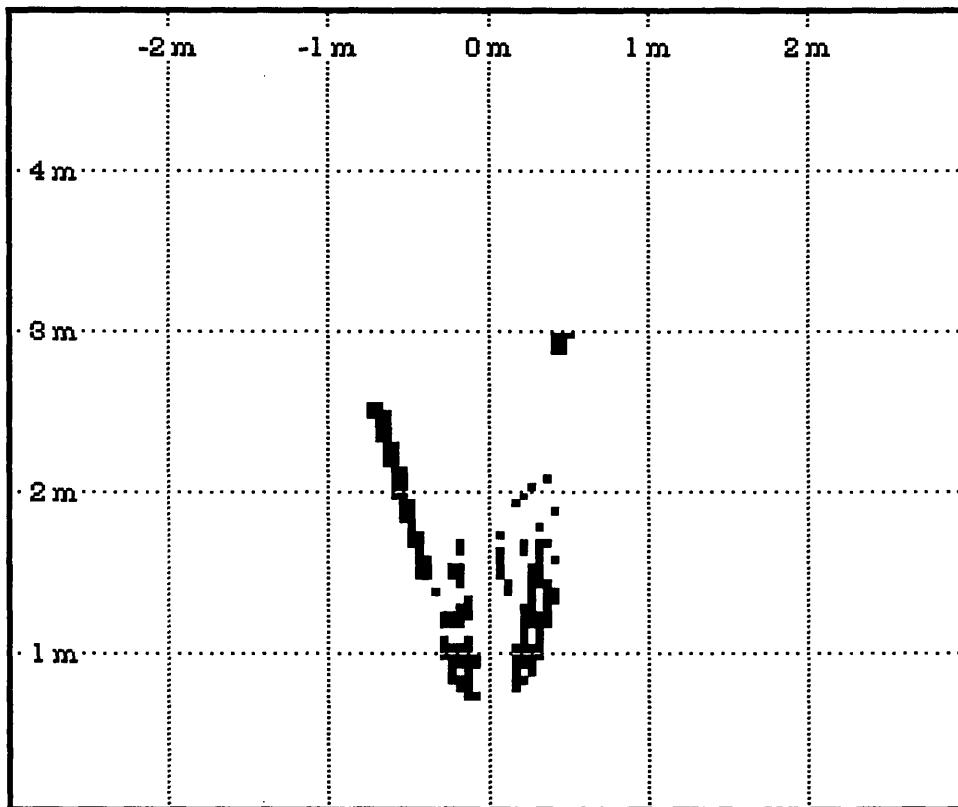


Figure 5.7 Obstacle Map from 4 TwoTires Images after Thresholding

An obstacle map was generated from each of the depth maps from the FineBooks sequence. Figures 5.8 - 5.10 show the 3 obstacle maps. In this case, the books stand out clearly as an obstacle in each of the obstacle maps. However, there is still some noise in the maps.

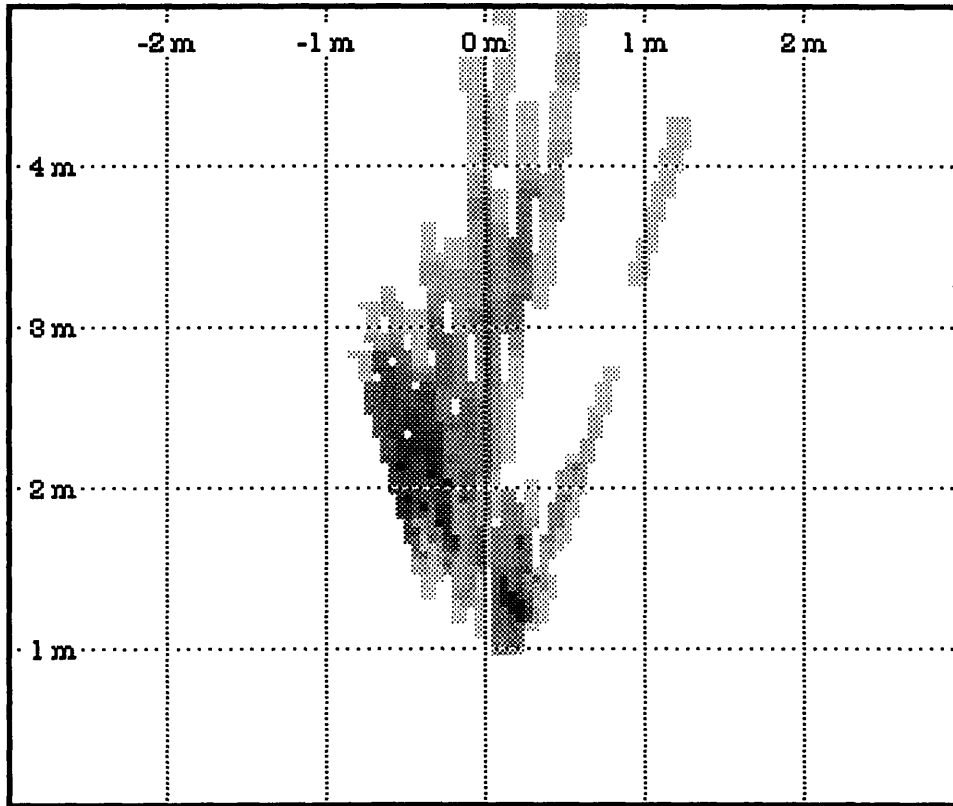


Figure 5.8 Obstacle Map from Depth Map for FineBooks Images 1 & 2

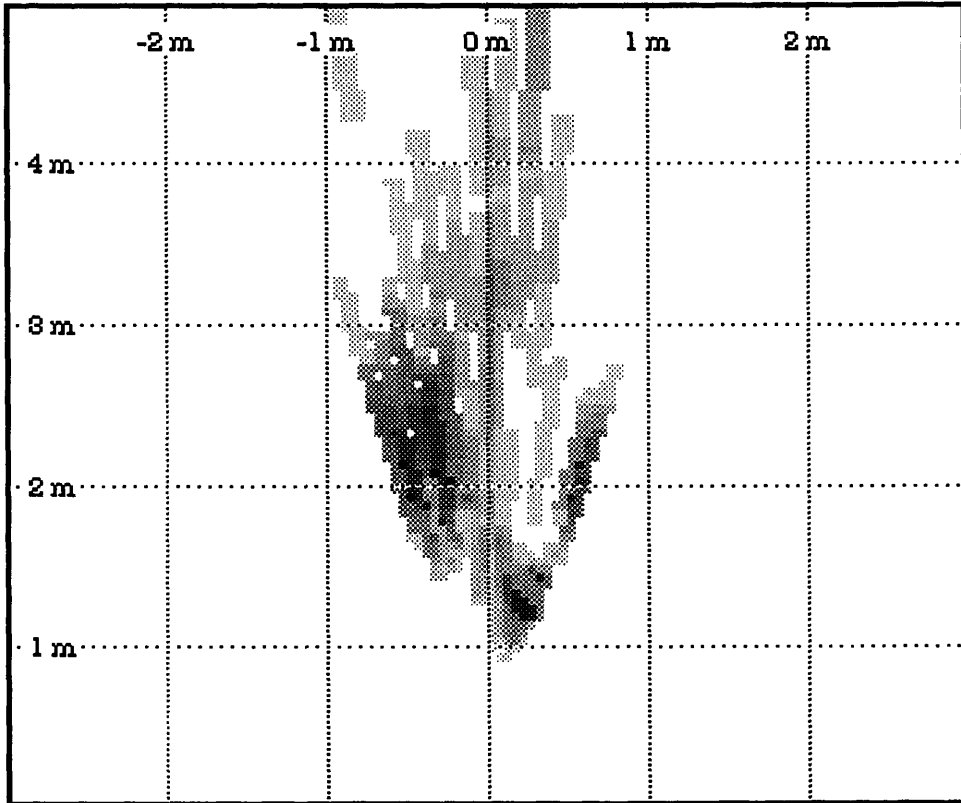


Figure 5.9 Obstacle Map from Depth Map for FineBooks Images 2 & 3

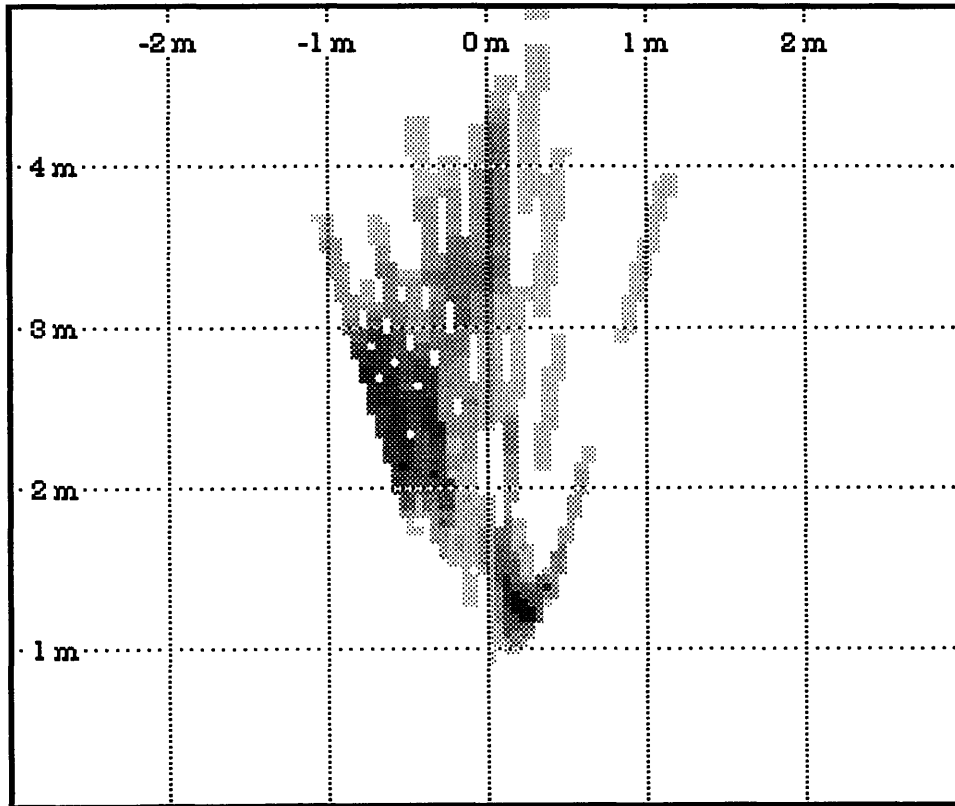


Figure 5.10 Obstacle Map from Depth Map for FineBooks Images 3 & 4

The 3 obstacle maps were then fused together, and the resulting obstacle map is shown in Figure 5.11. Now, the books as well as the background stand out clearly as obstacles. Additionally, the noise has been reduced.

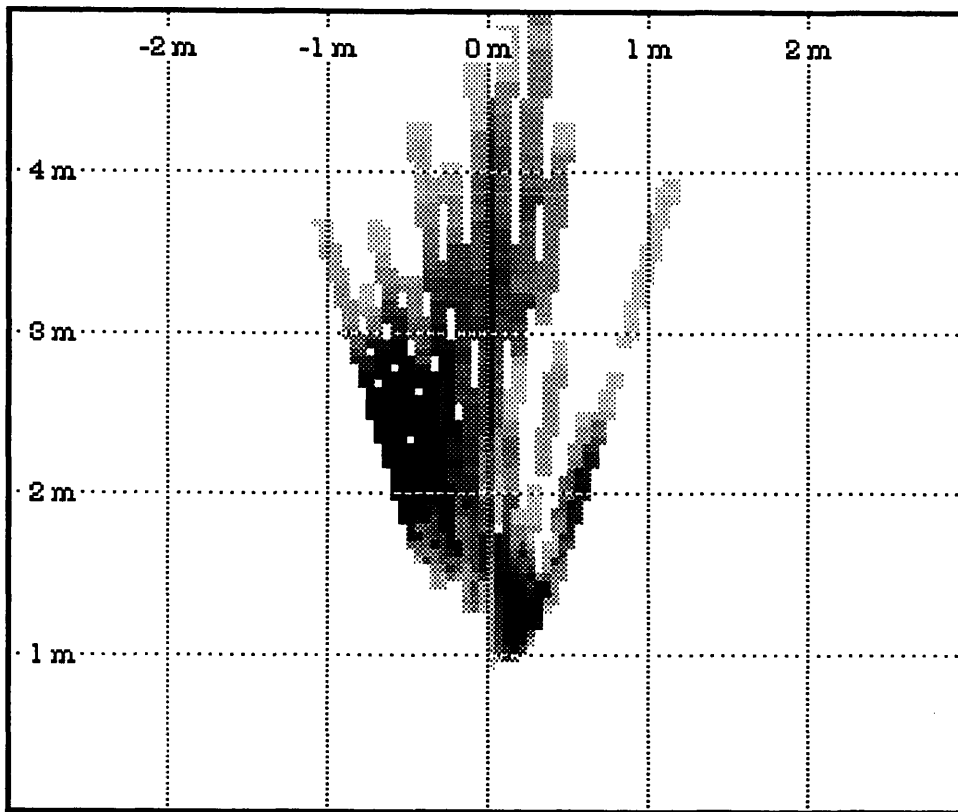


Figure 5.11 Obstacle Map from 3 FineBooks Depth Maps

The fused obstacle map was then thresholded to produce the obstacle map shown in Figure 5.12.

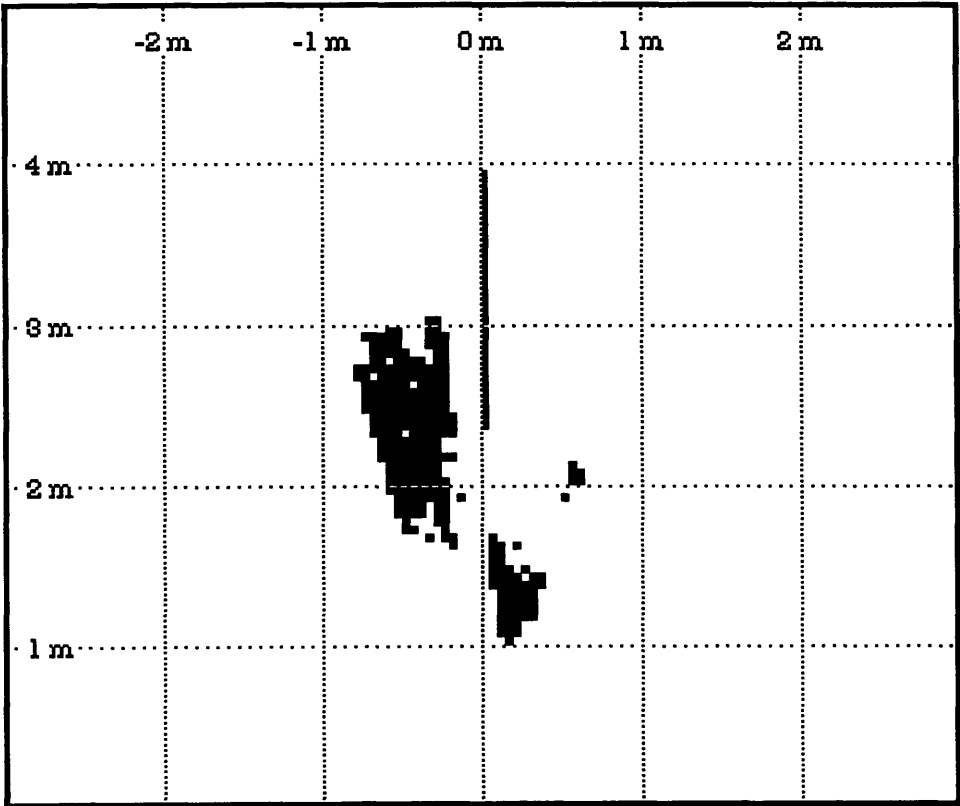


Figure 5.12 Obstacle Map from 3 FineBooks Images after Thresholding

Chapter Summary

This chapter discussed how the depth maps could be transformed into obstacle maps and used in hazard avoidance. Each depth value can be translated into a scene point by using the x and y coordinates of the corresponding data block. The world coordinate point can then be dropped onto the ground plane to mark an obstacle. An uncertainty region should be drawn around this point on the ground plane because there is an uncertainty in the actual depth of the point as well as the X coordinate. Several obstacle maps from a series of images may be combined to enhance obstacles and attenuate noise. The final step is to set a threshold that determines whether or not there is an obstacle at each increment.

Chapter 6 Conclusion

The aim of this project was to develop a motion vision system that could be implemented on a small, low-power vehicle, such as a Micro-Rover. It was desired to use the camera we already had on-board the Micro-Rover to estimate both its motion and an obstacle map of the environment. The motion estimates between images could be used in a dead-reckoning navigation system, and the obstacle map could be used for hazard avoidance.

Current motion vision systems require too many computations per second to be implemented on a small vehicle. The size and power constraints of the Micro-Rover limit the computational complexity of the vision system. Implementing the conventional brightness gradient method would require about 120 million computations per second, assuming 4, 256 x 256 images were processed, and each iteration for the motion required 10 steps. This is on the order of workstation or multiple-workstation computing to achieve a real-time motion vision system.

