EGOMOTION ESTIMATION FOR VEHICLE CONTROL

MARK BROPHY

Egomotion Estimation for Vehicle Control

by

© Mark Brophy

B. Sc.(Honours), Memorial University of Newfoundland (2005)

A thesis submitted to the

School of Graduate Studies

in partial fulfilment of the

requirements for the degree of

Master of Engineering

Department of Engineering and Applied Science

Memorial University of Newfoundland

July 2007

St. John's Newfoundland

Abstract

The focus of this thesis is a technique called egomotion estimation, which involves the extraction of motion parameters from a camera based on the nature of the motion field on a frame-by-frame basis. In general, this is a multi-step process that involves estimating the motion field, often referred to as the optical flow, from which the translation direction and rotation are then extracted. The optical flow field is normally generated by tracking a frame's strong features in the subsequent frame of a sequence. Examples of strong features include corners of objects or areas of high contrast within an image. The algorithms described in this thesis have been developed with the hopes of eventually being utilized as the primary sensor on a Draganflyer four-rotor helicopter (also known as a quadrotor) for self-motion estimation. A PD controller was implemented to stabilize the quadrotor, and its effectiveness has undergone initial testing in simulation.

The algorithms and implementations that follow, in their initial implementations, took over one minute to find a result on an Intel 3.0Ghz Xeon system. They are now running at a rate of about $5Hz$, which is certainly a noteable difference. The methods presented are by no means optimal. The author is continuing this research on egomotion estimation as a part of his doctoral studies.

List of Notation

x	a column vector
x^T	the transpose of vector x
$\mathbf{x}$	a column vector constructed by "stacking" x_i column vectors on one another
C	a matrix of dimension $m \times n$
$C^\perp$	the orthogonal complement of matrix C
$u(x)$	$2d$ flow vector at point x
t	$3d$ translation direction vector
$\hat{t}$	an estimate of the translation direction
ω	$3d$ rotation vector
$d(x_i)$	the depth of point x_i
p_i	the inverse depth of point x_i
$R_x(\omega_x)$	a matrix for rotating a vector about the x-axis by ω_x degrees
$R_y(\omega_y)$	a matrix for rotating a vector about the y-axis by ω_y degrees
$R_z(\omega_z)$	a matrix for rotating a vector about the z-axis by ω_z degrees
u_1	input to Draganflyer for translation along the z-axis
u_2	input to Draganflyer for rotation about the z-axis
u_3	input to Draganflyer for translation along the x-axis
u_4	input to Draganflyer for translation along the y-axis
F_i	upward force created by rotor i on the Draganflyer, where $i = 1, ..., 4$

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Motivation

Most modern remotely-operated vehicles (ROVs) come equipped with some sort of optical sensor for target tracking or sensing and avoidance, but generally use some other type of sensor to obtain their heading and locale. Clearly the study of extracting such data from a visual sensor makes sense due to practicality (economics, minimization, etc.), but also because of the interesting problems that exist in the area. The process of extracting the $3d$ motion of a visual sensor from its captured images is referred to as egomotion estimation.

In navigational situations where the ROV is a robot or even an unmanned aerial vehicle (UAV), an inertial navigation system (INS) is almost always utilized in some form. When fused with another sensor (such as a global positioning device), the accuracy with which a vehicle can navigate is increased substantially. Rather the purpose of the following research is not to compete with such a solution. The purpose

is to work towards a vision-based solution that will someday be used in parallel with currently-available technology, or in low-cost systems where a great deal of accuracy is not critical. The author chose a monocular system for specifically this reason, since stereo vision carries a dual expense. Not only must one must purchase two cameras instead of one, but the weight added to the payload is doubled. The Draganflyer in particular could not handle having any weight added to its payload. Furthermore, utilizing a monocular system resulted in a complex and interesting problem.

It is an attractive problem in a number of ways. So much of a human's navigational ability comes from what he/she sees, it is interesting to see if it is possible for a robot to navigate using only visual data. Thus, this thesis deals with problems in both computer vision and control, and the relationship between the two.

This topic has been covered fairly extensively, but is often presented with little focus on implementation. The following thesis aims to describe the implementation in a thorough and clear manner, and includes an explanation of the singular value decomposition (SVD) which is a useful tool when solving the egomotion estimation problem. The major contribution of this thesis is a detailed mathematical explanation of how to obtain the orthogonal complement using the singular value decomposition, and how to subsequently implement a routine that will find the orthogonal complement of the column space of a matrix. From the beginning of this research, it has been the author's goal to determine whether or not controlling an unstable 6 degree-of-freedom ROV is possible when using vision as the sole sensor. Utilizing egomotion estimation to accurately determine the pose of a UAV is uncommon, and the author hopes that both his current and future contributions to this area will be of use to others. Finally, a somewhat unique method of simulation was used for the controller,

and this has been documented.

1.2 Previous work

Heeger and Jepson's [2] seminal paper on recovering 3D motion and depth from an optical flow field of an arbitrary scene is the basis of this thesis, yet it is but one of many contributions to the extensive body of literature that exists on the topic of rigid motion estimation.

The method in [2] can be defined as a "linear subspace method", in that the candidate solution space for the translation direction is a constrained $2d$ space. It makes use of the *orthogonal complement* to obtain $n-6$ linear constraints and solving for the translation direction independently of rotation and depth. Before [2], Bruss and Horn made use of the same algebraic constraint, but instead of iterating over the entire solution space, they utilized a least squares approach to minimize the difference between the measured optical flow and an ideal flow field for the extracted translational, rotational and depth information [3]. A nonlinear equation solver like gradient-descent or Newton's method was utilized to obtain a motion estimate. Like [3], the egomotion algorithm outlined by Tomasi and Shi in [4] also made use of a nonlinear equation solver. Instead of measuring how features move, as most methods proposed in the literature typically do, it measured how the image *deforms* over time. By monitoring how the angle α formed by pairs of projection rays changes over time (the derivative of α is called the image deformation), they constructed n bilinear constraints (like Bruss and Horn did in [3]) to solve for the translation direction [5]. They used the same input data (point-to-point correspondences) as the other

algorithms, but merely interpreted the data differently by not constructing an optical flow field.

Heeger and Jepson's method was initially chosen due to its iterative nature. Although it may yield incorrect results at times, it will not get stuck in local minima. This is due to the fact that the algorithm iterates over a a constrained $2d$ solution space, as is explained in more detail in section 3.1. Its formulation is also much easier to comprehend for those unfamiliar with the nonlinear solvers utilized by most other egomotion methods.

Egomotion estimation is used to extract the translation direction and the rotation of a body based on how features of interest move in successive frames of video from a mounted camera in a *static* environment. Static is emphasized here because the presence of undetected and unexpected entities will result in inconsistencies in the image flow field. One must generate the optical flow field using one of the many algorithms to generate such a field from which one will extract the rigid motion from frame-to-frame.

Optical flow algorithms can be divided into three different types of approaches: *discrete*, *differential* and *continuous* [3]. Lucas and Kanade's approach to generating optical flow fields is discrete in the sense that it attempts to find matching brightness patterns at a selection of points in an image sequence; that is, it generates a sparse field by utilizing a subset of the available pixels in an image. It is the chosen method for this thesis, but many other methods have been proposed. Horn and Schunck's derivative-based method, for example, introduced a constraint of smoothness to solve the aperture problem [6], an issue that is described in section 2.3. It finds the flow for the image pair using spatiotemporal derivatives [7]. Block-based matching would

be an example of a continuous method because it generates a flow vector for every pixel in the image based on how image pairs align.

Chapter 2

Vision System

The egomotion estimation algorithm was written in C++ and the OpenCV library was used extensively. OpenCV has excellent methods for both matrix and image manipulation, and also includes methods for feature extraction and optical flow calculation. All work was completed in Linux, and the source was compiled using g++. Manolis Lourakis' implementation of the Levenberg-Marquardt nonlinear least squares minimization algorithm [8] was utilized extensively, as was David Stavens' optical flow source code [9]. A Canon Powershot A75 was utilized to obtain both video and image pairs.