An alternate way of implementing motion vision in real-time uses pattern matching and leaves considerably less computations to be performed by a microprocessor. An integrated circuit chip designed for HDTV applications will take a pattern from one image and find the best match within a region of another image. This chip can match 256 x 256 images at a rate of 52 images per second. The processor must then perform about 1.2 million computations per second, under the same assumptions and using the product of cosines weights. The number of computations is about 2 orders of magnitude less than the brightness gradient method at the expense of a single, low-power (can I make this claim without justifying it?) board.

The requirements of the motion vision system were that it produce motion estimates that are reasonably accurate and obstacle maps that could be used to avoid hazards. The proposed Micro-Rover mission, as outlined by the Jet Propulsion Laboratory in Pasadena, CA, requires a

navigational accuracy of 10%, i.e. the error in its position should be at most 10% of the distance traveled. Using the obstacle maps to avoid hazards requires that almost all hazards are recognized and very few false alarms are generated.

The motion was estimated for several translations along the optical axis, and the results show that the matching algorithm can produce reasonable estimates that are within 10%. Table 4.7 shows the estimates for 4 image pairs from the TwoTires sequence. The only estimate that is more than 10% from the actual value is the first, in which the flow vector to which the depth map was scaled is only 3 pixels long. The sensitivity analysis shows that the expected error for this case should be about 33%. The other 3 estimates are very close to the actual values, with errors of -2.4%, +3.4%, and -7.3%.

It is worthwhile to note at this point that the motion estimates are best when the flow vectors calculated from the matches are as long as possible. The depth map must be scaled by some measurement of the environment. In this case, the depth associated with a particular data block is measured, and the depth value at that data block is scaled to equal the measured depth. All other depth values in the depth map are scaled by this scaling factor. The scaling factor is related to the length of the flow vector associated with the particular data block, so if the length is off, the scaling factor will be incorrect.

An example will illustrate this point. Assume the depth for a particular data block is measured as 5 m, and the motion between images is a translation of 3 cm, which should produce a flow vector that is 3 pixels long at that data block. However, suppose that the matching produces a flow vector of length 2 pixels. This data block stands out as a farther point when its flow vector is 2 pixels as opposed to 3 pixels, so scaling its depth to 5 m will scale all depths to be closer than they actually are. Thus, the world will appear to be smaller than it actually is, and the resulting motion estimate will be less than the 3 cm actual translation.

It is important to remember that the sensitivity of a depth value to a 1 pixel error in the length of the flow vector is inversely proportional to this length. In the previous example, the scene would be scaled to be 1/3, or 33%, smaller than it actually was. However, if the flow vector associated with the particular data block were 8 pixels long, the error in the scene scaling would be only -12.5%.

This fact requires the development of a system in which the lengths of the flow vectors are fed back to the image capture hardware to adjust the time between images. The image capture hardware will "grab" an image, wait for a specified length of time, and then "grab" another image. The time between images can be set by the microprocessor after it calculates some metric on the length of the flow vectors and decides the appropriate time between images. Note that the flow vectors at the edge of the image should be the longest most of the time, so the metric may not need to incorporate all flow vectors.

When the motion estimates are combined with a low-pass filter, it is expected that very good results can be obtained. The Least-Squares equations remove the effects of most of the errors in the flow vectors, but there will be some variation in successive motion estimates that can be removed by filtering out the high-frequency components. The Micro-Rover cannot contribute to these high-frequency components because the motion of the mechanical structure cannot change that quickly.

Rotation remains a stumbling block for this vision system, as well as for most vision systems, because the field of view of the camera is very small. As discussed in Chapter 4, there is little or no distinction between flow vectors due to rotation and flow vectors due to lateral translation. The suggestion in this thesis is to change the model for the Micro-Rover's motion to eliminate the lateral translations which it cannot perform.

Additionally, this vision system places a constraint upon the rotation between images because the data block cannot move outside the

search window. The allowable rotation between images permits the data block to move to the edge of the search window. Appendix D discusses the maximum rotations for the case of 16 x 16 data blocks using 32 x 32 search windows.

This rotation constraint is obvious from the implementation of this vision system, however, the author expects it to also apply to other vision systems. Because they are not as intuitively easy, finding the constraint may not be as straight-forward. It is probably reasonable to assume that the brightness gradient method works when an image point has moved 8 pixels, but how about 16 or 24 pixels? It is not obvious whether or not it will work. Additionally, for a movement of 24 pixels, about 10% of the image is new, and one must wonder if the results are affected.

The depth maps obtained from this vision system can be used for obstacle avoidance when they are translated to a top-down view. The depth values correspond to scene points, or points on a structure or obstacle. The world coordinates of these points can be found, and then these points can be dropped down onto a ground plane. Each scene point should be represented not as a point on the ground plane, but as an uncertainty region to represent the uncertainty in both depth and width. A large number of overlapping uncertainty regions at an increment in the ground plane indicates that there is a high probability that there is an obstacle.

The obstacle maps must be integrated over time to filter out erroneous obstacles or noise that result from errors in the flow vectors. Since the image is quantized, a flow vector may be off by a small amount, resulting in an incorrect depth value. Additionally, a totally incorrect flow vector could get through the product of cosines filter and produce a totally incorrect depth value. The errors in the flow vectors, and hence depth values, generate stray uncertainty regions that are filtered out when several obstacle maps are combined.

Obstacle maps, generated by translating the depth maps and then integrating them over time to filter out erroneous obstacles, can be used for hazard avoidance. Chapter 5 shows various obstacles maps in which obstacles appear as large, black spots, and false alarms appear as small, gray spots. These obstacle maps can then be used by setting a threshold, above which an obstacle is said to be present. Alternately, one could set several thresholds, to which the Micro-Rover attaches varying levels of significance. Although these obstacle maps cannot reconstruct the environment, they should more than adequately provide the locations of obstacles or hazards.

The obstacle maps probably cannot be used for guiding a robotic arm to pick up an object, but it is expected that other sensors will be implemented in these situations. The errors in the depth estimates are inversely proportional to their length. The smallest error is therefore about 12.5% when 16 x 16 data blocks are used with 32 x 32 search windows. Additionally, the vision system is a *motion* vision system, i.e. the depth map can only be generated when there is motion between images, preferably motion that produces large flow vectors. However, it is expected that a robotic manipulator will have some tactile sensor that will assist it after the vision system guides it to a position that is close to the object of interest.

To provide depth information when the Micro-Rover is not moving, a binocular stereo system could be implemented using the same pattern matching approach. In this case, the motion is known because it is really the distance between the 2 camera positions. The depth values can be calculated directly, without iteration, by using the Least-Squares equations from Chapter 2.

One way of using this binocular stereo depth map is to select various depth values and use them to scale the depth map from the motion vision system. This is identical to how measured depth values from the laser range-finder would be incorporated into the vision system. Several measured depth values would be selected and subtracted from the depth values at the corresponding data blocks. The differences

would be squared and then summed. The Least-Squares estimate for the scaling factor minimizes the sum of squared errors when each of the depth map values is multiplied by this number.

An alternate way of using the binocular stereo depth map is to substitute it for the depth map from the motion vision system. In this case, solving for the motion of the Micro-Rover is no longer an iterative process. The Least-Squares equations for motion will yield direct estimates. This will considerably reduce the computational requirements of the microprocessor because all calculations are analytic.

A quick test of the ability of a binocular stereo system using the pattern matching approach is presented in Appendix E. It shows that binocular stereo could be used to provide some measurement of the environment and make the vision system an independent, complete sensor system.

The limited resolution of the flow vectors remains a stumbling block because it forces a level of discreteness on the vision system. There are 17 possible values for each component of a flow vector. This implies that there can be at most 289 different flow vectors. This can affect the final motion estimate, although using 196 different flow vectors should eliminate most of the effect. However, the depth estimates can only be very rough because there are only 289 different, discrete depth values. Additionally, if binocular stereo is used, the discreteness is even worse, as shown in Appendix E.

To reduce the effects of discreteness, a multi-resolution approach could be implemented to yield higher-resolution flow vectors. For example, assume that a 512 x 512 image were used. Every other pixel could be used to process a 256 x 256 image, as was done in this thesis. Then, the position of the best match could be used as a starting point in a higher-resolution matching. The first, 16 x 16 data block is actually a region in the 512 x 512 image that contains 32 x 32 pixels. These finer pixels could be divided into 4, 16 x 16 data blocks

and used to find the best match around the initial best match position. The resulting flow vector has effectively 33 possible values for each component.

The processing costs are very minimal since we now have 4 flow vectors for each data block and it is no longer necessary to calculate neighbor flow vectors. The original, lower-resolution 16 x 16 data block represents 4, higher-resolution, 16 x 16 data blocks, which can all be matched against an appropriate search window. This yields 4 flow vectors which can be used with a new weight to measure the quality of a match at the data block. The neighbor flow vectors, which were calculated so that a weight could be assigned to each data block, are no longer necessary.

In short, this thesis has shown that the pattern matching approach offers a viable alternative to the brightness gradient method that is commonly implemented because it produces comparable results at a fraction of the processing cost. The bulk of the processing can be performed in real-time on an integrated circuit chip originally designed for HDTV applications. The remaining number of calculations that must be performed by a microprocessor are about 2 orders of magnitude less than those required by the brightness gradient method. Thus, the pattern matching approach offers the ability to implement the motion estimation and obstacle map calculations in real-time on a PC-level processor at the expense of an additional piece of hardware.

Appendix A The Code That Performs the Matchings

The following code was extracted from TwinScreen™, the program that was written to test the pattern matching approach to motion vision. There are 2 main functions that perform the matchings. The first function is "Perform_Multiple_Matching()". It orchestrates the matching over an entire image. The variables for each data block and search window are set up in this function, and the second function is called to actually perform the matching. If the user requests that neighbor data blocks be processed, the second function is also called for each neighbor data block.

The second function is "Perform_Matching()". It actually calculates the match error between the data block and the search window for each possible position of the data block in the search window. If the data block is 16 x 16 and the search window is 32 x 32, this function processes 17 x 17, or 289 matches. The best match position as well as its error and the zero offset error are returned.

```

/*****
/***** #define's *****/
/*****/

#define NIL 0L

#define IMAGE_WIDTH  256    /* width of pixel map in pixels */
#define IMAGE_HEIGHT 216    /* height of a pixel map in pixels */

#define INFINITE_DEPTH 1000.0

/*****/
/***** camera calibration parameters *****/
/*****/

#define FOCAL_LENGTH      0.0085    /* principal distance in meters */
#define CCD_ARRAY_WIDTH  0.005     /* width of CCD array in meters */
#define PIXELS           256.0     /* horizontal resolution */
#define PRIN_POINT_Y     108.0     /* location of principal point */
#define PRIN_POINT_X     128.0     /* in terms of pixel elements */

/*****/
/***** typedef's *****/
/*****/
```

```

/* For each flow vector, the following data is stored: */
typedef struct FlowVectorRecord {
    Point theOffset;          /* the offset to the best match position */
    Point aveOffset; /* the ave best match offset with its 4 neighbors */
    long bestError;          /* the match error at the best match position */
    long aveError;           /* the average match error over all positions */
    long zeroError;         /* the match error at the zero offset position */
    int invVar;              /* the inverse variance weight */
    int offVar;              /* the offset to offset plus variance weight */
    int aveOffVar;          /* the ave offset to ave offset plus variance */
    int sumCosine;          /* the sum of cosines, or dot products, weight */
    int prodCosine;         /* the product of cosines weight */
    double depth;           /* the estimated depth, Z, at this data block */
} FlowVectorRecord, *FlowVectorPtr, **FlowVectorHandle;

```

```

/* For an image that contains an optical flow, this data is stored: */
typedef struct OpticalFlowRecord {
    Point topLeft;          /* top left corner of top left, or first, DB */
    Point botRight;        /* top left corner of bottom right, or last, DB */
    int dataHeight;         /* height of a data block */
    int dataWidth;          /* width of a data block */
    double U, V, W, A, B, C; /* motion parameters */
    Boolean neighborsUsed;   /* neighbor DB's used and weights calc'd? */
    Boolean depthMapUsed;   /* was a depth map calc'd? */
    Boolean showBoth;       /* show both arrows and values? */
    Boolean viewDepthMap;   /* show depth map or show weights? */
    Boolean showGoodVectors; /* show only vectors within range? */
    Boolean useDepthFilter; /* use depth values to filter? */
    double minWeight, maxWeight; /* min and max values displayed */
    double minDepth, maxDepth; /* min and max values displayed */
    int theWeight;          /* type of weight used for solving motion */
    FlowVectorHandle firstFlowVector; /* handle to FV record of 1st DB */
} OpticalFlowRecord, *OpticalFlowPtr, **OpticalFlowHandle;

```

```

/* Note: The flow vector records will be allocated in one contiguous
 * block with firstFlowVector being a handle to the first record. In
 * essence, we will have dynamically allocated an array in memory.
 */

```

```

/* For each image, the following information is stored: */
typedef struct ImageRecord {
    Str255 name;                /* name of window */
    WindowPtr imWindow;         /* pointer to image's window */
    ControlHandle horizScroll;  /* horizontal scroll bar */
    ControlHandle vertScroll;   /* vertical scroll bar */
    PixMapHandle imPixMap;      /* handle to pixel map */
    BitMap imBitMap;           /* bit map used if displaying on B&W monitor */
    Point displayPt;           /* pixel in top left corner of window */
    Rect imRect;                /* rect in pixel map displayed on screen */
    int magnification;          /* N by N screen pixels = 1 image pixel */
    struct ImageRecord **next; /* handle to next image record */
    OpticalFlowHandle flowData; /* handle to data used to draw arrows */
} ImageRecord, *ImagePtr, **ImageHandle;

```


The following function orchestrates the matching process over an entire image. The data block and search window positions are set-up and the function Perform_Matching is called to find the best match.

```

void Perform_Multiple_Match(saveMatchResults)
  Boolean saveMatchResults; /* should match results be saved to disk? */
{
  int dataHeight, dataWidth; /* height and width of data block */
  int searchHeight, searchWidth; /* height and width of search window */
  int dataRow, dataCol; /* coords of top left corner of data block */

  Point DB; /* number of data blocks vertically and horizontally */

  int continueMultiple, fileRefNum, i;

  FlowVectorHandle multipleArrayHnd; /* data block flow vector records */
  FlowVectorPtr currentDataBlockPtr; /* current flow vector record */

  FlowVectorHandle neighborArrayHnd; /* neighbor flow vector records */
  FlowVectorPtr currentNeighborPtr; /* current neighbor FV record */
  FlowVectorPtr topNeighborBlockPtr; /* top left neighbor FV record */
  FlowVectorPtr botNeighborBlockPtr; /* bottom left neighbor FV record */

  FlowVectorHandle newArrayHnd; /* handle to overlay result records */
  FlowVectorPtr newArrayPtr; /* pointer to overlay result records */

  OpticalFlowHandle flowDataHnd; /* temp handle to optical flow record */
  OpticalFlowPtr flowDataPtr; /* temp pointer to optical flow record */

  Rect theRect;
  Point topLeftNeighbor, botRightNeighbor;

  long bestMatch; /* best match error */
  long averageMatch; /* average match error over all positions */
  long zeroMatch; /* match error of zero offset position */

  long offsetPt_V, offsetPt_H; /* offset coords from Perform_Matching() */

  long sumOffsets_V, sumOffsets_H, sumSqrOffsets_V, sumSqrOffsets_H;
  double averageOffset_V, averageOffset_H; /* average flow vector */
  double offsetVariance_V, offsetVariance_H, offsetVariance;
  long offsetSqrD, lengthSqrD;
  double offsetLen, averageLen;
  double dotProduct, normalizingFactor, normalDotProduct;
  double sumCosines, prodCosines; /* sum and product of cosines weights */

  long currentTop_V, currentTop_H, currentBot_V, currentBot_H;

  long FVRsize; /* size of flow vector record in bytes */
  long OFRsize; /* size of optical flow record in bytes */

  /* find dimensions of data block and search window */
  Get_Image_Size(dataImage, &dataHeight, &dataWidth);
  Get_Image_Size(searchImage, &searchHeight, &searchWidth);

```

```

/* find number of data blocks vertically and horizontally */
DB.v = (botRight.v - topLeft.v) / dataHeight + 1;
DB.h = (botRight.h - topLeft.h) / dataWidth + 1;

Flow vector results are stored in memory referenced by multipleArrayHnd.

/* find size of flow vector record in bytes */
FVRsize = (long) sizeof(FlowVectorRecord);
/* allocate memory for DB.v times DB.h flow vector records */
multipleArrayHnd = (FlowVectorHandle)
    NewHandle(((long)DB.v)*((long)DB.h)*FVRsize);
if (!multipleArrayHnd) {
    Alert_User((UCP)"\pError allocating array for multiple results",
        (UCP)"\p", (UCP)"\pNot enough memory!", (UCP)"\p");
    return;
}

/* Allocate space for neighboring results array. If it can't be
 * allocated, don't calculate neighbor block results, but still
 * calculate data block results. Note that there is one more neighbor
 * block both vertically and horizontally.
 */
if (calcVariances) { /* compute neighboring data block matches? */
    neighborArrayHnd = (FlowVectorHandle)
        NewHandle(((long)DB.v+1L)*((long)DB.h+1L)*FVRsize);
    if (!neighborArrayHnd)
        Alert_User((UCP)"\pError allocating array for neighbor results",
            (UCP)"\p",
            (UCP)"\pData block matches will be calculated!",
            (UCP)"\p");
}
else
    neighborArrayHnd = 0L;

/* open file to save match results if requested */
if (saveMatchResults)
    if (!New_Match_File(&fileRefNum))
        saveMatchResults = FALSE;

```

```

/*****
/***** Perform Matching for Neighbor Blocks *****/
/*****

```

Find best match positions for the neighbor (staggered) data blocks.

```

if (neighborArrayHnd) {
    HLock(neighborArrayHnd);
    /* get pointer to first neighbor flow vector record */
    currentNeighborPtr = *neighborArrayHnd;
    /* offset data blocks 1/2 of a data block outward in all directions */
    topLeftNeighbor.v = topLeft.v - dataHeight/2;
    topLeftNeighbor.h = topLeft.h - dataWidth/2;
    botRightNeighbor.v = botRight.v + dataHeight/2;
    botRightNeighbor.h = botRight.h + dataWidth/2;

    /* perform match algorithm for all neighbor blocks */

```

```

for (dataRow=topLeftNeighbor.v;
     dataRow<=botRightNeighbor.v; dataRow+=dataHeight)
  for (dataCol=topLeftNeighbor.h;
       dataCol<=botRightNeighbor.h; dataCol+=dataWidth) {

  /* adjust data block and search block positions */
  theRect.top = dataRow;
  theRect.left = dataCol;
  theRect.bottom = dataRow + dataHeight;
  theRect.right = dataCol + dataWidth;
  Set_Image_Rect(dataImage, &theRect);
  theRect.top = dataRow - offSet.v;
  theRect.left = dataCol - offSet.h;
  theRect.bottom = theRect.top + searchHeight;
  theRect.right = theRect.left + searchWidth;
  Set_Image_Rect(searchImage, &theRect);
  /* Perform matching for selected data and search rectangles. */
  continueMultiple = Perform_Matching(&zeroMatch, &bestMatch,
                                       &offsetPt_V, &offsetPt_H, &averageMatch);
  /* If there was an error, return to main program. */
  if (!continueMultiple) {
    Trash_Image(multipleResult);
    Trash_Image(overlayResult);
    DisposHandle(multipleArrayHnd);
    DisposHandle(neighborArrayHnd);
    return;
  }
  /* save offsetPt in "neighborArray" */
  currentNeighborPtr->theOffset.v = offsetPt_V;
  currentNeighborPtr->theOffset.h = offsetPt_H;
  currentNeighborPtr++; /* point to next flow vector record */
}
}

```

```

/*****
/***** Perform Matching for Requested Data Blocks *****/
/*****

```

Find best match positions for the actual data blocks.

```

HLock(multipleArrayHnd);

/* get pointer to first data block flow vector record */
currentDataBlockPtr = *multipleArrayHnd;

for (dataRow=topLeft.v; dataRow<=botRight.v; dataRow+=dataHeight)
  for (dataCol=topLeft.h; dataCol<=botRight.h; dataCol+=dataWidth) {

  /* indenting is shifted to the left starting here */

  /* adjust data block and search block positions */
  theRect.top = dataRow;
  theRect.left = dataCol;
  theRect.bottom = dataRow + dataHeight;
  theRect.right = dataCol + dataWidth;
  Set_Image_Rect(dataImage, &theRect);
  theRect.top = dataRow - offSet.v;

```

```

theRect.left = dataCol - offSet.h;
theRect.bottom = theRect.top + searchHeight;
theRect.right = theRect.left + searchWidth;
Set_Image_Rect(searchImage, &theRect);

/* Perform matching for selected data and search rectangles. */
continueMultiple = Perform_Matching(&zeroMatch, &bestMatch,
                                     &offsetPt_V, &offsetPt_H, &averageMatch);
if (!continueMultiple) { /* if there was an error */
    Trash_Image(multipleResult);
    Trash_Image(overlayResult);
    DisposHandle(multipleArrayHnd);
    if (neighborArrayHnd) DisposHandle(neighborArrayHnd);
    return;
}
/* force match error to be at least one */
if (bestMatch < 1)
    bestMatch = 1;
if (zeroMatch < 1)
    zeroMatch = 1;
/* save results in flow vector handle */
currentDataBlockPtr->theOffset.v = (int) offsetPt_V;
currentDataBlockPtr->theOffset.h = (int) offsetPt_H;
currentDataBlockPtr->bestError = bestMatch;
currentDataBlockPtr->aveError = averageMatch;
currentDataBlockPtr->zeroError = zeroMatch;

/* calculate other record elements if neighbor blocks were processed */
if (neighborArrayHnd) {
    /* get pointer to top left neighbor flow vector record */
    topNeighborBlockPtr = *neighborArrayHnd;
    topNeighborBlockPtr += (long)
        ((dataRow-topLeft.v)/dataHeight*(DB.h+1));
    topNeighborBlockPtr += (long) ((dataCol-topLeft.h)/dataWidth);
    /* get pointer to bottom left neighbor flow vector record */
    botNeighborBlockPtr = *neighborArrayHnd;
    botNeighborBlockPtr += (long)
        (((dataRow-topLeft.v)/dataHeight+1)*(DB.h+1));
    botNeighborBlockPtr += (long) ((dataCol-topLeft.h)/dataWidth);

    /* initialize cosine weights */
    sumCosines = 0.0; /* sum of normalized dot products */
    prodCosines = 1.0; /* product of normalized dot products */
    /* initialize sum of flow vectors and sum of squared flow vectors */
    sumOffsets_V = offsetPt_V;
    sumOffsets_H = offsetPt_H;
    sumSqrOffsets_V = SquareLong(offsetPt_V);
    sumSqrOffsets_H = SquareLong(offsetPt_H);
    /* find length of data block flow vector */
    offsetSqrD = SquareLong(offsetPt_V) + SquareLong(offsetPt_H);
    offsetLen = sqrt((double) offsetSqrD);

    /* incorporate 4 neighbor flow vectors */
    for (i=0; i<2; i++){/* process left neighbors, then right neighbors */
        /* get top and bottom flow vectors */
        currentTop_V = (long) topNeighborBlockPtr->theOffset.v;
        currentTop_H = (long) topNeighborBlockPtr->theOffset.h;

```

```

currentBot_V = (long) botNeighborBlockPtr->theOffset.v;
currentBot_H = (long) botNeighborBlockPtr->theOffset.h;
/* add vertical and horizontal components to sum of flow vectors */
sumOffsets_V += currentTop_V + currentBot_V;
sumOffsets_H += currentTop_H + currentBot_H;
/* add vert and horiz components to sum of squared flow vectors */
sumSqrOffsets_V+=SquareLong(currentTop_V)+SquareLong(currentBot_V);
sumSqrOffsets_H+=SquareLong(currentTop_H)+SquareLong(currentBot_H);
/* compute dot product of top neighbor and data block FVs */
dotProduct = (double)
    (offsetPt_V*currentTop_V+offsetPt_H*currentTop_H);
/* compute squared length of neighbor FV */
lengthSqrD = SquareLong(currentTop_V) + SquareLong(currentTop_H);
/* compute product of neighbor FV length and data block FV length */
normalizingFactor = sqrt((double) lengthSqrD)*offsetLen;
/* normalize dot product */
if (normalizingFactor > 0.0)
    normalDotProduct = dotProduct/normalizingFactor;
else
    normalDotProduct = 0.0;
/* add dot product of top neighbor and BD FVs to "sumCosines" */
sumCosines += normalDotProduct;
/* multiply "prodCosines" by dot product */
prodCosines *= normalDotProduct;
/* if dot prod < 0, i.e. FVs were > 90 deg apart, prodCosines = 0 */
if (prodCosines < 0.0)
    prodCosines = 0.0;
/* do the same thing for bottom neighbor FV */
dotProduct = (double)
    (offsetPt_V*currentBot_V+offsetPt_H*currentBot_H);
lengthSqrD = SquareLong(currentBot_V) + SquareLong(currentBot_H);
normalizingFactor = sqrt((double) lengthSqrD)*offsetLen;
if (normalizingFactor > 0.0)
    normalDotProduct = dotProduct/normalizingFactor;
else
    normalDotProduct = 0.0;
sumCosines += normalDotProduct;
prodCosines *= normalDotProduct;
if (prodCosines < 0.0)
    prodCosines = 0.0;
/* increment pointers to neighbor blocks */
topNeighborBlockPtr++;
botNeighborBlockPtr++;
}
/* compute components of average flow vector */
averageOffset_V = ((double) sumOffsets_V) / 5.0;
averageOffset_H = ((double) sumOffsets_H) / 5.0;
/* compute length of average flow vector */
averageLen = sqrt((double) (averageOffset_V*averageOffset_V
    + (double) (averageOffset_H*averageOffset_H)));
/* compute components of variance of the 5 flow vectors */
offsetVariance_V = ((double)sumSqrOffsets_V)/5.0
    - averageOffset_V*averageOffset_V;
offsetVariance_H = ((double)sumSqrOffsets_H)/5.0
    - averageOffset_H*averageOffset_H;
/* compute combined variance */
offsetVariance = offsetVariance_H + offsetVariance_V;
/* store components of average flow vector */

```

```

if (averageOffset_V >= 0.0)
    currentDataBlockPtr->aveOffset.v = (int) (averageOffset_V + 0.5);
else
    currentDataBlockPtr->aveOffset.v = (int) (averageOffset_V - 0.5);
if (averageOffset_H >= 0.0)
    currentDataBlockPtr->aveOffset.h = (int) (averageOffset_H + 0.5);
else
    currentDataBlockPtr->aveOffset.h = (int) (averageOffset_H - 0.5);
/* store inverse variance weight in memory */
if (offsetVariance != 0.0) /* can't divide by zero */
    currentDataBlockPtr->invVar = (int) (100.0/offsetVariance + 0.5);
else
    currentDataBlockPtr->invVar = 100;
/* store offset to offset plus variance weight */
if (offsetLen != 0.0) /* weight equals zero if DB FV is zero */
    currentDataBlockPtr->offVar = (int)
        (100.0*offsetLen/(offsetLen+offsetVariance) + 0.5);
else
    currentDataBlockPtr->offVar = 0;
/* store average offset to average offset plus variance weight */
if (averageLen != 0.0) /* weight equals zero if average FV is zero */
    currentDataBlockPtr->aveOffVar = (int)
        (100.0*averageLen/(averageLen+offsetVariance) + 0.5);
else
    currentDataBlockPtr->aveOffVar = 0;
/* store sum of cosines weight */
if (sumCosines >= 0.0) /* average, then take absolute value */
    currentDataBlockPtr->sumCosine = (int)(100.0*(sumCosines/4.0)+0.5);
else
    currentDataBlockPtr->sumCosine = (int)(100.0*(-sumCosines/4.0)+0.5);
/* store product of cosines weight */
if (prodCosines >= 0.0)
    currentDataBlockPtr->prodCosine = (int) (100.0*prodCosines + 0.5);
else
    currentDataBlockPtr->prodCosine = 0;
} /* end if (neighborArrayHnd) */

else { /* neighbor blocks were not processed, so zero out weights */
    currentDataBlockPtr->aveOffset.v = (int) offsetPt_V;
    currentDataBlockPtr->aveOffset.h = (int) offsetPt_H;
    currentDataBlockPtr->invVar = 0;
    currentDataBlockPtr->offVar = 0;
    currentDataBlockPtr->aveOffVar = 0;
    currentDataBlockPtr->sumCosine = 0;
    currentDataBlockPtr->prodCosine = 0;
}

/* initialized depth to one */
currentDataBlockPtr->depth = 1.0;
/* increment pointer to data block flow vector */
currentDataBlockPtr++;

} /* end perform matching for requested data blocks */

```

```

/***** Save Arrows Data for Multiple Results Image *****/
/***** Save Arrows Data for Multiple Results Image *****/
/***** Save Arrows Data for Multiple Results Image *****/

```

Flow vector results are stored for the optical flow image.

```

/* get size of flow data record */
OFRsize = (long) sizeof(OpticalFlowRecord);
/* allocate space for a new flow data record */
flowDataHnd = (OpticalFlowHandle) NewHandle(OFRsize);
if (flowDataHnd) { /* if a record has been allocated... */
    /* assign flow data record to multiple result image */
    (*multipleResult)->flowData = flowDataHnd;
    HLock(flowDataHnd);
    flowDataPtr = *flowDataHnd;
    /* store record elements */
    flowDataPtr->topLeft.v = topLeft.v;
    flowDataPtr->topLeft.h = topLeft.h;
    flowDataPtr->botRight.v = botRight.v;
    flowDataPtr->botRight.h = botRight.h;
    flowDataPtr->dataHeight = dataHeight;
    flowDataPtr->dataWidth = dataWidth;
    flowDataPtr->U = 0.0; /* translation components */
    flowDataPtr->V = 0.0;
    flowDataPtr->W = 1.0;
    flowDataPtr->A = 0.0; /* rotation components */
    flowDataPtr->B = 0.0;
    flowDataPtr->C = 0.0;
    flowDataPtr->theWeight = ZERO_ERROR_TO_BEST_ERROR;
    /* mark whether neighbor blocks were used */
    if (neighborArrayHnd)
        flowDataPtr->neighborsUsed = TRUE;
    else
        flowDataPtr->neighborsUsed = FALSE;
    /* depth map calc'd is FALSE since a new optical flow was created */
    flowDataPtr->depthMapUsed = FALSE;
    /* mark show depth map as FALSE since we want weights displayed */
    flowDataPtr->showBoth = TRUE;
    flowDataPtr->viewDepthMap = FALSE;
    /* show good vectors is FALSE since we want to display all vectors */
    flowDataPtr->showGoodVectors = FALSE;
    flowDataPtr->useDepthFilter = FALSE;

    /* set minimum and maximum range for weights */
    flowDataPtr->minDepth = 0.0;
    flowDataPtr->maxDepth = 10000.0;
    flowDataPtr->minWeight = 0.0;
    flowDataPtr->maxWeight = 10000.0;

    /* assign "multipleArrayHnd" FV records to multiple result image */
    flowDataPtr->firstFlowVector = multipleArrayHnd;

    HUnlock(flowDataHnd);
} /* end if (flowDataHnd) */

```

```

/*****
/***** Save Arrows Data for Overlay Results Image *****/
/*****

```

Flow vector results are stored for the overlay results image.

```

/* allocate space for a new flow data record */
flowDataHnd = (OpticalFlowHandle) NewHandle(OFRsize);
if (flowDataHnd) { /* if a record has been allocated... */
    /* assign flow data record to overlay result image */
    (*overlayResult)->flowData = flowDataHnd;
    HLock(flowDataHnd);
    flowDataPtr = *flowDataHnd;
    /* store record elements */
    flowDataPtr->topLeft.v = topLeft.v;
    flowDataPtr->topLeft.h = topLeft.h;
    flowDataPtr->botRight.v = botRight.v;
    flowDataPtr->botRight.h = botRight.h;
    flowDataPtr->dataHeight = dataHeight;
    flowDataPtr->dataWidth = dataWidth;
    flowDataPtr->U = 0.0;
    flowDataPtr->V = 0.0;
    flowDataPtr->W = 1.0;
    flowDataPtr->A = 0.0;
    flowDataPtr->B = 0.0;
    flowDataPtr->C = 0.0;
    flowDataPtr->theWeight = ZERO_ERROR_TO_BEST_ERROR;
    /* mark whether neighbor blocks were used */
    if (neighborArrayHnd)
        flowDataPtr->neighborsUsed = TRUE;
    else
        flowDataPtr->neighborsUsed = FALSE;
    /* depth map calc'd is FALSE since a new optical flow was created */
    flowDataPtr->depthMapUsed = FALSE;
    /* mark show depth map as FALSE since we want weights displayed */
    flowDataPtr->showBoth = TRUE;
    flowDataPtr->viewDepthMap = FALSE;
    /* show good vectors is FALSE since we want to display all vectors */
    flowDataPtr->showGoodVectors = FALSE;
    flowDataPtr->useDepthFilter = FALSE;

    /* set minimum and maximum range for weights */
    flowDataPtr->minDepth = 0.0;
    flowDataPtr->maxDepth = 10000.0;
    flowDataPtr->minWeight = 0.0;
    flowDataPtr->maxWeight = 10000.0;

    /* allocate space for flow vector records */
    newArrayHnd = (FlowVectorHandle)
        NewHandle(((long)DB.v)*((long)DB.h)*FVRsize);
    if (newArrayHnd) {
        flowDataPtr->firstFlowVector = newArrayHnd;
        HLock(newArrayHnd);
        newArrayPtr = *newArrayHnd;
        /* copy flow vector record data from "multipleArrayHnd" */
        newArrayPtr = *newArrayHnd;
        currentDataBlockPtr = *multipleArrayHnd;
        for (dataRow=0; dataRow<DB.v; dataRow++)