The process of finding the egomotion of two successive frames requires one to first find the optical flow, and then use the flow vectors to find the translation and rotation parameters of the camera based on the orientation of the flow vectors. To find the optical flow between two successive images, one must first extract the strong features from the first image, and then find the positions of these features in the second frame. What exactly is meant by *strong* features is explained in section 2.2. After the optical

flow has been found, the flow field can be used to find an egomotion estimate.

An vital operation in finding optical flow is the image derivative. This will be covered before feature extraction and optical flow calculation.

(a) Image A.

(b) Image B.

Figure 2.1: Two consecutive frames of the author's desk in the INCA lab.

2.1 Image brightness derivative

It is a bit strange to speak of the brightness derivative of a pixel value p_i, but a digital image is merely a discrete representation of a continuous image. With this in mind, one must acknowledge that all pixel derivatives are approximations. The spatial derivative of a digitized image is one of the most important operations in image processing [10], and as a result some good estimation kernels have been developed.

The basic derivative filters are

$$\begin{bmatrix} h_x \end{bmatrix} = \begin{bmatrix} 1 & 0 & -1 \end{bmatrix},$$
$$\begin{bmatrix} h_y \end{bmatrix} = \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}^t.$$

Unfortunately, high signal noise will be in the resultant image when we apply these filters, so normally convolution with one of these kernels will be coupled with some sort of smoothing filter. In this case, the Sobel operator will be used [10]

$$\begin{bmatrix} h_x \end{bmatrix} = \frac{1}{4} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -1 \end{bmatrix},$$
$$\begin{bmatrix} h_y \end{bmatrix} = \frac{1}{4} \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}.$$

Each filter takes the derivative in one direction and proceeds to smooth in the orthogonal direction.

2.2 Feature selection

Two common ways of obtaining an estimation of the optical flow for an image include

- calculating the flow for each individual pixel, and
- calculating the flow for good features

The latter of which will be the method used in this paper. Basically, a good feature is normally a corner in an image, or some small area where there is a great deal of contrast in two directions. Recognizing such an area is easy visually, but it is a little harder mathematically. Each pixel p_i in the image is iterated over, and the spatial gradient matrix G is obtained

$$G = \sum_{W} \begin{pmatrix} I_u^2 & I_u I_v \\ I_u I_v & I_v^2 \end{pmatrix}$$

where W is a square window (normally of size 3×3) with p_i at the center [11]. I_u and I_v are the horizontal and vertical derivatives of p_i, respectively. Let λ_1 and λ_2 be the eigenvalues of G, p_i is considered a candidate feature if

$$min(\lambda_1, \lambda_2) > \lambda_t,$$

where λ_t is some predefined threshold. Following this calculation, it is ensured that the distance between all of the candidate features are a sufficient distance d apart from one another. The strongest corners are considered first and those corners that are within d are pruned.

The strong features from image A can be seen in figure 2.2. Note the pixels that are selected as strong features. For the most part, they have some sort of dynamic contrast surrounding them.

Figure 2.2: Strong features are extracted from Image A and highlighted.

2.3 The aperture problem

The aperture problem simply states that the motion of a homogeneous contour is locally ambiguous [12]. A motion sensor has a finite view of its surroundings, and if such a contour occupies its entire image plane, different physical motions are indistinguishable from one another. For example, a set of parallel lines "moving from left to right will produce the same spatiotemporal structure as a set of lines moving from top to bottom". Figure 2.3 better illustrates this phenomenon.

2.4 Finding the optical flow

A flow vector must be generated for each of the strong features identified in an image. Define $p = \begin{bmatrix} p_x & p_y \end{bmatrix}^t$ as a pixel coordinate with $A(p)$ and $B(p)$ being defined as the greyscale values of images A and B at point p, respectively.

Given an image point $u = \begin{bmatrix} u_x & u_y \end{bmatrix}^t$ on A, for a location v on the second image

(a) (b) (c)

Figure 2.3: No matter how the aperture is moved, the perceived motion always remains indistinguishable. In 2.3(b), the aperture is moved to the right, while in 2.3(c) it is moved to the left.

such that

$$\begin{aligned} v &= u + d \\ &= \begin{bmatrix} u_x + d_x \\ u_y + d_y \end{bmatrix}, \end{aligned}$$

d is referred to as the "image velocity" or optical flow at u.

Finding d is normally accomplished by minimizing the error over a window ω, as opposed to a single point (due to the aforementioned aperture problem). The residual is defined as [13]

$$\epsilon(d) = \epsilon(d_x, d_y) = \sum_{x=u_x-\omega_x}^{u_x+\omega_x} \sum_{y=u_y-\omega_y}^{u_y+\omega_y} (A(x,y) - B(x+d_x, y+d_y))^2$$

In other words, the best residual comes from the values of d_x and d_y that minimize the difference between A and B's pixels in window ω. $A(u)$ is assumed be almost equal to $B(v)$.

The flow field between figures 2.1(a) and 2.1(b) is displayed in figure 2.4. There are a couple of outliers, as is often the case, but the field correctly indicates a translational

(left pointing arrows) difference between images A and B. For a more robust system, one must implement some statistical methods to eliminate these inaccuracies, as seen in [11]. This will be necessary, as can be seen with the supposed vertical motion that the flow vectors indicate (image B was taken following a pure translation of the camera after A was taken).

Now that the flow field has been obtained, it is possible (using some assumptions on image geometry) to extract the three dimensional motion of the camera between successive images.

Figure 2.4: The optical flow field induced between image A and image B.

Chapter 3

Motion Estimation

3.1 Translation estimation

Once a field of flow vectors has been obtained, one can make an estimate of the 3d motion that the camera has undergone between two successive frames. At least six accurate flow vectors must be sampled from an image pair to solve for $3d$ motion, as proved in [14]. Multiple point correspondences are needed to uniquely determine rigid motion, as the x, y and z components of both the rotation and the translation, as well as the relative depth, must be solved for. If the samples are noisy, it will be necessary to use more flow vectors.

When estimating egomotion, one is given m 2d flow vectors u_i and their respective positions x_i in the image. The fundamental equation in egomotion estimation relates a 2d flow vector u at position $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ to its rigid $3d$ motion

$$u(x) = d(x)A(x)t + B(x)\omega, \tag{3.1}$$

where

$$A(x) = \begin{bmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -x_2 \end{bmatrix},$$

$$B(x) = \begin{bmatrix} -x_1x_2 & 1+x_1^2 & -x_2 \\ -1-x_2^2 & x_1x_2 & x_1 \end{bmatrix}.$$

The goal is to find the 3d motion parameters $t = [t_1, t_2, t_3]^T$ and $\omega = [\omega_1, \omega_2, \omega_3]^T$, as well as the depth vector d containing the depth $d(x_i)$ at each point x_i from the input data.

Say that m points are sampled from strong features in the image pair. The m 2d flow vectors can be "stacked" on top of one another to form a new $2m$ vector of the form

$$\mathbf{u} = \begin{bmatrix} u_1(1) & u_1(2) & \cdots & u_m(1) & u_m(2) \end{bmatrix}^T,$$

Then an equation can be written in the following way [2]

$$\mathbf{u} = A(t)p + B\omega \tag{3.2}$$

$$= C(t)q, \tag{3.3}$$

where

$$A(t) = \begin{bmatrix} A(x_1, y_1)t & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & & A(x_n, y_n)t \end{bmatrix},$$

$$B = \begin{bmatrix} B(x_1, y_1) \\ \vdots \\ B(x_n, y_n) \end{bmatrix},$$

and

$$q = \begin{bmatrix} \omega_1 & \omega_2 & \omega_3 & p_1 & \cdots & p_m \end{bmatrix}^T,$$

where p_1,..., p_m are the inverse depths of the m aforementioned points x_i. In other words, $p_i = 1/d(x_i)$. Then $C(t)$ can be written as

$$C(t) = \begin{bmatrix} | & | \\ A(t) & B \\ | & | \end{bmatrix}. \tag{3.4}$$

Notice that $A(t)$ and B only rely on the positions of the flow vectors and the translation vector t, where t can be any vector on the unit sphere. It is recommended by [2] to precalculate $A(t)$ and B for every possible t at every image position. Then when the algorithm is running, it is only a matter of looking up the results of an otherwise extremely time-consuming operation.