```



```

for (dataCol=0; dataCol<DB.h; dataCol++) {
    /* copy record elements */
    newArrayPtr->theOffset.v = currentDataBlockPtr->theOffset.v;
    newArrayPtr->theOffset.h = currentDataBlockPtr->theOffset.h;
    newArrayPtr->aveOffset.v = currentDataBlockPtr->aveOffset.v;
    newArrayPtr->aveOffset.h = currentDataBlockPtr->aveOffset.h;
    newArrayPtr->bestError = currentDataBlockPtr->bestError;
    newArrayPtr->aveError = currentDataBlockPtr->aveError;
    newArrayPtr->zeroError = currentDataBlockPtr->zeroError;
    newArrayPtr->invVar = currentDataBlockPtr->invVar;
    newArrayPtr->offVar = currentDataBlockPtr->offVar;
    newArrayPtr->aveOffVar = currentDataBlockPtr->aveOffVar;
    newArrayPtr->sumCosine = currentDataBlockPtr->sumCosine;
    newArrayPtr->prodCosine = currentDataBlockPtr->prodCosine;
    newArrayPtr->depth = currentDataBlockPtr->depth;
    /* get pointer to next flow vector records */
    newArrayPtr++;
    currentDataBlockPtr++;
}
} /* end if (mewArrayHnd) */
else
    (*overlayResult)->flowData = 0L;

HUnlock(flowDataHnd);
} /* end if (flowDataHnd) */

```

```

/*****
/***** Save Match Results to Disk *****/
/*****

```

Save flow vector results to disk.

```

if (saveMatchResults) {
    /* the match results will be stored in a file as follows:

    theOffset.v(1,1)    theOffset.v(1,2)    ... theOffset.v(1,DB.h)
    theOffset.h         theOffset.h         ... theOffset.h
    aveOffset.v         aveOffset.v         ... aveOffset.v
    aveOffset.h         aveOffset.h         ... aveOffset.h
    bestMatch          bestMatch          ... bestMatch
    aveMatch            aveMatch            ... aveMatch
    zeroMatch          zeroMatch          ... zeroMatch
    invVar             invVar             ... invVar
    offVar             offVar             ... offVar
    aveOffVar          aveOffVar          ... aveOffVar
    sumCosine          sumCosine          ... sumCosine
    prodCosine         prodCosine         ... prodCosine

    theOffset.v(2,1)    theOffset.v(2,2)    ... theOffset.v(2,DB.h)
    theOffset.h         theOffset.h         ... theOffset.h
    aveOffset.v         aveOffset.v         ... aveOffset.v
    aveOffset.h         aveOffset.h         ... aveOffset.h
    bestMatch          bestMatch          ... bestMatch
    aveMatch            aveMatch            ... aveMatch
    zeroMatch          zeroMatch          ... zeroMatch
    invVar             invVar             ... invVar
    offVar             offVar             ... offVar

```

```

aveOffVar          aveOffVar          ... aveOffVar
sumCosine          sumCosine          ... sumCosine
prodCosine         prodCosine         ... prodCosine

...

theOffset.v(DB.v,1)  theOffset.v(DB.v,2)  ... theOffset.v(DB.v,DB.h)
theOffset.h         theOffset.h         ... theOffset.h
aveOffset.v         aveOffset.v         ... aveOffset.v
aveOffset.h         aveOffset.h         ... aveOffset.h
bestMatch          bestMatch          ... bestMatch
aveMatch           aveMatch           ... aveMatch
zeroMatch          zeroMatch          ... zeroMatch
invVar             invVar             ... invVar
offVar            offVar            ... offVar
aveOffVar          aveOffVar          ... aveOffVar
sumCosine          sumCosine          ... sumCosine
prodCosine         prodCosine         ... prodCosine

```

with tabs between entries on a line and returns between lines. */

```

for (dataRow=0; dataRow<DB.v; dataRow++) {
  currentDataBlockPtr = *multipleArrayHnd + ((long)DB.h*(long)dataRow);
  for (dataCol=0; dataCol<DB.h; dataCol++) {
    Write_Int(fileRefNum, currentDataBlockPtr++->theOffset.v);
    Write_Tab(fileRefNum);
  }
  Write_Return(fileRefNum);
  currentDataBlockPtr = *multipleArrayHnd + ((long)DB.h*(long)dataRow);
  for (dataCol=0; dataCol<DB.h; dataCol++) {
    Write_Int(fileRefNum, currentDataBlockPtr++->theOffset.h);
    Write_Tab(fileRefNum);
  }
  Write_Return(fileRefNum);
  currentDataBlockPtr = *multipleArrayHnd + ((long)DB.h*(long)dataRow);
  for (dataCol=0; dataCol<DB.h; dataCol++) {
    Write_Int(fileRefNum, currentDataBlockPtr++->aveOffset.v);
    Write_Tab(fileRefNum);
  }
  Write_Return(fileRefNum);
  currentDataBlockPtr = *multipleArrayHnd + ((long)DB.h*(long)dataRow);
  for (dataCol=0; dataCol<DB.h; dataCol++) {
    Write_Int(fileRefNum, currentDataBlockPtr++->aveOffset.h);
    Write_Tab(fileRefNum);
  }
  Write_Return(fileRefNum);
  currentDataBlockPtr = *multipleArrayHnd + ((long)DB.h*(long)dataRow);
  for (dataCol=0; dataCol<DB.h; dataCol++) {
    Write_Long(fileRefNum, currentDataBlockPtr++->bestError);
    Write_Tab(fileRefNum);
  }
  Write_Return(fileRefNum);
  currentDataBlockPtr = *multipleArrayHnd + ((long)DB.h*(long)dataRow);
  for (dataCol=0; dataCol<DB.h; dataCol++) {
    Write_Long(fileRefNum, currentDataBlockPtr++->aveError);
    Write_Tab(fileRefNum);
  }
  Write_Return(fileRefNum);
}

```

```

currentDataBlockPtr = *multipleArrayHnd + ((long)DB.h*(long)dataRow);
for (dataCol=0; dataCol<DB.h; dataCol++) {
    Write_Long(fileRefNum, currentDataBlockPtr++->zeroError);
    Write_Tab(fileRefNum);
}
Write_Return(fileRefNum);
currentDataBlockPtr = *multipleArrayHnd + ((long)DB.h*(long)dataRow);
for (dataCol=0; dataCol<DB.h; dataCol++) {
    Write_Int(fileRefNum, currentDataBlockPtr++->invVar);
    Write_Tab(fileRefNum);
}
Write_Return(fileRefNum);
currentDataBlockPtr = *multipleArrayHnd + ((long)DB.h*(long)dataRow);
for (dataCol=0; dataCol<DB.h; dataCol++) {
    Write_Int(fileRefNum, currentDataBlockPtr++->offVar);
    Write_Tab(fileRefNum);
}
Write_Return(fileRefNum);
currentDataBlockPtr = *multipleArrayHnd + ((long)DB.h*(long)dataRow);
for (dataCol=0; dataCol<DB.h; dataCol++) {
    Write_Int(fileRefNum, currentDataBlockPtr++->aveOffVar);
    Write_Tab(fileRefNum);
}
Write_Return(fileRefNum);
currentDataBlockPtr = *multipleArrayHnd + ((long)DB.h*(long)dataRow);
for (dataCol=0; dataCol<DB.h; dataCol++) {
    Write_Int(fileRefNum, currentDataBlockPtr++->sumCosine);
    Write_Tab(fileRefNum);
}
Write_Return(fileRefNum);
currentDataBlockPtr = *multipleArrayHnd + ((long)DB.h*(long)dataRow);
for (dataCol=0; dataCol<DB.h; dataCol++) {
    Write_Int(fileRefNum, currentDataBlockPtr++->prodCosine);
    Write_Tab(fileRefNum);
}
Write_Return(fileRefNum);
Write_Return(fileRefNum);
} /* end for dataRow */
Close_File(fileRefNum);
} /* if (saveMatchResults) */

/* unlock multiple array (multiple result) so it can be relocated */
HUnlock(multipleArrayHnd);
/* unlock new array (overlay result) so it can be relocated in memory */
HUnlock(newArrayHnd);
/* unlock neighbor array and dispose of it */
HUnlock(neighborArrayHnd);
DisposHandle(neighborArrayHnd);

} /* end function Perform_Multiple_Match */

```

The following function finds the best match of a data block in a given search window.

```

/* This function performs the matching between the DB and the SW. */
int Perform_Matching(zeroOffset, bestMatch, offsetPt_V,

```

```

                                offsetPt_H, averageMatch)
long *zeroOffset, *bestMatch, *offsetPt_V, *offsetPt_H, *averageMatch;
{
    PixMapHandle dataPixMap, searchPixMap, matchPixMap;
    Rect dataRect, searchRect;
    int dataHeight, dataWidth;
    int searchHeight, searchWidth;
    int matchHeight, matchWidth;
    long topLeftDataPixel, topLeftSearchPixel;
    long vertOffset, horizOffset;
    long temp, searchAdjmnt;
    char *startDataPixel, *currentDataPixel;
    char *startSearchPixel, *currentSearchPixel;
    char *startMatchPixel, *currentMatchPixel;
    int deltaV, deltaH, dataV, dataH;
    long *matchArray, *currentCell;
    long highestValue, difference;
    double fractionPixel, newAverage;
    Point toZero;

```

The data and search images contain the pixel patterns to be matched. The pixel maps store each pixel as a byte. The byte corresponds to the brightness value of the pixel.

```

/* check if data and search windows have been opened */
if ((!dataImage)||(!searchImage)) {
    Alert_User((UCP)"\pData and search windows must be", (UCP)"\p",
              (UCP)"\popened to perform matchings!", (UCP)"\p");
    return(0);
}

/* find dimensions of match results */
Get_Image_Size(dataImage, &dataHeight, &dataWidth);
Get_Image_Size(searchImage, &searchHeight, &searchWidth);
matchHeight = searchHeight - dataHeight + 1;
matchWidth = searchWidth - dataWidth + 1;

```

MatchResult is a handle to an image record which has a pixel map as one of its fields. The pixel map stores the match errors as pixels where each pixel has been normalized between 0 and 127 - the gray scales used in this program.

```

/* if the matching window has not been opened... */
if (!matchResult)
    if (!Open_Window(&matchResult, MATCH_WINDOW, FALSE,
                    matchHeight, matchWidth, 8))
        return(0);

/* allocate space for matching array. Each cell is a 4 byte integer. */
matchArray = (LongPtr)
              NewPtr(((long)matchWidth)*((long)matchHeight)*4L+1000L);
if (!matchArray) {
    Alert_User((UCP)"\pError allocating array for matchings", (UCP)"\p",
              (UCP)"\pNot enough memory!", (UCP)"\p");
    return(0);
}

/* highest value of matching is used for normalization - init to 1 */
highestValue = 1L;
/* get data and search rectangles to later find byte offsets */

```

```

    Get_Image_Rect(dataImage, &dataRect);
    Get_Image_Rect(searchImage, &searchRect);
/* get handles to data and search pixel maps */
    dataPixMap = (*dataImage)->imPixMap;
    searchPixMap = (*searchImage)->imPixMap;
    matchPixMap = (*matchResult)->imPixMap;
/* compute starting address in data pixel map */
    horizOffset = (long) dataRect.left;
    vertOffset = ((long) dataRect.top) * ((long) IMAGE_WIDTH);
    topLeftDataPixel = (long) (*dataPixMap)->baseAddr;
    topLeftDataPixel += vertOffset + horizOffset;
/* compute starting address in search pixel map */
    horizOffset = (long) searchRect.left;
    vertOffset = ((long) searchRect.top) * ((long) IMAGE_WIDTH);
    topLeftSearchPixel = (long) (*searchPixMap)->baseAddr;
    topLeftSearchPixel += vertOffset + horizOffset;
/* cycle through all possible DB positions in the search window */
    for (deltaV=0; deltaV<matchHeight; deltaV++)
        for (deltaH=0; deltaH<matchWidth; deltaH++) {
            startDataPixel = (CharPtr) topLeftDataPixel;
            searchAdjmnt = (long) (deltaV * IMAGE_WIDTH + deltaH);
            startSearchPixel = (CharPtr) (topLeftSearchPixel + searchAdjmnt);
/* compute cell address in matching array */
            temp = (long) matchArray;
            temp += 4L * (long) (deltaV * matchWidth + deltaH);
            currentCell = (LongPtr) temp;
/* clear current cell */
            *currentCell = 0L;
/* cycle through all data pixels */
            for (dataV = 0; dataV < dataHeight; dataV++)
                for (dataH = 0; dataH < dataWidth; dataH++) {
/* find address of current data pixel */
                    temp = (long) startDataPixel;
                    temp += (long) (dataV * IMAGE_WIDTH + dataH);
                    currentDataPixel = (CharPtr) temp;
/* find address of current search pixel */
                    temp = (long) startSearchPixel;
                    temp += (long) (dataV * IMAGE_WIDTH + dataH);
                    currentSearchPixel = (CharPtr) temp;
/* compute difference between pixel values */
                    difference = ((long)*currentSearchPixel) -
                                ((long)*currentDataPixel);

                    if (difference < 0)
                        difference = -difference;
                    *currentCell += difference;
                }
            if (*currentCell > highestValue)
                highestValue = *currentCell;
        }
}

/* initialize best match, zero offset, and offset point */
toZero.h = matchWidth / 2; /* zero match in center of search window */
if (matchWidth % 2 == 0) toZero.h -= 1;
toZero.v = matchHeight / 2; /* zero match in center of search window */
if (matchHeight % 2 == 0) toZero.v -= 1;
temp = (long) matchArray;
temp += 4L * (long) (toZero.v * matchWidth + toZero.h);
*zeroOffset = *((LongPtr) temp);

```

```

*bestMatch = *zeroOffset;
*offsetPt_V = 0L;
*offsetPt_H = 0L;
newAverage = 0.0;
/* compute starting address in matching pixel map */
temp = (long) (*matchPixMap)->baseAddr;
startMatchPixel = (CharPtr) temp;
for (deltaV=0; deltaV<matchHeight; deltaV++)
    for (deltaH=0; deltaH<matchWidth; deltaH++) {
        /* compute current matching pixel address */
        temp = (long) startMatchPixel;
        temp += (long) (deltaV * IMAGE_WIDTH + deltaH);
        currentMatchPixel = (CharPtr) temp;
        /* compute cell address in matching array */
        temp = (long) matchArray;
        temp += 4L * (long) (deltaV * matchWidth + deltaH);
        currentCell = (LongPtr) temp;
        /* check if current cell is less than the best match */
        if (*currentCell < *bestMatch) {
            *offsetPt_V = deltaV - toZero.v;
            *offsetPt_H = deltaH - toZero.h;
            *bestMatch = *currentCell;
        }
        newAverage = newAverage * ((double)((deltaV*matchWidth)+deltaH));
        newAverage += (double) *currentCell;
        newAverage /= ((double) ((deltaV*matchWidth) + deltaH) + 1);
        /* normalize current cell */
        fractionPixel = ((double)*currentCell) / ((double)highestValue);
        /* convert to byte value and store in matching pixel map */
        *currentMatchPixel = (char) (127.0 * fractionPixel);
    }

/* dispose of pointer to match array */
DisposPtr(matchArray);
/* unlock pixel maps and images */
HUnlock(dataPixMap);
HUnlock(searchPixMap);
HUnlock(matchPixMap);
HUnlock(dataImage);
HUnlock(searchImage);
HUnlock(matchResult);

/* set average */
*averageMatch = (long) (newAverage+0.5);
/* invalidate match window */
Inval_Image(matchResult);
return(1);
} /* end function Perform_Matching() */

```

Appendix B The Code That Calculates Motion

The following code was used to perform the iteration to solve for the motion. The iteration for the motion is performed in steps. At each step, the user may choose to update the depth map from the current motion estimate and then compute a new motion estimate from the updated depth map. Alternatively, the user may opt to scale the depth map to simulate the vision system's response to a sensor, such as a laser range-finder, that makes some measure on the environment. Additionally, the user may enter an estimate of the vehicle's motion and then continue the iteration based upon this starting value.

```

/*****
/***** typedef for data block record *****/
/*****
/*****

typedef struct {
    double u,v; /* flow vector */
    double x,y; /* data block coordinates */
    double s33; /* elements of S matrix */
    double q11,q12,q21,q31,q32; /* elements of Q matrix */
    double r11,r12,r22; /* elements of R matrix */
    double Z; /* depth to scene */
    double W; /* data block weight */
} DataBlockRecord, *DataBlockPtr, **DataBlockHnd;

/*****
/***** global variables for this file *****/
/*****

Point topLf, botRt, numDBs; /* boundaries of processing and # of DB's */
int dataHeight, dataWidth; /* height and width of data blocks */

ImageHandle theMatchResult; /* the image which contains the FV's */

OpticalFlowHandle theOpticalFlowHnd; /* handle to optical flow data */

FlowVectorHandle currentDataBlockHnd; /* handle to flow vector array */

long DBRsize; /* size of data block record in bytes */
DataBlockHnd theDataBlockHnd; /* handle to DB data records */
DataBlockPtr theDataBlockPtr; /* pointer to DB data records */

double PELwidth, PELfactor; /* pixel dimension factors */

double U,V,W; /* translation vector components */
double A,B,C; /* rotation vector components */

```

```

double theMin, theMax;
Boolean showGoodVectors;

int theWeight;

```

This function puts up the dialog box which the user can use to solve the motion parameters, set the motion parameters, set the depth to a constant value, and scale the depth.

```

void Solve_Motion_Dialog()
{
    OpticalFlowPtr theOpticalFlowPtr;
    DialogPtr theDialog;
    GrafPtr currentPort;
    int itemHit; /* dialog item hit */
    int itemType; /* type of dialog item */
    int theInteger; /* item text converted to integer */
    Handle theItem; /* handle to dialog item */
    Rect box;
    Str255 theText;
    Boolean constantDepthBox, scaleFactorBox, saveMotionBox, newMotionBox;
    double constantDepth, scaleFactor, temp;
    Boolean U_OK, V_OK, W_OK, A_OK, B_OK, C_OK, CONST_OK, SCALE_OK;

    if (!Init_Solve_Array()) return;

    /* put up dialog box */
    theDialog = GetNewDialog(SOLVE_DIALOG, NIL, (WindowPtr)(-1));
    if (!theDialog) {
        Alert_User((UCP)"\pSolve Motion dialog could not be opened.",
            (UCP)"\p", (UCP)"\pNot enough memory!", (UCP)"\p");
        return;
    }

    /* get initial values for motion parameters */
    HLock(theOpticalFlowHnd);
    theOpticalFlowPtr = *theOpticalFlowHnd;
    U = theOpticalFlowPtr->U;
    V = theOpticalFlowPtr->V;
    W = theOpticalFlowPtr->W;
    A = theOpticalFlowPtr->A;
    B = theOpticalFlowPtr->B;
    C = theOpticalFlowPtr->C;
    HUnlock(theOpticalFlowHnd);

    /* initialize other parameters */
    constantDepth = 1.0;
    scaleFactor = 1.0;

    /* place current translation and rotation in boxes */
    GetDItem(theDialog, TRANS_U, &itemType, &theItem, &box);
    Double_To_String(U, theText, 3);
    SetIText(theItem, theText);
    GetDItem(theDialog, TRANS_V, &itemType, &theItem, &box);
    Double_To_String(V, theText, 3);

```



```

SetIText(theItem, theText);
GetDItem(theDialog, TRANS_W, &itemType, &theItem, &box);
Double_To_String(W, theText, 3);
SetIText(theItem, theText);
GetDItem(theDialog, ROT_A, &itemType, &theItem, &box);
Double_To_String(A, theText, 3);
SetIText(theItem, theText);
GetDItem(theDialog, ROT_B, &itemType, &theItem, &box);
Double_To_String(B, theText, 3);
SetIText(theItem, theText);
GetDItem(theDialog, ROT_C, &itemType, &theItem, &box);
Double_To_String(C, theText, 3);
SetIText(theItem, theText);
/* select U component of translation */
SetIText(theDialog, TRANS_U, 0, 32767);

/* place other values in boxes */
GetDItem(theDialog, CONST_DEPTH_VALUE, &itemType, &theItem, &box);
Double_To_String(constantDepth, theText, 1);
SetIText(theItem, theText);
GetDItem(theDialog, SCALE_DEPTH_VALUE, &itemType, &theItem, &box);
Double_To_String(scaleFactor, theText, 1);
SetIText(theItem, theText);

/* initialize constant depth check box */
constantDepthBox = TRUE;
GetDItem(theDialog, CONST_DEPTH_CHECK, &itemType, &theItem, &box);
SetCtlValue(theItem, (int) constantDepthBox);
/* initialize scale factor check box */
scaleFactorBox = FALSE;
GetDItem(theDialog, SCALE_DEPTH_CHECK, &itemType, &theItem, &box);
SetCtlValue(theItem, (int) scaleFactorBox);
/* initialize save motion check box */
saveMotionBox = FALSE;
GetDItem(theDialog, SAVE_MOTION_CHECK, &itemType, &theItem, &box);
SetCtlValue(theItem, (int) saveMotionBox);
/* initialize new motion check box */
newMotionBox = FALSE;
GetDItem(theDialog, NEW_MOTION_CHECK, &itemType, &theItem, &box);
SetCtlValue(theItem, (int) newMotionBox);

/* hilite SOLVE button */
GetPort(&currentPort);
SetPort(theDialog);
GetDItem(theDialog, CONTINUE_SOLVE, &itemType, &theItem, &box);
PenSize(3, 3);
InsetRect(&box, -4, -4);
FrameRoundRect(&box, 16, 16);
SetPort(currentPort);

/* The user can select one of the following commands:
*
* Stop Button - Exits this function and closes the dialog box.
*               Useful when the motion and depth map have been
*               masterfully calculated.
*
* Solve Button - Use current motion parameters and find the least-

```

```

*           squares depth value at each data block.  The depth
*           map is then used to solve for the least-squares fit
*           motion.
*
* Use New Motion Parameters    -   Look at new motion parameters entered
*                               in editable text boxes.
*
* Use Constant Depth Check Box -   An option that can be checked
*                               before pressing "Solve".  Instead
*                               of calculating the new depth map,
*                               all depth values are set to a
*                               constant.
*
* Scale Depth Map Check Box    -   An option that can be checked before
*                               pressing "Solve".  Instead of
*                               calculating the new depth map, all
*                               depth values are scaled by a
*                               constant.
*/

```

```

ModalDialog(NIL, &itemHit); /* cycles until an enabled item is hit */
while (!(itemHit==STOP_SOLVE)) {

```

```

switch (itemHit) {
case CONTINUE_SOLVE: /* continue the iteration for the motion */

    /******
    /** If constant depth map check box is selected **
    /******

    if (constantDepthBox) {
    /* check constant depth value */
    GetDItem(theDialog,CONST_DEPTH_VALUE,&itemType,&theItem,&box);
    GetIText(theItem, theText);
    /* if constant depth value is a valid number */
    if (String_To_Double(theText, &temp)) {
        CONST_OK = TRUE; /* signal all is OK */
        constantDepth = temp; /* assign it to constantDepth */
    }
    else {
        CONST_OK = FALSE;
        Alert_User((UCP)"\pThe constant depth map value",(UCP)"\p",
            (UCP)"\pshould be a number!",(UCP)"\p");
        SelIText(theDialog,CONST_DEPTH_VALUE,0,32767); /* highlight */
    }
    /* if constant value is OK, set all depth values equal to it */
    if (CONST_OK) {
        Constant_Depth(constantDepth);
        GetDItem(theDialog,CONST_DEPTH_VALUE,&itemType,&theItem,&box);
        Double_To_String(constantDepth, theText, 1);
        SetIText(theItem, theText);
        constantDepthBox = FALSE;
    }
    } /* end if (constantDepthBox) */

```

```

/*****
/** If scale depth map box is selected **/
*****/

else if (scaleFactorBox) {
/* check scale factor value */
  GetDItem(theDialog, SCALE_DEPTH_VALUE, &itemType, &theItem, &box);
  GetIText(theItem, theText);
/* if depth map scaling factor is a valid number */
  if (String_To_Double(theText, &temp)) {
    SCALE_OK = TRUE; /* signal all is OK */
    scaleFactor = temp; /* assign it to scaleFactor */
  }
  else {
    SCALE_OK = FALSE;
    Alert_User((UCP) "\pThe depth map scale factor", (UCP) "\p",
              (UCP) "\pshould be a number!", (UCP) "\p");
    SelIText(theDialog, SCALE_DEPTH_VALUE, 0, 32767); /* highlight */
  }
/* if scale factor is OK, scale all depth values */
  if (SCALE_OK) {
    Scale_Depth(scaleFactor);
    GetDItem(theDialog, SCALE_DEPTH_VALUE, &itemType, &theItem, &box);
    Double_To_String(scaleFactor, theText, 1);
    SetIText(theItem, theText);
    scaleFactorBox = FALSE;
  }
} /* end if (scaleFactorBox) */

/*****
/** If new motion box is selected **/
*****/

else if (newMotionBox) {
/* check values entered in boxes */
  GetDItem(theDialog, TRANS_U, &itemType, &theItem, &box);
  GetIText(theItem, theText);
  if (String_To_Double(theText, &temp)) { /* if a # is present */
    U_OK = TRUE; /* signal all is OK */
    U = temp; /* assign it to U */
  }
  else {
    U_OK = FALSE;
    Alert_User((UCP) "\pThe U component of translation", (UCP) "\p",
              (UCP) "\pshould be a number!", (UCP) "\p");
    SelIText(theDialog, TRANS_U, 0, 32767); /* highlight U box */
  }
  GetDItem(theDialog, TRANS_V, &itemType, &theItem, &box);
  GetIText(theItem, theText);
  if (String_To_Double(theText, &temp)) { /* if a # is present */
    V_OK = TRUE; /* signal all is OK */
    V = temp; /* assign it to V */
  }
  else {
    V_OK = FALSE;
    Alert_User((UCP) "\pThe V component of translation", (UCP) "\p",
              (UCP) "\pshould be a number!", (UCP) "\p");
    SelIText(theDialog, TRANS_V, 0, 32767); /* highlight V box */
  }
}

```

```

}
GetDlgItem(theDialog, TRANS_W, &itemType, &theItem, &box);
GetWindowText(theItem, theText);
if (String_To_Double(theText, &temp)) { /* if a # is present */
    W_OK = TRUE; /* signal all is OK */
    W = temp; /* assign it to W */
}
else {
    W_OK = FALSE;
    Alert_User((UCP)"\pThe W component of translation", (UCP)"\p",
        (UCP)"\pshould be a number!", (UCP)"\p");
    SelIText(theDialog, TRANS_W, 0, 32767); /* highlight W box */
}
GetDlgItem(theDialog, ROT_A, &itemType, &theItem, &box);
GetWindowText(theItem, theText);
if (String_To_Double(theText, &temp)) { /* if a # is present */
    A_OK = TRUE; /* signal all is OK */
    A = temp; /* assign it to A */
}
else {
    A_OK = FALSE;
    Alert_User((UCP)"\pThe A component of rotation", (UCP)"\p",
        (UCP)"\pshould be a number!", (UCP)"\p");
    SelIText(theDialog, ROT_A, 0, 32767); /* highlight A box */
}
GetDlgItem(theDialog, ROT_B, &itemType, &theItem, &box);
GetWindowText(theItem, theText);
if (String_To_Double(theText, &temp)) { /* if a # is present */
    B_OK = TRUE; /* signal all is OK */
    B = temp; /* assign it to B */
}
else {
    B_OK = FALSE;
    Alert_User((UCP)"\pThe B component of rotation", (UCP)"\p",
        (UCP)"\pshould be a number!", (UCP)"\p");
    SelIText(theDialog, ROT_B, 0, 32767); /* highlight B box */
}
GetDlgItem(theDialog, ROT_C, &itemType, &theItem, &box);
GetWindowText(theItem, theText);
if (String_To_Double(theText, &temp)) { /* if a # is present */
    C_OK = TRUE; /* signal all is OK */
    C = temp; /* assign it to C */
}
else {
    C_OK = FALSE;
    Alert_User((UCP)"\pThe C component of rotation", (UCP)"\p",
        (UCP)"\pshould be a number!", (UCP)"\p");
    SelIText(theDialog, ROT_C, 0, 32767); /* highlight C box */
}
}

/* if all motion parameters are OK, find new depth map */
if (U_OK && V_OK && W_OK && A_OK && B_OK && C_OK)
    Solve_Depth_From_Motion();

/* reset check box */
newMotionBox = FALSE;
}

```

```

/*****
/** Otherwise, find motion from depth, then find new depth map */
*****/

else {
    Solve2_Motion_From_Depth(); /* use 4 DOF model */
    Solve_Depth_From_Motion();
}
break;
case CONST_DEPTH_CHECK:
    constantDepthBox = 1 - constantDepthBox;
    break;
case SCALE_DEPTH_CHECK:
    scaleFactorBox = 1 - scaleFactorBox;
    break;
case SAVE_MOTION_CHECK:
    saveMotionBox = 1 - saveMotionBox;
    break;
case NEW_MOTION_CHECK:
    newMotionBox = 1 - newMotionBox;
    break;
default:
    break;
}

/* place current translation and rotation in boxes */
GetDlgItem(theDialog, TRANS_U, &itemType, &theItem, &box);
Double_To_String(U, theText, 3);
SetDlgItemText(theItem, theText);
GetDlgItem(theDialog, TRANS_V, &itemType, &theItem, &box);
Double_To_String(V, theText, 3);
SetDlgItemText(theItem, theText);
GetDlgItem(theDialog, TRANS_W, &itemType, &theItem, &box);
Double_To_String(W, theText, 3);
SetDlgItemText(theItem, theText);
GetDlgItem(theDialog, ROT_A, &itemType, &theItem, &box);
Double_To_String(A, theText, 3);
SetDlgItemText(theItem, theText);
GetDlgItem(theDialog, ROT_B, &itemType, &theItem, &box);
Double_To_String(B, theText, 3);
SetDlgItemText(theItem, theText);
GetDlgItem(theDialog, ROT_C, &itemType, &theItem, &box);
Double_To_String(C, theText, 3);
SetDlgItemText(theItem, theText);

/* initialize constant depth check box */
GetDlgItem(theDialog, CONST_DEPTH_CHECK, &itemType, &theItem, &box);
SetCtlValue(theItem, (int) constantDepthBox);
/* initialize scale factor check box */
GetDlgItem(theDialog, SCALE_DEPTH_CHECK, &itemType, &theItem, &box);
SetCtlValue(theItem, (int) scaleFactorBox);
/* initialize save motion check box */
GetDlgItem(theDialog, SAVE_MOTION_CHECK, &itemType, &theItem, &box);
SetCtlValue(theItem, (int) saveMotionBox);
/* initialize new motion check box */
GetDlgItem(theDialog, NEW_MOTION_CHECK, &itemType, &theItem, &box);

```

```

        SetCtlValue(theItem, (int) newMotionBox);

/* hilite SOLVE button */
GetPort(&currentPort);
SetPort(theDialog);
GetDItem(theDialog, CONTINUE_SOLVE, &itemType, &theItem, &box);
PenSize(3, 3);
InsetRect(&box, -4, -4);
FrameRoundRect(&box, 16, 16);
SetPort(currentPort);

    ModalDialog(NIL, &itemHit); /* cycles until an enabled item is hit */
}

/* save depth values in flow vector records */
Retreive_Depth_Values();

/* save motion estimates in optical flow record */
HLock(theOpticalFlowHnd);
theOpticalFlowPtr = *theOpticalFlowHnd;
theOpticalFlowPtr->U = U;
theOpticalFlowPtr->V = V;
theOpticalFlowPtr->W = W;
theOpticalFlowPtr->A = A;
theOpticalFlowPtr->B = B;
theOpticalFlowPtr->C = C;
HUnlock(theOpticalFlowHnd);

Inval_Image(theMatchResult);

DisposDialog(theDialog);

}

```

The following function takes the current motion estimate and computes the depth value at each data block.

```

/* This function takes the motion and solves for the least-squares fit
 * depth value at each data block position.
 */
void Solve_Depth_From_Motion()
{
    int dataRow, dataCol;
    double SuT, SuTsq, SvT, SvTsq, numerator;
    double RuP, RvP, lfHalf, rtHalf, denominator;

    HLock(theDataBlockHnd);
    theDataBlockPtr = *theDataBlockHnd;

    for (dataRow=topLf.v; dataRow<=botRt.v; dataRow+=dataHeight)
        for (dataCol=topLf.h; dataCol<=botRt.h; dataCol+=dataWidth) {
            /* calculate numerator */
            SuT = theDataBlockPtr->x * W - U;
            SuTsq = SuT * SuT;
            SvT = theDataBlockPtr->y * W - V;
            SvTsq = SvT * SvT;

```

```

    numerator = SuTsq + SvTsq;
    /* calculate denominator */
    RuP=theDataBlockPtr->q11*A-theDataBlockPtr->q12*B
        +theDataBlockPtr->y*C;
    RvP=theDataBlockPtr->q21*A-theDataBlockPtr->q11*B
        -theDataBlockPtr->x*C;
    lfHalf = (theDataBlockPtr->u - RuP) * SuT;
    rtHalf = (theDataBlockPtr->v - RvP) * SvT;
    denominator = lfHalf + rtHalf;
    /* check if denominator is zero */
    if (denominator != 0.0)
        theDataBlockPtr->Z = numerator / denominator;
    else
        theDataBlockPtr->Z = INFINITE_DEPTH;
    if (theDataBlockPtr->Z <= 0.0)
        theDataBlockPtr->Z = INFINITE_DEPTH;
    theDataBlockPtr++;
}

HUnlock(theDataBlockHnd);
}

```

The following function computes the motion of the camera (Micro-Rover) from the current depth map. This function uses the 6 degrees of freedom model.

```

/* This function takes the depth values and solves for the least-squares
 * fit motion using the 6 DOF model.
 */
void Solve1_Motion_From_Depth()
{
    double parameter[7];          /* motion parameters, indices 1->6 */
    double rightSide[7];         /* right hand side, indices 1->6 */
    double matrix[7][7];        /* matrix, indices 1->6, 1->6 */
    double temp;
    double invZ, invZsq;        /* inverse depth and inverse depth squared */
    double weight;              /* weight on flow vector */
    double multiplier;          /* scales a row for Gaussian elimination */
    int i,j,k;
    int dataRow, dataCol;

    /*****
    /* zero out matrix and rightSide */
    *****/

    for (i=1; i<7; i++) {
        rightSide[i] = 0.0;      /* zero right hand side */
        for (j=1; j<7; j++)
            matrix[i][j] = 0.0; /* zero matrix element */
    }

    /*****
    /* create 6x6 array */
    *****/