The translation vector can be thought of as any vector on the unit sphere. Iterating over the candidate space is easy when spherical coordinates are used. By using $-180 \geq \theta \leq 180$ and $0 \geq \phi \leq 180$ and $\rho = 1$, all whole-numbered vectors are represented. The conversion to Cartesian is as follows:

$$\begin{aligned} t_1 &= \rho \sin\phi \cos\theta, \\ t_2 &= \rho \sin\phi \sin\theta, \\ t_3 &= \rho \cos\phi. \end{aligned}$$

Since $\rho = 1$, the solution space of the recovered translation is actually $2d$, as can be seen in figure 3.1, which is a plot of the residual surface of the translation direction between images 3.2(a) and 3.2(b), the minimum of which was $t =$

$\begin{bmatrix} 0.918465 & 0.139034 & 0.370258 \end{bmatrix}$. As an aside, the recovered rotation was $\omega = \begin{bmatrix} 0.007685727 & -0.09866409 & 0.05950551 \end{bmatrix}$, in radians.

The residual function, $E(t)$, is defined over the the entire candidate translation space

$$E(t,q) = ||\mathbf{u} - C(t)q||^2 . \tag{3.5}$$

Equation 3.3 states that the optical flow at n points equals $C(t)q$, so it makes sense that the (t,q) pair resulting in the smallest least squares estimate in equation 3.5 would be the best prediction of the translation, rotation and depth.

As shown in the appendix of [2], it is possible to reduce equation 3.5

$$E(t) = \left|\left|\mathbf{u}^T C^{\perp}(t)\right|\right|^2 , \tag{3.6}$$

and solve first for only t. The candidate t that minimizes the residual will also yield a minimal residual value in equation 3.6 as in equation 3.5. Thus, t can then be used to solve for q. Calculation of $C^{\perp}$, the orthogonal complement, is covered next.

3.1.1 Calculating $C^{\perp}$

Any $m \times n$ matrix A can be written as [15]

$$A = USV^T, \tag{3.7}$$

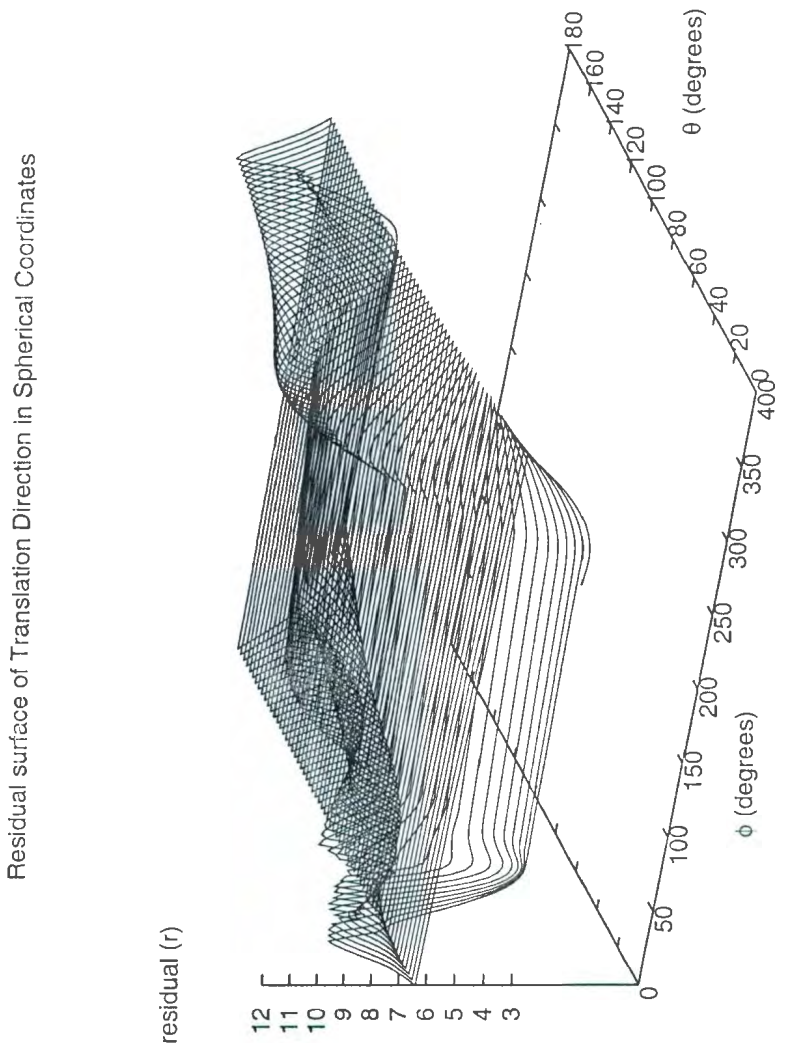
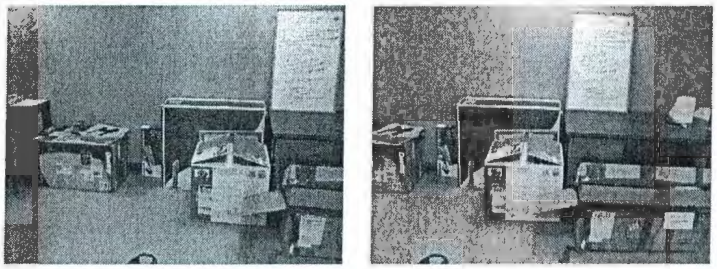


Figure 3.1: A plot of the residual surface of the translation direction incurred between the capturing of figures 3.2(a) and 3.2(b). The solution space can be expressed in $2d$ using spherical coordinates.



(a) (b)

Figure 3.2: 3.2(b) was taken after 3.2(a) and a substantial camera translation was incurred. The flow field is fairly noisy and results in residual values which are quite large on the residual surface.

using singular value decomposition (SVD) where the following *orthogonal* matrices exist

$$U = \left[u_1, ..., u_m \right] \in \mathbb{R}^{m \times m},$$
$$V = \left[v_1, ..., v_n \right] \in \mathbb{R}^{n \times n},$$

and $\Sigma_1 \in \mathbb{R}^{m \times n}$ with the form

$$\begin{bmatrix} \sigma_1 & 0 & ... & 0 \\ 0 & \sigma_2 & & \\ ... & & ... & \\ 0 & & & \sigma_p \end{bmatrix}.$$

and $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_p \geq 0$ where p is the lesser of m and n.

Let the rank of $A = r$. This is also the rank of $A^t A$ and exactly the number of nonzero eigenvalues in Σ [15]. Define

$$V_1 = [v_1, ..., v_r],$$

(a) Computer-generated image A.

(b) Computer-generated image B.

(c) The resultant flow field of images A and B overlayed on A.

Figure 3.3: Two computer-generated frames that are often seen in egomotion estimation literature. The translation direction incurred by the camera between the two "captures" is precisely along the z-axis.

and

$$V_2 = [v_{r+1}, ..., v_n],$$

as the set of eigenvectors associated with the nonzero and zero eigenvalues in Σ, respectively. Using this notation, it may be easier to think of Σ as

$$\Sigma = \begin{bmatrix} \Sigma_1 & 0 \\ 0 & 0 \end{bmatrix}$$

where $\Sigma_1 = \text{diag}(\sigma_1, \sigma_2, ..., \sigma_r)$ and the zero vectors fill Σ such that its dimensions are $m \times n$.