```

```

HLock(theDataBlockHnd);
theDataBlockPtr = *theDataBlockHnd;

for (dataRow=topLf.v; dataRow<=botRt.v; dataRow+=dataHeight)
  for (dataCol=topLf.h; dataCol<=botRt.h; dataCol+=dataWidth) {

/* if we are not showing only good vectors, or if we are not using
 * any ratios, or if we are showing only good vectors and this
 * vector is within range ...
 */
  if ((!showGoodVectors)|| (theWeight == NO_RATIO)||
      ((theDataBlockPtr->W>=theMin)&&(theDataBlockPtr->W<=theMax))) {

    invZ = 1.0/theDataBlockPtr->Z;
    invZsq = invZ * invZ;
    weight = theDataBlockPtr->W;
/* add S sub-matrix */
    matrix[1][1] += invZsq;
    matrix[1][3] -= theDataBlockPtr->x * invZsq;
    matrix[2][2] += invZsq;
    matrix[2][3] -= theDataBlockPtr->y * invZsq;
    matrix[3][1] -= theDataBlockPtr->x * invZsq;
    matrix[3][2] -= theDataBlockPtr->y * invZsq;
    matrix[3][3] += theDataBlockPtr->s33 * invZsq;
/* add Q sub-matrix */
    matrix[1][4] -= theDataBlockPtr->q11 * invZ;
    matrix[1][5] += theDataBlockPtr->q12 * invZ;
    matrix[1][6] -= theDataBlockPtr->y * invZ;
    matrix[2][4] -= theDataBlockPtr->q21 * invZ;
    matrix[2][5] += theDataBlockPtr->q11 * invZ;
    matrix[2][6] += theDataBlockPtr->x * invZ;
    matrix[3][4] += theDataBlockPtr->q31 * invZ;
    matrix[3][5] -= theDataBlockPtr->q32 * invZ;
/* add QT sub-matrix */
    matrix[4][1] -= theDataBlockPtr->q11 * invZ;
    matrix[4][2] -= theDataBlockPtr->q21 * invZ;
    matrix[4][3] += theDataBlockPtr->q31 * invZ;
    matrix[5][1] += theDataBlockPtr->q12 * invZ;
    matrix[5][2] += theDataBlockPtr->q11 * invZ;
    matrix[5][3] -= theDataBlockPtr->q32 * invZ;
    matrix[6][1] -= theDataBlockPtr->y * invZ;
    matrix[6][2] += theDataBlockPtr->x * invZ;
/* add R sub-matrix */
    matrix[4][4] += theDataBlockPtr->r11;
    matrix[4][5] -= theDataBlockPtr->r12;
    matrix[4][6] -= theDataBlockPtr->x;
    matrix[5][4] -= theDataBlockPtr->r12;
    matrix[5][5] += theDataBlockPtr->r22;
    matrix[5][6] -= theDataBlockPtr->y;
    matrix[6][4] -= theDataBlockPtr->x;
    matrix[6][5] -= theDataBlockPtr->y;
    matrix[6][6] += theDataBlockPtr->s33;
/* add D vector */
    rightSide[1] -= theDataBlockPtr->u * invZ;
    rightSide[2] -= theDataBlockPtr->v * invZ;
    rightSide[3] += theDataBlockPtr->u * theDataBlockPtr->x * invZ;

```



```

        rightSide[3] += theDataBlockPtr->v * theDataBlockPtr->y * invZ;
/* add E vector */
        rightSide[4] += theDataBlockPtr->u * theDataBlockPtr->q11;
        rightSide[4] += theDataBlockPtr->v * theDataBlockPtr->q21;
        rightSide[5] -= theDataBlockPtr->u * theDataBlockPtr->q12;
        rightSide[5] -= theDataBlockPtr->v * theDataBlockPtr->q11;
        rightSide[6] += theDataBlockPtr->u * theDataBlockPtr->y;
        rightSide[6] -= theDataBlockPtr->v * theDataBlockPtr->x;
    }
    theDataBlockPtr++;
}

/*****/
/* set very small numbers equal to zero */
/*****/

for (i=1; i<7; i++) {
    ROFF(&rightSide[i]);    /* zero right hand side */
    for (j=1; j<7; j++)
        ROFF(&matrix[i][j]);    /* zero matrix element */
}

/*****/
/* perform Gaussian elimination */
/*****/

for (i=1; i<6; i++)    /* loop through first 5 rows in matrix */
    for (j=i+1; j<7; j++) {    /* loop through all rows below current */
        /* scaling factor */
        multiplier = matrix[j][i]/matrix[i][i];
        /* loop through the 6 elements in the row */
        for (k=i; k<7; k++)
            /* subtract row element */
            matrix[j][k]-=multiplier*matrix[i][k];
        /* subtract scaled right side */
        rightSide[j]-=multiplier*rightSide[i];
    }

/*****/
/* set very small numbers equal to zero */
/*****/

for (i=1; i<7; i++) {
    ROFF(&rightSide[i]);    /* zero right hand side */
    for (j=1; j<7; j++)
        ROFF(&matrix[i][j]);    /* zero matrix element */
}

/*****/
/* pick off answers */
/*****/

```

```

for (i=6; i>0; i--) {
    temp = rightSide[i];
    /* step through all parameters below current parameter */
    for (j=6;j>i;j--)
        /* subtract from right hand side */
        temp -= matrix[i][j]*parameter[j];

    if (matrix[i][i] != 0.0) /* check if divide by zero */
        parameter[i] = temp / matrix[i][i];
    else {
        Alert_User((UCP)"\pError solving for motion", (UCP)"\p",
            (UCP)"\pDivide by zero!!!", (UCP)"\p");
        return;
    }
}

/*****
/* assign parameters to motion variables */
*****/

U = ROE(parameter[1]);
V = ROE(parameter[2]);
W = ROE(parameter[3]);
A = ROE(parameter[4]);
B = ROE(parameter[5]);
C = ROE(parameter[6]);
}

```

The following function computes the motion of the camera (Micro-Rover) from the current depth map. This function uses the 4 degrees of freedom model.

```

/* This function takes the depth values and solves for the least-squares
 * fit motion using the 4 DOF model.
 */
void Solve2_Motion_From_Depth()
{
    double parameter[7]; /* motion parameters, indices 1->6 */
    double rightSide[7]; /* right hand side, indices 1->6 */
    double matrix[7][7]; /* matrix, indices 1->6, 1->6 */
    double temp;
    double invZ, invZsq; /* inverse depth and inverse depth squared */
    double weight; /* weight on flow vector */
    double multiplier; /* scales a row for Gaussian elimination */
    int i,j,k;
    int dataRow, dataCol;

    /*****
    /* zero out matrix and rightSide */
    *****/
}

```

```

for (i=1; i<7; i++) {
    rightSide[i] = 0.0;      /* zero right hand side */
    for (j=1; j<7; j++)
        matrix[i][j] = 0.0; /* zero matrix element */
}

/*****
/* create 6x6 array */
*****/

HLock(theDataBlockHnd);
theDataBlockPtr = *theDataBlockHnd;

for (dataRow=topLf.v; dataRow<=botRt.v; dataRow+=dataHeight)
    for (dataCol=topLf.h; dataCol<=botRt.h; dataCol+=dataWidth) {
        if ((!showGoodVectors)|| (theWeight == NO_RATIO)||
            ((theDataBlockPtr->W>=theMin)&&(theDataBlockPtr->W<=theMax))) {
            invZ = 1.0/theDataBlockPtr->Z;
            invZsq = invZ * invZ;
            weight = theDataBlockPtr->W;
            /* add S sub-matrix */
            matrix[1][1] += invZsq;
            matrix[1][3] -= theDataBlockPtr->x * invZsq;
            matrix[2][2] += invZsq;
            matrix[2][3] -= theDataBlockPtr->y * invZsq;
            matrix[3][1] -= theDataBlockPtr->x * invZsq;
            matrix[3][2] -= theDataBlockPtr->y * invZsq;
            matrix[3][3] += theDataBlockPtr->s33 * invZsq;
            /* add Q sub-matrix */
            matrix[1][4] -= theDataBlockPtr->q11 * invZ;
            matrix[1][5] += theDataBlockPtr->q12 * invZ;
            matrix[1][6] -= theDataBlockPtr->y * invZ;
            matrix[2][4] -= theDataBlockPtr->q21 * invZ;
            matrix[2][5] += theDataBlockPtr->q11 * invZ;
            matrix[2][6] += theDataBlockPtr->x * invZ;
            matrix[3][4] += theDataBlockPtr->q31 * invZ;
            matrix[3][5] -= theDataBlockPtr->q32 * invZ;
            /* add QT sub-matrix */
            matrix[4][1] -= theDataBlockPtr->q11 * invZ;
            matrix[4][2] -= theDataBlockPtr->q21 * invZ;
            matrix[4][3] += theDataBlockPtr->q31 * invZ;
            matrix[5][1] += theDataBlockPtr->q12 * invZ;
            matrix[5][2] += theDataBlockPtr->q11 * invZ;
            matrix[5][3] -= theDataBlockPtr->q32 * invZ;
            matrix[6][1] -= theDataBlockPtr->y * invZ;
            matrix[6][2] += theDataBlockPtr->x * invZ;
            /* add R sub-matrix */
            matrix[4][4] += theDataBlockPtr->r11;
            matrix[4][5] -= theDataBlockPtr->r12;
            matrix[4][6] -= theDataBlockPtr->x;
            matrix[5][4] -= theDataBlockPtr->r12;
            matrix[5][5] += theDataBlockPtr->r22;
            matrix[5][6] -= theDataBlockPtr->y;
            matrix[6][4] -= theDataBlockPtr->x;
            matrix[6][5] -= theDataBlockPtr->y;

```

```

        matrix[6][6] += theDataBlockPtr->s33;
/* add D vector */
        rightSide[1] -= theDataBlockPtr->u * invZ;
        rightSide[2] -= theDataBlockPtr->v * invZ;
        rightSide[3] += theDataBlockPtr->u * theDataBlockPtr->x * invZ;
        rightSide[3] += theDataBlockPtr->v * theDataBlockPtr->y * invZ;
/* add E vector */
        rightSide[4] += theDataBlockPtr->u * theDataBlockPtr->q11;
        rightSide[4] += theDataBlockPtr->v * theDataBlockPtr->q21;
        rightSide[5] -= theDataBlockPtr->u * theDataBlockPtr->q12;
        rightSide[5] -= theDataBlockPtr->v * theDataBlockPtr->q11;
        rightSide[6] += theDataBlockPtr->u * theDataBlockPtr->y;
        rightSide[6] -= theDataBlockPtr->v * theDataBlockPtr->x;
    }
    theDataBlockPtr++;
}

/*****
/* set very small numbers equal to zero */
*****/

for (i=1; i<7; i++) {
    ROFF(&rightSide[i]);    /* zero right hand side */
    for (j=1; j<7; j++)
        ROFF(&matrix[i][j]);    /* zero matrix element */
}

/*****
/* perform Gaussian elimination */
*****/

for (i=3; i<6; i++)        /* loop through last 3 rows in matrix */
    for (j=i+1; j<7; j++) {    /* loop through all rows below current */
        /* scaling factor */
        multiplier = matrix[j][i]/matrix[i][i];
        /* loop through the 6 elements in the row */
        for (k=i; k<7; k++)
            /* subtract row element */
            matrix[j][k] -= multiplier*matrix[i][k];
        /* subtract scaled right side */
        rightSide[j] -= multiplier*rightSide[i];
    }

/*****
/* set very small numbers equal to zero */
*****/

for (i=1; i<7; i++) {
    ROFF(&rightSide[i]);    /* zero right hand side */
    for (j=1; j<7; j++)
        ROFF(&matrix[i][j]);    /* zero matrix element */
}

```

```

/*****/
/* pick off answers */
/*****/

for (i=6; i>2; i--) {          /* step up from last parameter */
    temp = rightSide[i];      /* get right hand side value */
/* step through all parameters below current parameter */
    for (j=6;j>i;j--)
/* subtract from right hand side */
        temp -= matrix[i][j]*parameter[j];

    if (matrix[i][i] != 0.0) /* check if divide by zero */
        parameter[i] = temp / matrix[i][i];
    else {
        Alert_User((UCP)"\pError solving for motion", (UCP)"\p",
                    (UCP)"\pDivide by zero!!!", (UCP)"\p");
        return;
    }
}

/*****/
/* assign parameters to motion variables */
/*****/

U = 0.0;
V = 0.0;
W = ROE(parameter[3]);
A = ROE(parameter[4]);
B = ROE(parameter[5]);
C = ROE(parameter[6]);
}

```

The following function scales the depth map.

```

/* This function scales the depth value at each data block position by
 * the constant scale factor.
 */
void Scale_Depth(scaleFactor)
double scaleFactor;
{
    int dataRow, dataCol;

    HLock(theDataBlockHnd);
    theDataBlockPtr = *theDataBlockHnd;

    for (dataRow=topLf.v; dataRow<=botRt.v; dataRow+=dataHeight)
        for (dataCol=topLf.h; dataCol<=botRt.h; dataCol+=dataWidth)
            theDataBlockPtr++->Z *= scaleFactor;

    HUnlock(theDataBlockHnd);
}

```

The following function sets the depth map to a constant value.

```
/* This function sets the depth value at each data block position equal
 * to the constant depth value.
 */
void Constant_Depth(constantDepth)
    double constantDepth;
{
    int dataRow, dataCol;

    HLock(theDataBlockHnd);
    theDataBlockPtr = *theDataBlockHnd;

    for (dataRow=topLf.v; dataRow<=botRt.v; dataRow+=dataHeight)
        for (dataCol=topLf.h; dataCol<=botRt.h; dataCol+=dataWidth)
            theDataBlockPtr++->Z = constantDepth;

    HUnlock(theDataBlockHnd);
}
```

The following function initializes the data block data.

```
/* This function allocates memory for and initializes the records of the
 * data block data. The data consists of various matrix elements that
 * are used to solve for the motion. These matrix elements need only be
 * computed once since they are dependent upon the data block position.
 * The DB data also contains the appropriate weight selected by the user
 * (i.e. the current weight that is displayed in the window). Finally,
 * the depth value, which is the only real variable for each DB is
 * stored.
 */
int Init_Solve_Array(void)
{
    int dataRow, dataCol; /* counter variables */
    double zeroError,bestError,aveError;
    double invVar, offVar;
    double aveOffVar, sumCosine, prodCosine;
    double depth;

    Point offsetPt, aveOffset; /* flow vector and average flow vector */

    double x, y; /* normalized center of data block */

    OpticalFlowPtr theOpticalFlowPtr;

    FlowVectorPtr currentDataBlockPtr;

    /* front window is either multiple result or overlay result */
    theMatchResult = Find_Image(FrontWindow());
    /* if front window is not an image, i.e. weird error, then exit */
    if (!theMatchResult) return(0);

    HLock(theMatchResult);
```

```

/* get handle to optical flow record */
theOpticalFlowHnd = (*theMatchResult)->flowData;
/* exit if there is no handle */
if (!theOpticalFlowHnd) return (0);

HLock(theOpticalFlowHnd);
theOpticalFlowPtr = *theOpticalFlowHnd;

/* get handle to first flow vector record */
currentDataBlockHnd = theOpticalFlowPtr->firstFlowVector;
HLock(currentDataBlockHnd);
currentDataBlockPtr = *currentDataBlockHnd;

/* find match parameters */
topLf.v = theOpticalFlowPtr->topLeft.v;
topLf.h = theOpticalFlowPtr->topLeft.h;
botRt.v = theOpticalFlowPtr->botRight.v;
botRt.h = theOpticalFlowPtr->botRight.h;
dataHeight = theOpticalFlowPtr->dataHeight;
dataWidth = theOpticalFlowPtr->dataWidth;
showGoodVectors = theOpticalFlowPtr->showGoodVectors;
theMin = theOpticalFlowPtr->minWeight;
theMax = theOpticalFlowPtr->maxWeight;
theWeight = theOpticalFlowPtr->theWeight;

/* set depth map flag so that depth values can be displayed */
theOpticalFlowPtr->depthMapUsed = TRUE;

/* find number of data blocks both horizontally and vertically */
numDBs.v = (botRt.v - topLf.v) / dataHeight + 1;
numDBs.h = (botRt.h - topLf.h) / dataWidth + 1;

/* if less than 6 data blocks, there aren't enough constraints */
if (numDBs.v*numDBs.h < 6) {
    Alert_User((UCP)"\pNot enough data blocks to solve for", (UCP)"\p",
              (UCP)"\pcamera motion. Min 6 DB's required!", (UCP)"\p");
    return(0);
}

/* dynamically allocate memory for data block records */
DBRsize = (long) sizeof(DataBlockRecord);
theDataBlockHnd = (DataBlockHnd)
    NewHandle(DBRsize*((long)numDBs.v)*((long)numDBs.h));
if (!theDataBlockHnd) {
    Alert_User((UCP)"\pMemory could not be allocated", (UCP)"\p",
              (UCP)"\pfor data block data!", (UCP)"\p");
    return(0);
}

/* width of pixels in meters */
PELwidth = CCD_ARRAY_WIDTH / PIXELS;
/* width normalized to focal length */
PELfactor = PELwidth / FOCAL_LENGTH;

HLock(theDataBlockHnd);
theDataBlockPtr = *theDataBlockHnd;