Now, from equation 3.7

$$A = U\Sigma V^t$$

multiply both sides by V

$$AV = U\Sigma V^t V.$$

Since V is orthogonal, it follows that

$$AV = U\Sigma,$$

furthermore

$$[Av_1, ..., Av_r, Av_{r+1}, ..., Av_n] = [\sigma_1 u_1, ..., \sigma_r u_r, 0, ..., 0].$$

Hence

$$Av_j = \sigma_j u_j$$

for $j = 1, ..., r$. The resulting vector formed from Av_j is of the same dimension as u_j, so $u_j = Av_j$ multiplied by some scaling factor $(1/\sigma_j)$ for $j = 1, ..., r$. It then makes

sense to split U into $[U_1, U_2]$ where

$$\begin{aligned} U_1 &= [u_1, ..., u_r], \\ U_2 &= [u_{r+1}, ..., u_m], \end{aligned}$$

$$Av_j = 0u_j, \text{for } j = r+1, r+2, ..., m$$

So

$$Av_j = 0, \text{for } j = r+1, r+2, ..., m$$

U is, by definition, an orthogonal matrix [15]. As shown by [16], the first r columns of U form an orthonormal basis for the column space of A, the matrix that we are decomposing. Since U is an orthogonal matrix, each of its remaining $m - r$ column vectors (U_2) are also orthogonal both every vector in U_1 and every other vector in U_2. U_2 therefore has $m - r$ column vectors, and it indeed forms a basis for the left null space of A [16], aka: the orthogonal complement of the column space of A.

The orthogonal complement is $[u_{r+1}, ..., u_m]$.

3.2 Rotation estimation

Once the translation t between the two frames has been found, obtaining the rotation ω is a much less complicated ordeal. It can be calculated by solving the linear least squares problem as described by Zhang and Tomasi [17], where vectors are stacked in the same fashion as they were in the translation estimator.

$$\omega_k = \underset{\omega}{\text{argmin}} \frac{1}{m} \sum_{\{x\}} ||\tau_k(x)(u(x) - B(x)\omega)||^2, \tag{3.8}$$

where $\{x\}$ is the set the m flow vectors' positions and τ is a unit-norm vector that is orthogonal to

$$e = \frac{A(x)t}{||A(x)t||}.$$

Solving equation 3.8 using the Levenberg-Marquardt nonlinear least squares algorithm (as seen in section 3.3.2) will yield an estimate of ω.

3.3 Realtime motion estimation

The previously described algorithm was first tested on a pair of computer-generated images, as seen in figures 3.3(a) and 3.3(b). Between them, the virtual camera performed a translation along the positive z-axis. In other words, $t = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T$ precisely, with a rotation of zero in all three axes. Using the previously described implementation of Jepson and Heeger's egomotion estimation algorithm, the estimated translation from the flow field in figure 3.3(c) was $\hat{t} = \begin{bmatrix} 0.00639774 & 0.0257688 & 0.999647 \end{bmatrix}^T$. While the algorithm itself was sound, it took well over a minute to process the translation direction alone between the two images in the pair. This is often the case with egomotion algorithms, and herein lies the crux of this thesis: to use vision as the primary system for the control of a UAV. In order to control a UAV successfully, estimates must be obtained at a much faster rate. Since the helicopter utilized in this thesis is controllable (with difficulty) by an operator, updates must arrive at a rate of approximately $10Hz$ (human reaction time). This is especially true for rotary vehicles, as they are inherently less stable than their fixed-wing counterparts. Thus, various optimizations were needed to drastically improve the performance of the motion estimation.

Initially, it was suspected that it would be possible to make egomotion estimates at a rate of about 3-4 frames per second using Heeger and Jepson's method [2]. However, it was found that a translation estimate between a pair of frames was taking upwards of a minute, even on the aforementioned Xeon system. Upon completion of the implementation. Obviously, this was not sufficient. Furthermore, when the camera induces a large displacement between frames, even when pyramidal Lucas-Kanade optical flow estimation with a very low error tolerance and a large number of iterations, optical flow estimation breaks down due to a large number of false positives. Thus, the faster the motion of the body, the more frequent estimates must be completed.

One technique used was to decrease the number of features utilized in the flow field. In [9], 400 features are used. For operation in realtime, 50 features were utilized in finding frame-to-frame correspondences. This means that the orthogonal complement can be found much more quickly than before; the singular value decomposition of a smaller set of vectors is less taxing in terms of memory usage and CPU cycles. The found features with the smallest error in the scene were utilized, so little was lost in terms of accuracy when recovering the rigid motion.

3.3.1 Realtime translation estimation

In [2], it is stated that the candidate translation space is the unit sphere, but that only half the unit sphere needs to be considered since the solutions on the front and back halves are identical. Furthermore, the solution space is small, so the residual function can be evaluated "using a practical amount of memory and compute time"

[2]. Unfortunately, even using only the front half of the unit sphere it was found that calculation of the translation direction in less than a second was still impossible. The solution space was 3600 different unit vectors, and even increasing ϕ and θ by a factor of 2 when iterating over the candidate solution space (thus reducing the solution space by a factor of four) still took too long. However, it was noticed during the evaluation of residuals over the unit hemisphere that the "correct" estimate often differed from the next-best estimate by approximately 0.0001. Thus, by setting the stopping error ($||\epsilon||^2$) to 0.0001 in the Levenberg-Marquardt solver, good predictions of the translation direction became obtainable in on the order of tens of seconds with the unconstrained solver.

In further attempts to speed up the results, a heuristic method was created by the author. Given that the camera is attached to a body that is governed by some equations of motion, one can make certain assumptions on the nature of motion. Since it is necessary to obtain an estimate of optical flow from nearly every consecutive pair of frames, limits may be put on rate at which the translation may vary between frames.

$\hat{T}$ is defined as the initial estimate of the translation direction of the camera. This estimate is used only for finding the translation direction when evaluating the first two frames of the video stream. After this, the recovered T from the previous flow field becomes the new $\hat{T}$.

A patch surrounding the vector on the unit sphere is iterated over for each $\hat{T}$. The candidate vector, $\hat{T}$, is converted to spherical coordinates, thus obtaining a vector of the form $\hat{T}_{sph} = \begin{bmatrix} 1 & \phi & \theta \end{bmatrix}$ (since $\rho = 1$), and the solution space is all vectors such that $\theta - 10 \leq \theta_i \leq \theta + 10$ and $\phi - 10 \leq \phi_i \leq \phi + 10$. This method was abandoned after it was revealed that the update rate was still insufficient for online

estimation. Furthermore, if a number of consecutive poor estimates were made on the translation direction, the algorithm would drift out of the correct solution space and would continue to deliver poor estimates for some time, even if the predictions were utilizing good flow fields!

Substantial effort was put into using a nonlinear least squares solver to obtain a realtime solution. Manolis Lourakis' C/C++ implementation of the Levenberg-Marquardt algorithm [8] was utilized. Out of the box, the library took multiple seconds to obtain a translation estimate (as mentioned above), so box constraints were placed on the nonlinear solver as follows

$$-1.0 \leq T_x \leq 1.0, \tag{3.9}$$

$$-1.0 \leq T_y \leq 1.0, \tag{3.10}$$

$$-1.0 \leq T_z \leq 1.0. \tag{3.11}$$

3.3.2 Levenberg-Marquardt method

Levenberg-Marquardt is a technique that finds the minimum of a multivariate function in an iterative fashion. Given a function f, the method tries to find a parameter vector p that minimizes the difference between the estimated measurement vector $\hat{x}$ and input x, the measured vector. More specifically,

$$\hat{x} = f(p), \hat{x} \in R^n. \tag{3.12}$$

Aside from the measured vector, the algorithm also requires an initial guess as input, p_0. The best possible guess, p^+, minimizes the squared distance $\epsilon^T \epsilon$, where $\epsilon = x - \hat{x}$