```

```

for (dataRow=topLf.v; dataRow<=botRt.v; dataRow+=dataHeight)
  for (dataCol=topLf.h; dataCol<=botRt.h; dataCol+=dataWidth) {

/* get flow vector and weights from data handle */
  offsetPt.v = currentDataBlockPtr->theOffset.v;
  offsetPt.h = currentDataBlockPtr->theOffset.h;
  aveOffset.v = currentDataBlockPtr->aveOffset.v;
  aveOffset.h = currentDataBlockPtr->aveOffset.h;
  bestError = (double) currentDataBlockPtr->bestError;
  aveError = (double) currentDataBlockPtr->aveError;
  zeroError = (double) currentDataBlockPtr->zeroError;
  invVar = (double) currentDataBlockPtr->invVar;
  offVar = (double) currentDataBlockPtr->offVar;
  aveOffVar = (double) currentDataBlockPtr->aveOffVar;
  sumCosine = (double) currentDataBlockPtr->sumCosine;
  prodCosine = (double) currentDataBlockPtr->prodCosine;
  depth = currentDataBlockPtr->depth;

/* normalize DB position and flow vector */
  x = theDataBlockPtr->x = (((double) (dataCol+dataWidth/2))
                          -PRIN_POINT_X)*PELfactor;
  y = theDataBlockPtr->y = (((double) (dataRow+dataHeight/2))
                          -PRIN_POINT_Y)*PELfactor;
  theDataBlockPtr->u = - offsetPt.h * PELfactor;
  theDataBlockPtr->v = - offsetPt.v * PELfactor;
/* take appropriate weight */
  switch (theWeight) {
    case NO_RATIO:
      theDataBlockPtr->W = 1.0;
      break;
    case ZERO_ERROR_TO_BEST_ERROR:
      theDataBlockPtr->W = zeroError/bestError;
      break;
    case AVERAGE_ERROR_TO_BEST_ERROR:
      theDataBlockPtr->W = aveError/bestError;
      break;
    case INVERSE_VARIANCE:
      theDataBlockPtr->W = invVar;
      break;
    case OFFSET_TO_OFFSET_PLUS_VARIANCE:
      theDataBlockPtr->W = offVar;
      break;
    case AVEOFFSET_TO_AVEOFFSET_PLUS_VARIANCE:
      theDataBlockPtr->W = aveOffVar;
      break;
    case AVE_COSINE:
      theDataBlockPtr->W = sumCosine;
      break;
    case PROD_COSINE:
      theDataBlockPtr->W = prodCosine;
      break;
    default:
      theDataBlockPtr->W = 1.0;
      break;
  }
/* initialize matrix elements */
  theDataBlockPtr->s33 = x*x + y*y;
  theDataBlockPtr->q11 = x*y;

```



```

    theDataBlockPtr->q12 = x*x + 1.0;
    theDataBlockPtr->q21 = y*y + 1.0;
    theDataBlockPtr->q31 = y*y*y + (x*x + 1.0)*y;
    theDataBlockPtr->q32 = x*x*x + (y*y + 1.0)*x;
    theDataBlockPtr->r11 = y*y*y*y + 2.0*y*y + x*x*y*y + 1.0;
    theDataBlockPtr->r12 = x*x*x*y + 2.0*x*y + x*y*y*y;
    theDataBlockPtr->r22 = x*x*x*x + 2.0*x*x + x*x*y*y + 1.0;

    /* initialize depth */
    theDataBlockPtr->Z = depth;

    currentDataBlockPtr++; /* increment to next flow vector record */
    theDataBlockPtr++;    /* increment to next DB data record */
}
HUnlock(theDataBlockHnd);
HUnlock(theOpticalFlowHnd);
HUnlock(currentDataBlockHnd);
HUnlock(theMatchResult);

return(1);
}

```

The following function stores the depth values in the flow vector records.

```

/* This function retrieves the depth values from the data block records
 * and stores the depth values in the flow vector records so that the
 * depth map can be displayed either by itself or overlaid on the
 * underlying image.
 */
void Retrieve_Depth_Values(void)
{
    int dataRow, dataCol;    /* counter variables */

    FlowVectorPtr currentDataBlockPtr;

    HLock(currentDataBlockHnd);
    currentDataBlockPtr = *currentDataBlockHnd;

    HLock(theDataBlockHnd);
    theDataBlockPtr = *theDataBlockHnd;

    for (dataRow=topLf.v; dataRow<=botRt.v; dataRow+=dataHeight)
        for (dataCol=topLf.h; dataCol<=botRt.h; dataCol+=dataWidth) {
            /* store depth value */
            currentDataBlockPtr->depth = theDataBlockPtr->Z;
            /* increment to next flow vector record */
            currentDataBlockPtr++;
            /* increment to next DB data record */
            theDataBlockPtr++;
        }

    HUnlock(theDataBlockHnd);
    HUnlock(currentDataBlockHnd);
}

```

The following functions set the variable equal to zero if it is less than 1e-10.

```
double ROE(theDouble)
  double theDouble;
{
  double theAbsolute;

  if (theDouble > 0.0)
    theAbsolute = theDouble;
  else
    theAbsolute = -theDouble;

  if (theAbsolute < 1.0e-10)
    return(0.0);
  else
    return(theDouble);
}

void ROFF(theDouble)
  double *theDouble;
{
  *theDouble = ROE(*theDouble);
}
```

Appendix C The Code That Generates Obstacle Maps

The following code generates obstacle maps from depth maps. Each depth value from the depth map of the front window is added to an intermediate obstacle map. An uncertainty region represents the location of a scene point. The intermediate obstacle map is then fused with the obstacle map to enhance the locations of any obstacles and attenuate the locations of noise.

```
/*
*****
***** #define's
*****
*/

#define MAP_WIDTH 6.0 /* in meters */
#define MAP_DEPTH 5.0 /* in meters */
#define MAP_RESOLUTION 0.05 /* in meters */

/* the uncertainty in depth is constrained to 1/2 meter */
#define MAX_UNCERTAINTY ((int) (0.5 / MAP_RESOLUTION))

#define FILTER_THRESHOLD 80 /* threshold to decide obstacle */

Handle obstaclePreMap; /* intermediate obstacle map */

int numIncrmtsWide, numIncrmtsDeep; /* width and height of map */
```

The following function opens a window for the obstacle map and allocates memory for the intermediate obstacle map.

```
/* This function initializes or reinitializes the obstacle map image.
 * The pixel map is sized or resized according to the desired map size
 * and resolution with each pixel representing an incremental area.
 */
int Init_Obstacle_Map(theObstacleImage)
ImageHandle *theObstacleImage;
{
    double mapWidth, mapDepth, mapResolution;

    if (*theObstacleImage)
        Trash_Image(*theObstacleImage);

    mapWidth = MAP_WIDTH;
    mapDepth = MAP_DEPTH;
    mapResolution = MAP_RESOLUTION;

    numIncrmtsWide = (int) (mapWidth / mapResolution);
    numIncrmtsDeep = (int) (mapDepth / mapResolution);

    if (!Open_Window(theObstacleImage, OBSTACLE_WINDOW, FALSE,
```

```

        numIncrmtsDeep, numIncrmtsWide, 3))
return(0);

obstaclePreMap = NewHandle(((long) numIncrmtsWide)
                            * ((long) numIncrmtsDeep) + 1000L);
if (!obstaclePreMap) {
    Trash_Image(*theObstacleImage);
    return(0);
}
}

```

The following function uses the depth map of the front window to update the obstacle map with additional uncertainty regions.

```

void Update_Obstacle_Map(theObstacleImage, theOpticalFlow,
                        IU, IV, IW, IA, IB, IC)
    ImageHandle theObstacleImage;
    OpticalFlowHandle theOpticalFlow;
/* current position and orientation of Micro-Rover */
double IU, IV, IW, IA, IB, IC;
{
    ImagePtr theObstacleImagePtr; /* pointer to obstacle image record */
    OpticalFlowPtr theOpticalFlowPtr; /* pointer to optical flow record */
    FlowVectorHandle theDataBlockHandle; /* handle to flow vector record */
    FlowVectorPtr theDataBlockPtr; /* pointer to flow vector record */
    PixMapHandle thePixMap;
    Point topLf, botRt; /* top left and bottom right DB's */
    int dataHeight, dataWidth, dataRow, dataCol;
    double x, y, xComp, yComp, len, depth;
    double PELwidth, PELfactor; /* camera characteristics */
    CharPtr thePixel, theIncrmtObstacle;
    int i, j;
    int grayScaleCount; /* gray scale incremental unit */
    Boolean viewDepthMap, showGoodVectors, useDepthFilter, drawVectorFlag;
    int theWeight, bestError, aveError, zeroError;
    int invVar, offVar, aveOffVar;
    int ratioInt, sumCosine, prodCosine;
    double theMin, theMax, theValue;
    int rowBytes;

    HLock(theOpticalFlow);
    theOpticalFlowPtr = *theOpticalFlow;

/* get top left and bottom right data block positions */
    topLf.h = theOpticalFlowPtr->topLeft.h;
    topLf.v = theOpticalFlowPtr->topLeft.v;
    botRt.h = theOpticalFlowPtr->botRight.h;
    botRt.v = theOpticalFlowPtr->botRight.v;
    dataHeight = theOpticalFlowPtr->dataHeight;
    dataWidth = theOpticalFlowPtr->dataWidth;

    viewDepthMap = theOpticalFlowPtr->viewDepthMap;
    showGoodVectors = theOpticalFlowPtr->showGoodVectors;
    useDepthFilter = theOpticalFlowPtr->useDepthFilter;
    if (useDepthFilter) {

```

```

    theMin = theOpticalFlowPtr->minDepth;
    theMax = theOpticalFlowPtr->maxDepth;
}
else {
    theMin = theOpticalFlowPtr->minWeight;
    theMax = theOpticalFlowPtr->maxWeight;
}
theWeight = theOpticalFlowPtr->theWeight;

PELwidth = CCD_ARRAY_WIDTH/PIXELS;    /* width of pixels in meters */
PELfactor = PELwidth/FOCAL_LENGTH;    /* width norm'd to focal length */

/* get handle to first flow vector */
theDataBlockHandle = theOpticalFlowPtr->firstFlowVector;
HLock(theDataBlockHandle);
/* get pointer to first flow vector */
theDataBlockPtr = *theDataBlockHandle;

grayScaleCount = 128 / 16;

/***** Clear (zero-out) obstacle Pre-Map *****/
/***** Create obstacle pre-map from data block depth values *****/

HLock(obstaclePreMap);
theIncrmtObstacle = *obstaclePreMap;

for (i=0; i<numIncrmtsDeep; i++)
    for (j=0; j<numIncrmtsWide; j++)
        *theIncrmtObstacle++ = 0;

/***** Create obstacle pre-map from data block depth values *****/

for (dataRow=topLf.v; dataRow<=botRt.v; dataRow+=dataHeight)
    for (dataCol=topLf.h; dataCol<=botRt.h; dataCol+=dataWidth) {
        xComp = (double) theDataBlockPtr->theOffset.h;
        yComp = (double) theDataBlockPtr->theOffset.v;
        bestError = theDataBlockPtr->bestError;
        aveError = theDataBlockPtr->aveError;
        zeroError = theDataBlockPtr->zeroError;
        invVar = theDataBlockPtr->invVar;
        offVar = theDataBlockPtr->offVar;
        aveOffVar = theDataBlockPtr->aveOffVar;
        sumCosine = theDataBlockPtr->sumCosine;
        prodCosine = theDataBlockPtr->prodCosine;
        depth = theDataBlockPtr->depth;

        switch (theWeight) {
            case NO_RATIO:
                ratioInt = 1;
                break;

```

```

    case ZERO_ERROR_TO_BEST_ERROR:
        ratioInt = (int) zeroError/bestError;
        break;
    case AVERAGE_ERROR_TO_BEST_ERROR:
        ratioInt = (int) aveError/bestError;
        break;
    case INVERSE_VARIANCE:
        ratioInt = invVar;
        break;
    case OFFSET_TO_OFFSET_PLUS_VARIANCE:
        ratioInt = offVar;
        break;
    case AVEOFFSET_TO_AVEOFFSET_PLUS_VARIANCE:
        ratioInt = aveOffVar;
        break;
    case AVE_COSINE:
        ratioInt = sumCosine;
        break;
    case PROD_COSINE:
        ratioInt = prodCosine;
        break;
}
theValue = (double) ratioInt;

if (useDepthFilter)
    drawVectorFlag = ((depth >= theMin)&&(depth <= theMax));
else
    drawVectorFlag = ((theValue >= theMin)&&(theValue <= theMax));

/* If we want to show all flow vectors, or if the value is between
 * min and max... */
if ((!showGoodVectors)|| (drawVectorFlag)) {
    /* "x" and "y" are the actual camera coordinates in meters
     * divided by the focal length in meters, i.e. "x" and "y"
     * are tangents of angles in the xz and yz planes. */
    x = (((double) (dataCol+dataWidth/2))-PRIN_POINT_X)*PELfactor;
    y = (((double) (dataRow+dataHeight/2))-PRIN_POINT_Y)*PELfactor;
    len = sqrt(xComp*xComp + yComp*yComp);
    /* add uncertainty region to intermediate obstacle map */
    Add_Obstacle(x, y, len, depth, IU, IV, IW, IA, IB, IC);
}
theDataBlockPtr++;
}

/*****
/***** Translate obstacle pre-map to image pixel map *****/
/*****/

HLock(theObstacleImage);
theObstacleImagePtr = *theObstacleImage;
thePixMap = theObstacleImagePtr->imPixMap;
rowBytes = (*thePixMap)->rowBytes;
BitClr(&rowBytes, 0L); /* clear high bit for pixel map */
theIncrmtObstacle = *obstaclePreMap;

/* cycle through each row of the obstacle pixel map */

```

```

for (i=0; i<numIncrmtsDeep; i++) {
/* find pointer to first pixel of line */
  thePixel = (CharPtr) ((*thePixMap)->baseAddr
                        + ((long)i)*((long)rowBytes));
/* cycle through each column of the pixel map */
  for (j=0; j<numIncrmtsWide; j++) {
/* if the count of the increment is zero */
    if (*theIncrmtObstacle == 0)
/* but the obstacle map shows an obstacle */
      if (*thePixel > 0)
/* lighten the shade of gray displayed at that pixel */
        *thePixel -= grayScaleCount;
      else /* already white */
        ;
/* if additional counts would go past black */
    else if (*theIncrmtObstacle * grayScaleCount > (127 - *thePixel))
/* set equal to black */
      *thePixel = 127;
/* otherwise, add additional counts to present count */
    else
      *thePixel += *theIncrmtObstacle * grayScaleCount;
/* go to next pixel in pixel map */
    thePixel++;
/* go to next increment in obstacle map */
    theIncrmtObstacle++;
  }
}

HUnlock(obstaclePreMap);
HUnlock(theObstacleImage);
HUnlock(theDataBlockHandle);
HUnlock(theOpticalFlow);

Inval_Image(theObstacleImage);
}

```

The following function adds an uncertainty region to the intermediate obstacle map based upon the x and y image coordinates, the depth, and the length of the flow vector.

```

void Add_Obstacle(x, y, len, depth, IU, IV, IW, IA, IB, IC)
  double x, y, len, depth;
  double IU, IV, IW, IA, IB, IC;
{
  double WX, WY; /* world coordinates */
  double thePoint_X, thePoint_Z;
  int theIncrmt_X, theIncrmt_Z, tempIncrmt_Z;
  int minIncrmt_X, maxIncrmt_X;
  int rowIncrmt, colIncrmt;
  double heightDiff;
  Boolean skipMatch;
  CharPtr theObstaclePreMapPtr, temp;
  int i, j, depthUncertainty;
  double tanAngle, tanMinAngle, tanMaxAngle;
  double xDist, minDist, maxDist;

```

```

WX = x * depth;
WY = y * depth;

/*****
/***** Place uncertainty region around match point *****/
/*****

/* find depth uncertainty in terms of incremental areas */
depthUncertainty = (int) (depth / len / MAP_RESOLUTION);
if (depthUncertainty > MAX_UNCERTAINTY)
    depthUncertainty = MAX_UNCERTAINTY;

/* find angle of ray to scene point, as well as min and max angles */
tanAngle = x; /* x normalized to p.d. */
xDist = FOCAL_LENGTH * tanAngle; /* x in meters */
maxDist = xDist + 4.0 * PIXEL_WIDTH; /* add 4 pixel uncertainty */
minDist = xDist - 4.0 * PIXEL_WIDTH;
tanMaxAngle = maxDist / FOCAL_LENGTH; /* find tangent of max angle */
tanMinAngle = minDist / FOCAL_LENGTH; /* find tangent of min angle */

/* if the match is not in the floor (ground) plane... */
if (!skipMatch) {
    theObstaclePreMapPtr = *obstaclePreMap;
    /* find match point on the obstacle map */
    thePoint_X = WX + IU; /* add Micro-Rover's offset "IU" */
    thePoint_Z = depth + IW; /* add Micro-Rover's offset "IW" */
    /* find match point in terms of incremental grid points */
    theIncrmt_X = (int) (thePoint_X / MAP_RESOLUTION);
    theIncrmt_Z = (int) (thePoint_Z / MAP_RESOLUTION);

    /* from the min depth to the maximum depth... */
    for (i=-depthUncertainty; i<=depthUncertainty; i++) {
        /* compute current depth */
        tempIncrmt_Z = theIncrmt_Z + i;
        /* find left and right edge increments */
        minIncrmt_X = (int) (tanMinAngle * ((double)tempIncrmt_Z));
        maxIncrmt_X = (int) (tanMaxAngle * ((double)tempIncrmt_Z));
        /* mark all increments between the left and right edges */
        for (j = minIncrmt_X; j <= maxIncrmt_X; j++) {
            temp = (CharPtr) theObstaclePreMapPtr;
            rowIncrmt = numIncrmtsDeep - tempIncrmt_Z;
            colIncrmt = numIncrmtsWide/2 + j;
            if ((rowIncrmt >= 0)&&(rowIncrmt < numIncrmtsDeep)) {
                temp += (long) (rowIncrmt * numIncrmtsWide);
                if ((colIncrmt >= 0)&&(colIncrmt < numIncrmtsWide)) {
                    temp += (long) colIncrmt;
                    *temp += 1; /* increment pre-map */
                }
            }
        }
    } /* end for through increments in a row */
} /* end for through depth uncertainty range */

}
}

```


The following function performs a binary test on the count at each increment in the obstacle map. If the count is greater than "FILTER_THRESHOLD", the increment is marked with black, otherwise it's marked with white.

```
void Filter_Obstacle()
{
    int height, width;
    ImagePtr theObstacleImagePtr;
    PixMapHandle thePixMap;
    int rowBytes;
    CharPtr thePixelPtr;
    int i, j;

    Get_Image_Size(obstacleImage, &height, &width);

    HLock(obstacleImage);
    theObstacleImagePtr = *obstacleImage;
    thePixMap = theObstacleImagePtr->imPixMap;
    rowBytes = (*thePixMap)->rowBytes;
    BitClr(&rowBytes, 0L); /* clear high bit for pixel map */

    for (i=0; i<height; i++){ /* for every row in the pixel map */
        /* find pointer to first pixel in row */
        thePixelPtr = (CharPtr) ((*thePixMap)
                                ->baseAddr+((long)i)*((long)rowBytes));
        for (j=0; j<width; j++){ /* for each pixel in the pixel map */
            if (*thePixelPtr >= FILTER_THRESHOLD)
                *thePixelPtr = 127;
            else
                *thePixelPtr = 0;
            thePixelPtr++;
        }
    }

    HUnlock(obstacleImage);
    Inval_Image(obstacleImage);
}
```


Appendix D Maximum Rotation Between Images

This appendix discusses how the maximum length of the flow vector places a constraint on the maximum rotation between images. When the Micro-Rover is turning, this maximum rotation sets an upper bound on the time between images.