For a small step size ($||\delta_p||$),

$$f(p + \delta_p) \approx f(p) + J\delta_p \tag{3.13}$$

where J is the Jacobian matrix of f. At each step, the algorithm tries to find the δ_p that minimizes $||x - f(p + \delta_p)||$, which is approximately equal to $||x - f(p) - J\delta_p|| = ||\epsilon - J\delta_p||$. The minimum is attained when $J\delta_p - \epsilon$ is orthogonal to the column space of J

$$J^T(J\delta_p - \epsilon) = 0 \tag{3.14}$$

which leads to

$$J^T J\delta_p = J^T\epsilon \tag{3.15}$$

In 3.15, $J^T J$ is an approximation of the Hessian. The Levenberg-Marquardt method usually makes a slight modification of this matrix

$$N\delta_p = J^T\epsilon \tag{3.16}$$

where the off-diagonal elements of N are equal to the corresponding elements in $J^T J$, but the elements on the diagonal are such that

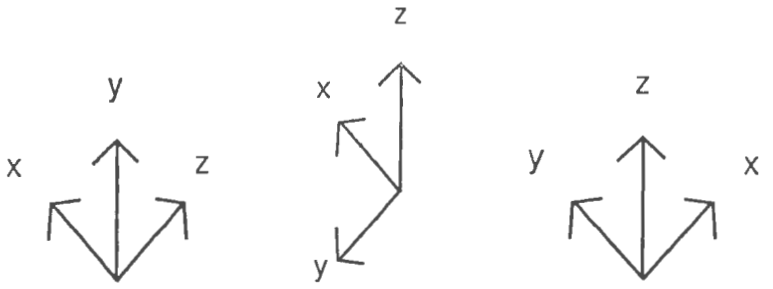
$$N_{ii} = \mu + \left[J^T J\right]_{ii} \tag{3.17}$$

where $\mu > 0$ is a *damping term.*

3.3.3 Coordinate transformation

Since the camera is the sensor upon which the control laws are based, it is necessary that the positional estimates it gives are in the same space as the Draganflyer. In this

configuration, it is possible to simply relabel the axes to transform the camera's coordinate system to that of the Draganflyer. Prior to (and possibly following) the writing of this thesis, the camera's position was (will be) such that merely relabeling the axes was (is) impossible. For such situations, the following coordinate transformation is very relevant.



(a) Camera (b) Intermediate (c) Draganflyer

Figure 3.4: Coordinate system 3.4(a) must be converted to 3.4(c).

The conversion from the camera system to the Draganflyer system is a two step process. First, a -90^o rotation about the x-axis will result in the coordinate system seen in figure 3.4(b), followed by a rotation -90^o rotation about the z-axis in the new system. This will yield the Draganflyer coordinate system as seen in 3.4(c).

The sign of the rotation direction about the axis is determined using the right hand rule. Simply point your right thumb along the axis in question in the positive direction, and curl your fingers. The direction in which the fingers curl in the direction of positive rotation.

Say one wishes to transform a vector in 3D space, and thus has a vector of the form $x = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}^t$. The dimension of the vector is extended and given a value

of 1 (thus becoming of the form $x = \begin{bmatrix} x_1 & x_2 & x_3 & 1 \end{bmatrix}^t$), and then is multiplied by the appropriate rotation matrices:

$$R_x(\theta_x) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x & 0 \\ 0 & \sin\theta_x & \cos\theta_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3.18}$$

$$R_y(\theta_y) = \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3.19}$$

$$R_z(\theta_z) = \begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 & 0 \\ \sin\theta_z & \cos\theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.20}$$

So, any translation or rotation vector that we obtain from the camera will have to be manipulated in the following way

$$\hat{x} = R_z(R_x x),$$

where $\theta_x = 90^o$ and $\theta_z = -90^o$. The new vector, $\hat{x}$, may now be utilized in the PD controller seen section 4.2.1.

Chapter 4

Quadrotor Model and Control

The Draganflyer's motions are normally manipulated using a Futaba 4-channel remote controller by a human. To obtain autonomous flight the same controller is used, but it is connected to a PC via a PCBuddy cable, which essentially converts RS232 commands to PWM (pulse width modulated) signals allowing for wireless control of the aircraft. The egomotion algorithm supplies the yaw(ϕ), pitch(ψ) and roll(θ) to the PD controllers.

4.1 Draganflyer dynamics

Two of the rotors of the Draganflyer rotate clockwise, while two rotate counter-clockwise. Adjacent rotors spin in opposite directions, as can be seen by the thin arrows in figure 4.1. The logic behind controlling the Draganflyer is fairly simple. By modifying the individual motors' speeds, the helicopter can be maneuvered in both directions along the x, y and z axes. Note in figure 4.1 that the z-axis is in the upwards/downwards direction, a common coordinate system used in robotics

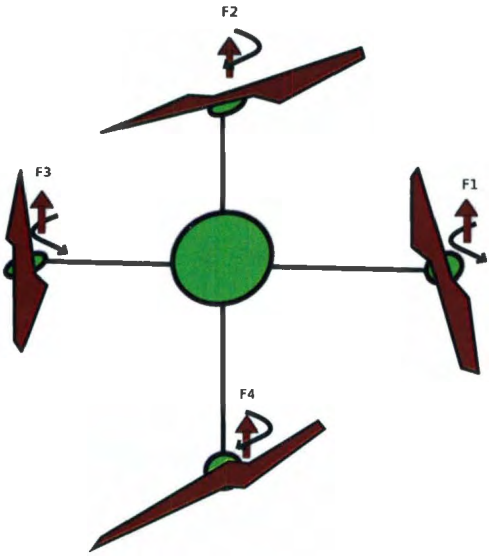


Figure 4.1: A sketch of the Draganflyer. The thin arrows indicate the direction in which the corresponding rotors move, while the thick red arrows indicate the forces.

literature. To increase the height of the Draganflyer, increase the speeds of all four rotors simultaneously. Utilizing the notation introduced in figure 4.1, this means increasing F_1, F_2, F_3 and F_4. Motion along the positive x-axis can be obtained by increasing the speeds of rotors 3 and 4, and decreasing the speeds of rotors 1 and 2. When thought of visually, this will result in a rotation about the y-axis, increasing the altitudes of rotors 3 and 4 and decreasing the altitudes of rotors 1 and 2. Once the desired tilt (the degree of rotation about the y-axis) has been reached, the four rotors will return to equal speeds, and the thrust of the angled body will propel it along the x-axis.

The same can be said for motion along the positive y-axis, except that the speeds of rotors 2 and 3 will be decreased while the speeds of rotors 1 and 4 will be increased. The altitudes of rotors 1 and 4 increase, the altitudes of rotors 2 and 3 decrease, and

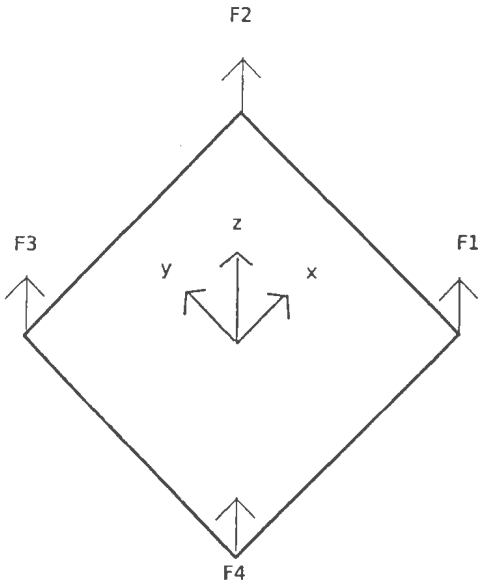


Figure 4.2: 3D quadrotor model, as adapted from [1]. Note that the x and y-axes are *not* parallel to the rods that make up the frame, as seen in 4.1. In fact, the axes are perpendicular to them.

the body then rotates about the x-axis. Once the desired tilt (about the x-axis) has been reached, the four rotors should return to equal speeds, and the quadrotor will move along the positive y-axis. For both the x and y axes, motion in the negative direction can be obtained by instead increasing the speeds of the motors that were decreased, and decreasing the speeds of the motors that were previously increased.

A clockwise motion about the z-axis can be obtained by increasing the speeds of rotors 2 and 4. This will produce a moment larger than the opposing moment created by rotors 1 and 3 in the opposite direction. A counter-clockwise motion can be obtained by increasing the speeds of rotors 2 and 4 instead.

The four inputs to the system will be defined with the previous statements in

mind. Table 4.1 gives both descriptions and values for the constants used in defining the inputs and the system equations, as seen in [18]. Note that the inputs are in different units. u_1 represents the total thrust on the body along the $z-$axis, while u_2 and u_3 are pitch and roll inputs and u_4 is the yawing moment [1].

Table 4.1: Draganflyer model parameters

Parameter	Description	Value	Units
g	gravity	9.81	m/s^2
m	vehicle mass	0.468	kg
J_1	roll inertia	4.9×10^{-3}	$kg \cdot m^2$
J_2	pitch inertia	4.9×10^{-3}	$kg \cdot m^2$
J_3	yaw inertia	8.8×10^{-3}	$kg \cdot m^2$
l	center to blade length	0.225	m

From [1],

- $u_1 = (F_1 + F_2 + F_3 + F_4)/m$

  to increase the lift, increase the thrust of all four rotors equally.

- $u_2 = (-F_1 - F_2 + F_3 + F_4)/J_1$

  to translate along the positive x-axis, increase thrusts of rotors 1 and 4 equally [19].

- $u_3 = (-F_1 + F_2 + F_3 - F_4)/J_2$

  to translate along the positive y-axis, increase thrusts of rotors 3 and 4 equally.

- $u_4 = C(F_1 - F_2 + F_3 - F_4)/J_3$

  to perform a clockwise rotation about z, increase thrusts of 1 and 3 to overcome the moment created by 2 and 4. C is the force-to-moment scaling factor, valued at 1.3 through experimental analysis in [20].

Finally, the model for the quadrotor used in the simulations is defined as

$$\ddot{x} = u_1(\cos\phi \sin\theta \cos\psi + \sin\phi \sin\psi) \tag{4.1}$$

$$\ddot{y} = u_1(\sin\phi \sin\theta \cos\psi - \cos\phi \sin\psi) \tag{4.2}$$

$$\ddot{z} = u_1(\cos\theta \cos\psi) - g \tag{4.3}$$

$$\ddot{\theta} = u_2 l \tag{4.4}$$

$$\ddot{\psi} = u_3 l \tag{4.5}$$

$$\ddot{\phi} = u_4 \tag{4.6}$$

4.2 PD control theory

A PD controller is a form of a proportional, integral, derivative (PID) controller, where each of these elements are used to control a plant. These elements take the feedback from the plant and the system command signal, and use them to produce the system output [21].

The derivations of the PD controllers that follow in section 4.2.1 are quite theoretical in nature, and are essentially the same as the controller found in [1]. While both [1] and [20] both do an excellent job deriving the controller, they give little to no implementation details.

On the other hand, [21] offers a very good tutorial on how to implement a PD

controller in C, as well as how to estimate quantities like $\dot{\psi}$, $\dot{\phi}$, $\dot{\theta}$ and $\dot{z}$. Much of the code written for controlling the Draganflyer was based on it. Controls for the yaw(ϕ), pitch(θ), roll(ψ) and height(z) are derived in [1] by linearizing about hovering mode (ie: $\theta = \phi = 0$, $u_1 = 1$). Indeed, the controller that was implemented by the author was designed for the sole purpose of getting the quadrotor to hover. The purpose of this thesis from the outset was to demonstrate that optical flow is accurate enough to be used as the primary sensor for controlling an inherently unstable vehicle, not to implement a full-blown control system responsible for the navigation of a quadrotor.

4.2.1 PD controller

From the $\ddot{y}$ term in equation 4.1, if θ, and ϕ are set to 0 and u_1 is set to 1, one obtains

$$\ddot{y} = K_p\,(y_d - y) + K_d\,(\dot{y}_d - \dot{y}) = -sin\psi.$$

Note that y_d and $\dot{y}_d$ are both zero, so

$$\ddot{y} = -K_p y - K_p \dot{y} = -sin\psi. \tag{4.7}$$

By negating the arcsin of both sides of equation 4.7, we get

$$\psi_d = arcsin(K_p y + K_d \dot{y}), \tag{4.8}$$

where ψ_d is the desired tilt angle. If the derivative of equation 4.8 is taken,

$$\dot{\psi}_d = \frac{K_p \dot{y} K_d \ddot{y}}{\sqrt{1 - K_p^2 y^2 - 2K_p K_d y\dot{y} - K_d^2 \dot{y}^2}}, \tag{4.9}$$

an expression for the *desired tilt angle velocity* is obtained.

The motion along the y-axis can be controlled by using a PD controller for input u_3, which as indicated in 4.1 equals $-F_1 + F_2 + F_3 - F_4$. By decreasing the thrust

of two adjacent rotors, the helicopter rotates around the axis upon which they both lie (F_1 and F_4 share the x-axis). The helicopter will then translate in the direction of the downward tilt (along the y-axis) once the desired tilt has been acquired and (approximately) equal thrust has been restored among all rotors. To obtain the desired tilt, u_3 is controlled via a PD controller

$$u_3 = K_{p1}\left(\psi_d - \psi\right) + K_{d1}\left(\dot{\psi}_d - \dot{\psi}\right), \tag{4.10}$$

using the aforementioned definitions of ψ_d and $\dot{\psi}_d$.

In the same way, one can design a controller for motion along the x-axis. This time u_2 will be modified by decreasing F_1 and F_2, resulting in rotors 1 and 2 tilting downward until the desired θ_d has been achieved, at which point (approximately) equal thrust will be returned to all rotors and the helicopter will translate in the direction of the downward tilt of rotors 1 and 2 (along the x-axis). If one assigns $\phi = \psi = 0$ and $u_1 = 1$,

$$\ddot{x} = sin\theta, \tag{4.11}$$

Now, to control $\ddot{x}$ using a PD controller, [1] asserts

$$\ddot{x} = -K_p\left(x_d - x\right) + K_d\left(\dot{x}_d - \dot{x}\right). \tag{4.12}$$

By equating the right-hand sides of equations 4.11 and 4.12

$$u_2 = K_{p2}\left(\theta_d - \theta\right) + K_{d2}\left(\dot{\theta}_d - \dot{\theta}\right), \tag{4.13}$$

where

$$\theta_d = arcsin\left(-K_p x - K_d \dot{x}\right), \tag{4.14}$$

$$\dot{\theta}_d = -\frac{K_p \dot{x} + K_d \ddot{x}}{\sqrt{1 - K_p^2 x^2 - 2K_p K_d x\dot{x} - K_d^2 \dot{x}^2}}. \tag{4.15}$$

Yawing motion is much less complicated, as rotation about ϕ does not result in a translation along any axis. Thus, ϕ_d and $\dot{\phi}_d$ can be defined arbitrarily, either by the operator or some intelligent system that is operating at a higher level than the PD controller. In [1], the following is given

$$u_4 = K_{p4}(\phi_d - \phi) + K_{d4}(\dot{\phi}_d - \dot{\phi}). \tag{4.16}$$

Chapter 5

Simulation Results

5.1 Egomotion simulations

In the previous sections, egomotion estimates were performed on various image pairs and video sequences. Determining the degree of accuracy was difficult for two reasons:

- The focal length of the (Canon A75) camera used in egomotion estimation calculations is an approximation.
- It is extremely difficult to generate highly accurate image sequences due to factors like unleveled tables and the lack of a turret for controlling rotation accurately to multiple decimal places.

Conversely, computer-generated images make this task much easier:

- The exact focal length of the "camera" is known (defaults to 1).
- Sequences in which the camera-induced motion is known can be generated with ease.

Povray (shorthand for Persistence of Vision Ray-Tracer) renders $3d$ scenes with a technique called raytracing [22]. As input, Povray reads a text file containing information on the camera, objects and lighting that are contained in a scene. It was utilized to generate various image sequences and test how precisely the egomotion of the camera could be extracted. Computer-generated imagery proved extremely useful when creating image sequences containing combined motions that would have been quite difficult to perform by hand. Generating a pure translation along the x-axis, for example, is a fairly easy task. Generating an accurate sequence when the camera is translating along the z-axis while rotating precisely 2^o is quite another story though. Figures 5.1(a), 5.1(b), 5.2(a) and 5.2(b) were all generated with Povray, and the egomotion estimates from the generated flow fields can be seen in table 5.1.