When the camera rotates, the light from a scene point irradiates a different point in the image plane. Consider the model in Figure D.1, where a 2-dimensional model of a camera is used to simplify the analysis. Initially, the point in the image plane which the light from a scene point irradiates was a distance x from the principal point. After the camera rotates, the new, irradiated point is a distance x' from the principal point.

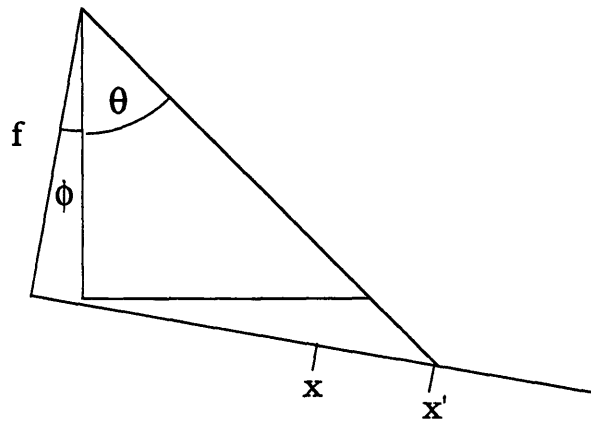


Figure D.1 Rotation of the Camera Causes the Irradiated Point to Move

The original angle, θ , formed by the light ray passing through the center of projection and irradiating the image plane is related to the point on the image plane, x , and the principal distance, f , by:

$$\theta = \tan^{-1}\left(\frac{x}{f}\right)$$

After the camera has been rotated, the new angle, $\theta + \phi$, is related to the new point on the image plane, x' , and the principal distance by:

$$\theta + \phi = \tan^{-1}\left(\frac{x'}{f}\right)$$

The new point on the image plane is the sum of the original point, x , plus some change, Δx . This change is the length of the flow vector, which must stay within the search window that we are using. The change, since it is not normalized but is in the true metric coordinates, can be factored into the width of a pixel element, w_{pel} , and some number of pixel elements, N_{pel} :

$$x' = x + \Delta x = x + N_{\text{pel}}w_{\text{pel}}$$

The maximum rotation allowable between successive images can be determined when the principal distance, width of the CCD array, width of each pixel element, and size of the search window have been determined. The equation for the maximum rotation is:

$$\phi = \tan^{-1}\left(\frac{x}{f}\right) - \tan^{-1}\left(\frac{x - N_{\text{pel}}w_{\text{pel}}}{f}\right)$$

The camera characteristics of the Pulnix camera that was used to test the matching algorithm were presented in Chapter 4. The dimensions of the CCD array were estimated to be 5 mm x 5 mm. The focal length of the lens was assumed to be 8.5 mm. There were 256 pixels horizontally in the images we tested. The pixel width, w_{pel} , is 5 mm divided by 256, or 19.5 μm . The maximum length of a flow vector is 8 pixels.

Using the equation above, the maximum rotation is:

$$\phi = \tan^{-1}\left(\frac{2.5\text{mm}}{8.5\text{mm}}\right) - \tan^{-1}\left(\frac{2.5\text{mm} - 8(19.5\mu\text{m})}{8.5\text{mm}}\right) = 0.97^\circ$$

As shown in Table D.1, the maximum allowable rotation between images for various points in the image plane does not change very

much across the image. This is a result of the narrow field of view of the camera.

Table D.1 Maximum Rotation Between Images

Image Point (mm)	Max Rotation (deg)
0.0	1.05
0.5	1.05
1.0	1.04
1.5	1.02
2.0	1.00
2.5	0.97

However, if the camera had a wider field of view, there would be a significant difference between the maximum rotation for points close to the principal point and points close to the edges of the image plane. Table D.2, when contrasted to Table D.1, shows this disparity.

Table D.2 Maximum Rotation Between Images for a Wide FOV

Image Point (mm)	Max Rotation (deg)
3.0	0.94
3.5	0.90
4.0	0.87
4.5	0.83
5.0	0.79
5.5	0.75

The turning rate of the Micro-Rover will determine the time between 2 images. If the Micro-Rover is turning at a rate of 20° per second, the time between images should be about 50 msec because the allowable rotation between images is 1°. However, 25 msec between images is the upper limit if the Micro-Rover is turning at a rate of 40° per second.

The processing rate is not related to the time between images because the time between motion estimates is not equal to the time between images. The vision system will capture 2 images and perform some processing to estimate the motion between the images. Some time later, 2 more images will be grabbed and processed. The time between images may be 50 msec when the Micro-Rover is turning at 20° per second, but the time between motion estimates may be 200 msec, a 5 Hz processing rate.

The only constraint on the processing rate is that it be much faster than the rate at which the Micro-Rover changes direction. The mechanical structure of the Micro-Rover will be characterized by certain time constants. The time between motion estimates must be much smaller than all of these time constants. A reasonable minimum time constant for the mechanical structure is 1 second. A 5 Hz processing rate is reasonable under this assumption and will make certain that the motion of the Micro-Rover doesn't change much between motion estimates.

Appendix E Binocular Stereo Using Pattern Matching

As a final note, the use of the matching algorithm in a binocular stereo system was tested to determine whether a vision system could be built that was a robust, stand-alone piece of hardware. In our discussion up until this point, the intent was to solve the scale factor ambiguity problem using the laser range-finder. However, if the vision system were entirely self-contained, it would provide a much-desired level of redundancy to our Micro-Rover.

Another scene from the computer/storage room was used to test the binocular stereo system. The camera was positioned at 2 points that were 1 inch apart. The 6 DOF motion model was used with the translation equal to the 1 inch separation and the rotation equal to zero. The depth map was estimated from the set motion.

The results, as shown in Figure E.1, look pretty good, although there is some scaling problem. The NYNEX white pages were only 3 feet away, however they were estimated at about 4 feet. The stacked books were actually 4.5 feet away, although they were estimated at between 5 and 6 feet. It is suspected that the assumptions about the camera characteristics that we made earlier are finally coming back to haunt us. They may not have affected the previous estimates which didn't involve any lateral translation. However, in this case, we introduced lateral translation.

One interesting fact to note is the "discreteness" of the depth values; there are only about five different depth values. Since our lateral translation was limited to the x axis, only the u components of the flow vectors affected the depth estimates. The u component varied between 0 and 8 because we used a 16 x 16 data block in a 32 x 32 search window. Thus, only 9 possible values for the depth estimates were possible.

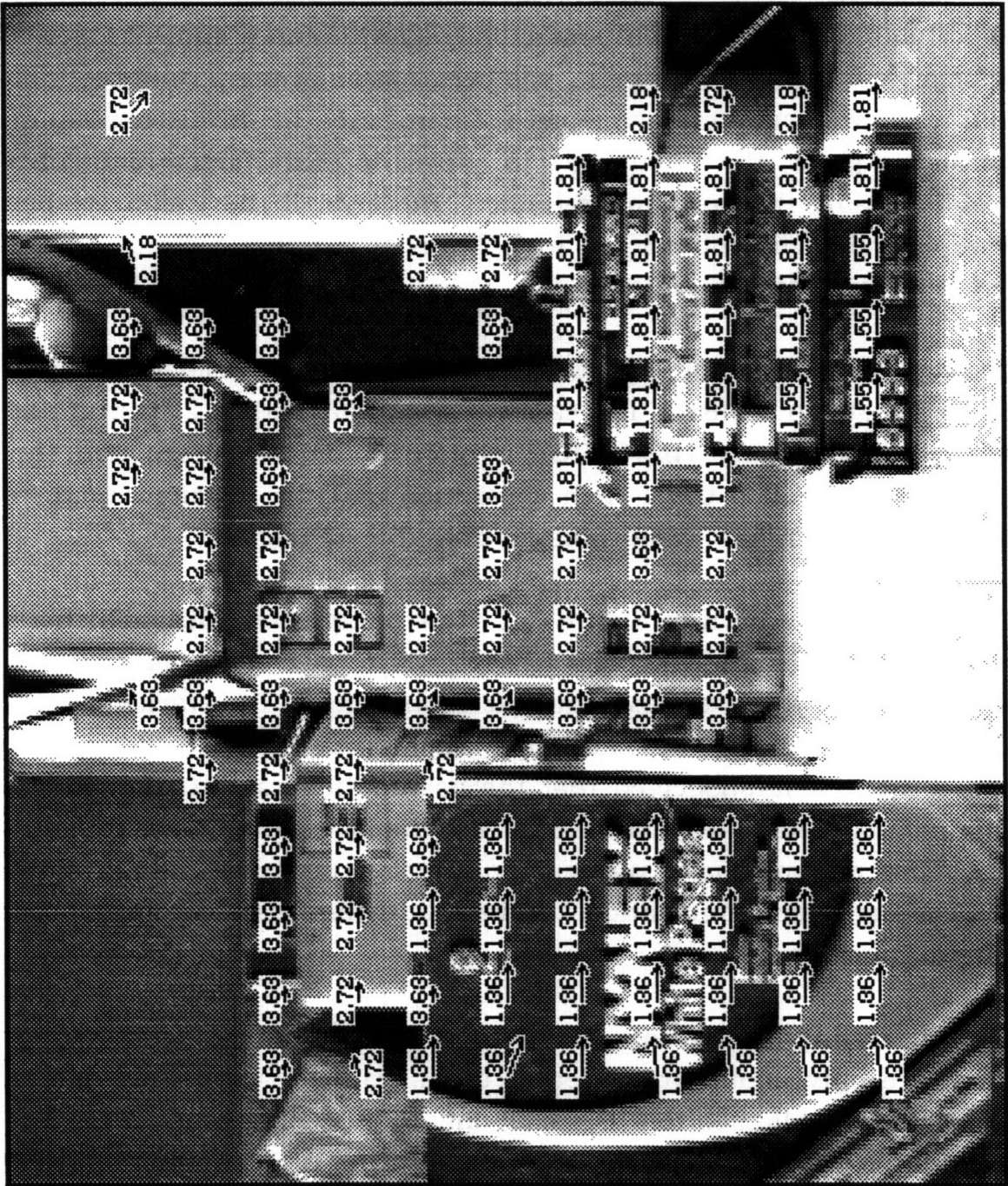


Figure E.1 Depth Map for a Binocular Stereo System Using Pattern Matching

References

- [1] Apple Computer Co., Inside Macintosh, v. 1-5, Addison-Wesley, Reading, MA, 1985-6.
- [2] David J. Braunegg, MARVEL: A System for Recognizing World Locations with Stereo Vision, MIT PhD Thesis in Electrical Engineering and Computer Science, June, 1990.
- [3] Anna R. Bruss and Berthold K. P. Horn, "Passive Navigation", *Computer Vision, Graphics, and Image Processing*, v. 21, no. 1, 1983, pp. 3-20.
- [4] Marco Campani, Marco Straforini, and Alessandro Verri, "A First Order Differential Technique for Optical Flow", Proceedings of SPIE, v. 1388, International Society for Optical Engineering, Bellingham, WA, 1991, pp. 409-414.
- [5] Olivier D. Faugeras and Steve Maybank, "Motion from Point Matches: Multiplicity of Solutions", *International Journal of Computer Vision*, v. 4, 1990, pp. 225-246.
- [6] John Gilbert, Design of a Micro-Rover for a Moon/Mars Mission, MIT SM Thesis in Mechanical Engineering, CSDL-T-1163, December, 1992.
- [7] Joachim Heel, "Direct Dynamic Motion Vision", Proceedings of the 1990 IEEE International Conference on Robotics and Automation, IEEE Computer Society Press, 1990, pp. 1142-1147.
- [8] Berthold K. P. Horn and Brian G. Schunck, "Determining Optical Flow", *Artificial Intelligence*, v. 17, 1981, pp. 185-203.
- [9] Berthold K. P. Horn and E. J. Weldon Jr., "Direct Methods for Recovering Motion", *International Journal of Computer Vision*, No. 2, 1988, pp. 51-76.

- [10] Berthold K. P. Horn, Robot Vision, MIT Press, 1988.
- [11] Berthold K. P. Horn, "Relative Orientation", *International Journal of Computer Vision*, v. 4, 1990, pp. 59-78.
- [12] Berthold K. P. Horn, "Relative Orientation Revisited", Unpublished paper, 1990.
- [13] Longuet-Higgins, H. C., and Prazdny, K., "The interpretation of a moving retinal image", Image Understanding 1984, Ablex Publishing Corp., 1984, pp. 179-193.
- [14] William Kaliardos, Sensors for Autonomous Navigation and Hazard Avoidance on a Planetary Micro-Rover, MIT SM Thesis in Aerospace Engineering, CSDL-T-1186, June, 1993.
- [15] LSI Logic Corp., *LSI Logic CCITT Video Compression Databook*, Sept. 1991.
- [16] Calvin Ma, Dynamics, Control, and System Simulation of a Planetary Rover, MIT SM Thesis in Mechanical Engineering, CSDL-T-1174, June, 1993.
- [17] Eric Malafeew, An Autonomous Control System for a Planetary Micro-Rover, MIT SM Thesis in Mechanical Engineering, CSDL-T-1173, May, 1993.
- [18] Hans P. Moravec and Alberto Elfes, "High Resolution Maps from Wide Angle Sonar", Proceedings of the IEEE Conference on Robotics and Automation, St. Louis, 1985.
- [19] Shahriar Negahdaripour and Berthold K. P. Horn, "Direct Passive Navigation", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. PAMI-9, No. 1, January 1987, pp. 168-176.

[20] Steven Schondorf, Systems Engineering for a Mars Micro-Rover, MIT SM Thesis in Aerospace Engineering, CSDL-T-1122, June, 1992.

[21] Gideon Stein, Internal Camera Calibration Using Rotation and Geometric Shapes, MIT SM Thesis in Electrical Engineering and Computer Science, February, 1993.

[22] Gilbert Strang, Introduction to Applied Mathematics, Wellesley-Cambridge Press, Wellesley, MA, 1986.

[23] Frederick Su, "Large-Area Scientific CCD's from Memory Device to Imager", *OE Reports*, No. 111, Society of Photo-Optical Instrumentation Engineers, Bellingham, WA, Feb. 1993.

[24] Raymond Suorsa and Banavar Sridhar, "Validation of Vision-Based Obstacle Detection Algorithms for Low-Altitude Helicopter Flight", Proceedings of SPIE, v. 1388, International Society for Optical Engineering, Bellingham, WA, 1991, pp. 90-103.

[25] M. Ali Taalebinezhaad, "Direct Robot Motion Vision by Fixation", Proceedings of the 1991 IEEE International Conference on Robotics and Automation, v. 1, IEEE Service Center, Piscataway, NJ, 1991, pp. 626-631.

[26] William B. Thompson and Ting-Chuen Pong, "Detecting Moving Objects," *International Journal of Computer Vision*, v. 4, 1990, pp. 39-57.

[27] Paul A. Viola, Adaptive Gaze Control, MIT SM Thesis in Electrical Engineering and Computer Science, October, 1990.

204