5.2 Combined vision and controller simulations

Sample C(++) code exists for interfacing a PC with a remote controller via the PCBuddy and a plethora of information is available for designing PID controllers in said language(s). C++ was chosen for the controller based on this, but also because of its great performance.

It is often the case that when one is implementing a controller for a system, he/she designs both a model and a corresponding controller in Matlab to test the general performance of the gains, and then writes the actual controller in C or some other language. In this case, the model was coded in Matlab, but the controller was not. Instead, it was written in C++ and a two-way communication channel was implemented between the two (each ran on a separate machine). During the

simulations, the model would send a message containing the ϕ, ψ and θ as well as the x, y and z displacements, and the controller would send back the control inputs (u_1, u_2, u_3 and u_4). Each time the model received a control update, it plugged the inputs into the *ode45* solver along with the system model for 0.2 seconds, and returned the resultant rotational and translational displacements to the controller. These can be seen in the graphs at the end of the chapter. The *ode45* solver integrates a system of differential equations over a user- defined timespan [23]. A function written by the user contains the equations, and the handle of this function is passed as an argument to the solver. Matlab solves the system numerically.

After running simulations on both the vision and control algorithms separately, the image sequences in figures 5.1(b), 5.2(a) and 5.2(b) were inputted into the full-blown implementation, where

- the optical flow field was generated from point-to-point correspondences,
- the estimate for the translation direction was made from the flow field,
- the estimate for the rotation was made from the recovered translation direction and the flow field,
- coordinate transformations (as described in section 3.3.3) were performed on the translation direction and rotation to convert them to the draganflyer frame from the camera frame, and
- the transformed rotation values were then passed as input to their respective PD controllers.

The resultant output of the PD controllers can be seen in Table 5.2. A rotation of 2^o about the z-axis in the camera frame is a 2^o (pitching) rotation about the x-axis in the Draganflyer frame. Thus, when the camera rotates about z-axis in figures 5.1(b) and 5.2(b) it makes sense that u_2 is the largest in magnitude.

(a) Translation on the negative z-axis with zero rotation.

(b) Rotation of 3 degrees about the z-axis in the positive direction with no translational component.

Figure 5.1: Estimating the egomotion of a simulated camera in a sparse Povray scene. 5.1(a) contains only translation, while 5.1(b) contains only rotation.

(a) Translation along positive z-axis with positive rotation about the x-axis of 2 degrees.

(b) Translation along positive x-axis with positive rotation about z-axis of 2 degrees.

Figure 5.2: Again estimating the egomotion of a simulated camera, but this time on scenes with both translational and rotational components.

flow diagram	actual t	estimated t	actual ω	estimated ω
5.1(a)	$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 0.0642482 \\ 0.197736 \\ 0.978148 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} -0.002583029 \\ 0.02339799 \\ 0.001033989 \end{bmatrix}$
5.1(b)	n/a	n/a	$\begin{bmatrix} 0 \\ 0 \\ 0.052333333 \end{bmatrix}$	$\begin{bmatrix} 0.002519399 \\ 0.01115045 \\ 0.03215757 \end{bmatrix}$
5.2(a)	$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$	$\begin{bmatrix} 0.431632 \\ -0.371138 \\ 0.822162 \end{bmatrix}$	$\begin{bmatrix} 0.03490658 \\ 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 0.0266253 \\ 0.0168376 \\ 0.0180777 \end{bmatrix}$
5.2(b)	$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 0.838671 \\ -0.544639 \\ 0 \end{bmatrix}$	$\begin{bmatrix} 0 \\ 0 \\ 0.05233333 \end{bmatrix}$	$\begin{bmatrix} 0.02211699 \\ 0.03040064 \\ 0.08450596 \end{bmatrix}$

Table 5.1: Predictions of motion in computer-generated image pairs.

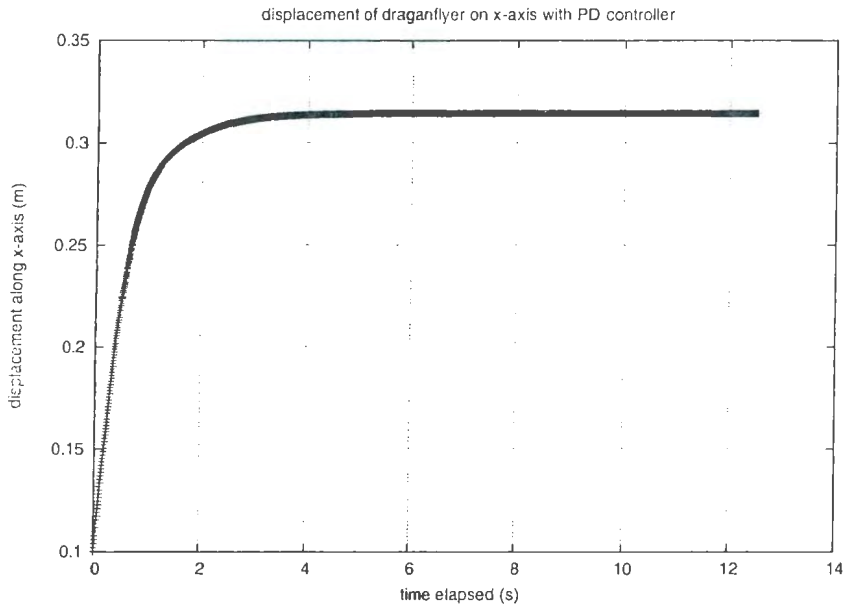


Figure 5.3: Effect of closed-loop control on displacement along the x-axis.

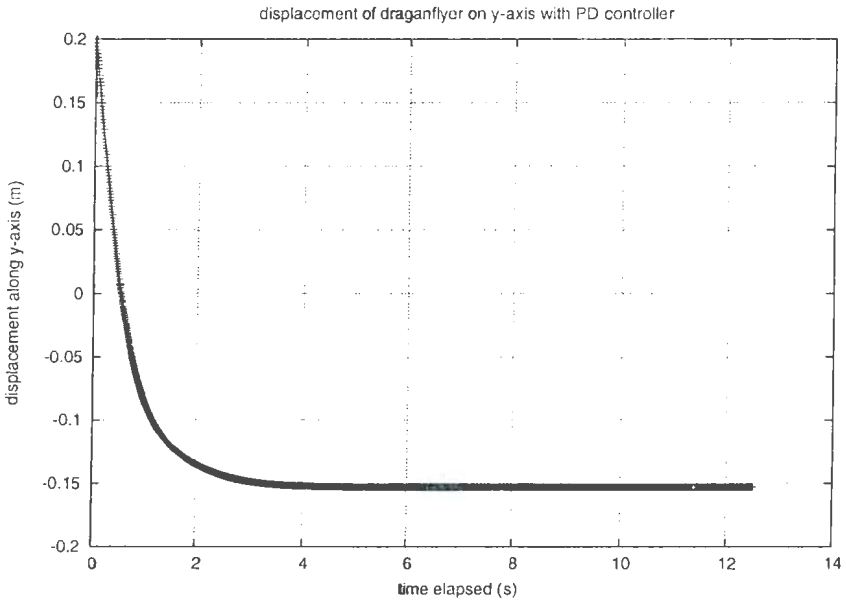


Figure 5.4: Effect of closed-loop control on displacement along the y-axis.

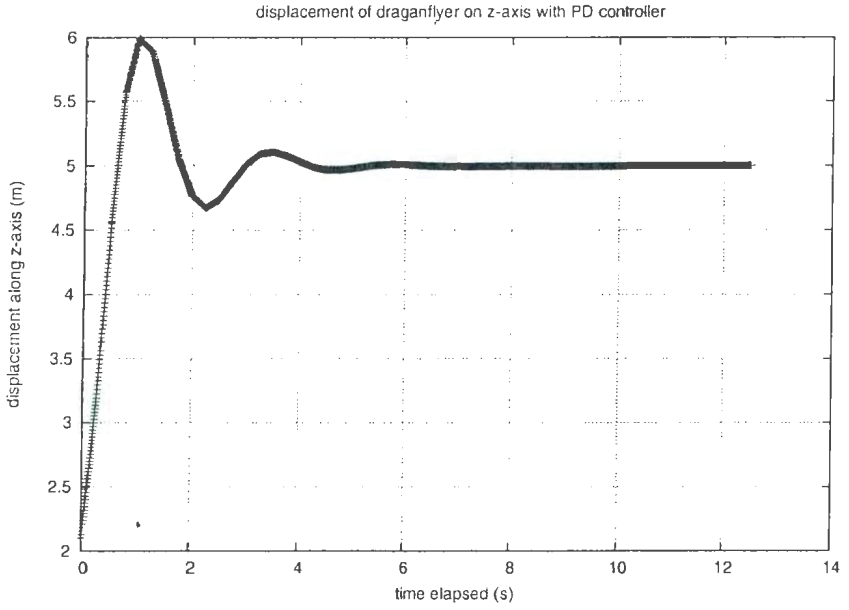


Figure 5.5: Effect of closed-loop control on displacement along the z-axis.

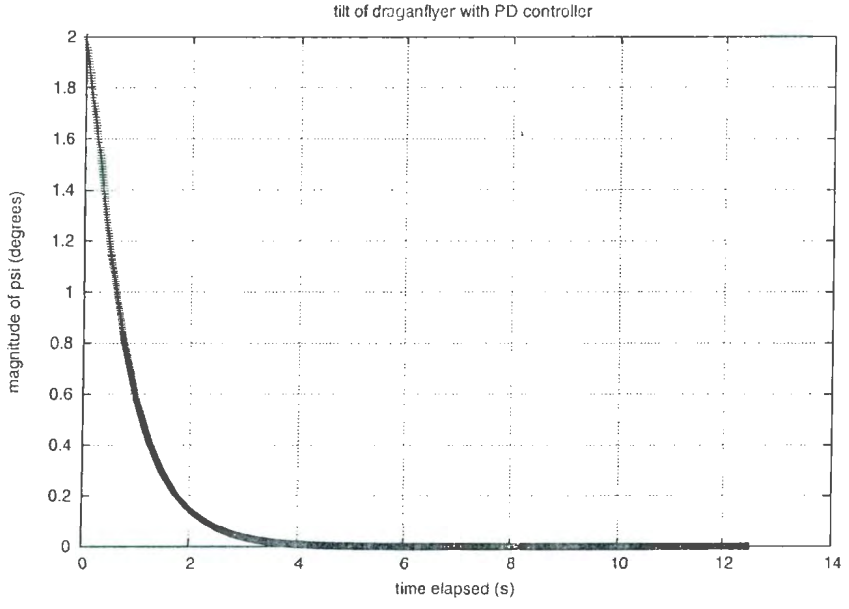


Figure 5.6: Effect of closed-loop control on ψ rotation.

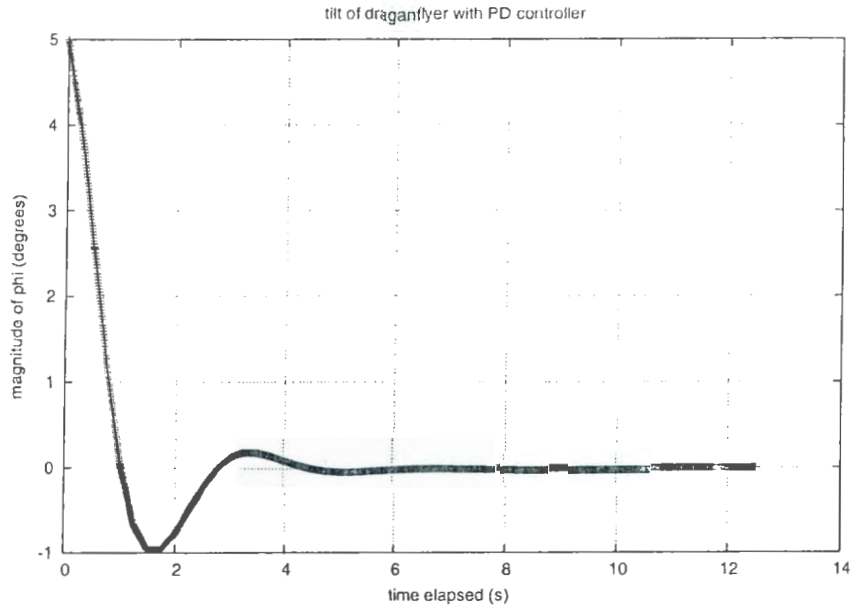


Figure 5.7: Effect of closed-loop control on ϕ rotation.

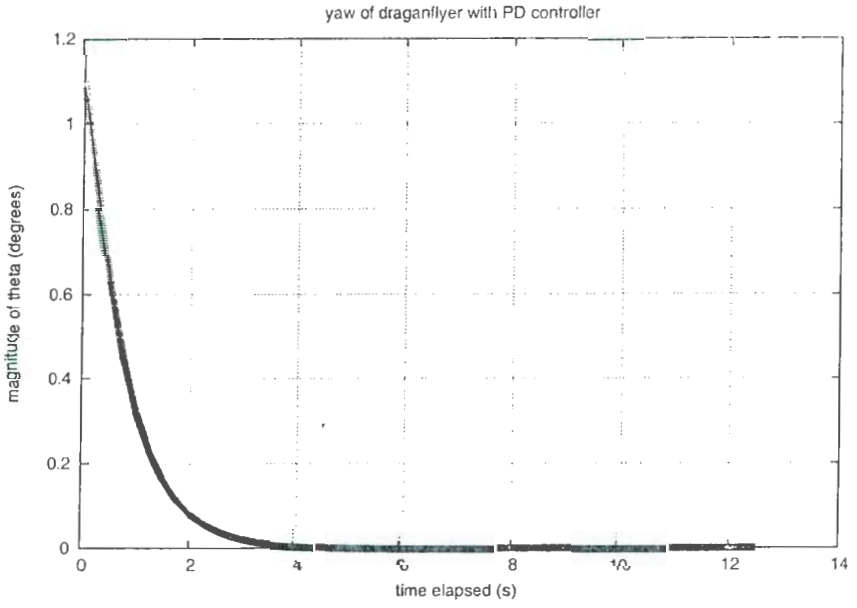


Figure 5.8: Effect of closed-loop control on θ rotation.

figure	transformed rotation vector	u_2	u_3	u_4
5.1(b)	$\begin{bmatrix} -1.84010 \\ -0.144351 \\ 0.638874 \end{bmatrix}$	5.52029	0.433053	-0.511099
5.2(a)	$\begin{bmatrix} -1.03578 \\ -1.52552 \\ 0.964723 \end{bmatrix}$	3.10733	4.57655	-0.771779
5.2(b)	$\begin{bmatrix} -4.84184 \\ -1.26721 \\ 1.74183 \end{bmatrix}$	14.5255	3.80163	-1.39346

Table 5.2: The outputs of the PD controllers in response to the flow fields. Note that u_1 is not included as it controls positioning on the z-axis, and magnitude of translation cannot be extracted from an image sequence.

Chapter 6

Conclusions

6.1 Findings

The egomotion estimates presented in section 3 are generally of a high degree of accuracy. This is due, at least in part, to the high-quality flow fields used as input to the estimator. When the estimator gives suboptimal results, it is normally due to either excessive outliers in the flow field, or ambiguous flow fields.

One previously mentioned type of ambiguity is the aperture problem, as explained in section 2.3. Normally when one speaks of ambiguous flow fields, they are referring to a flow field where it is indistinguishable whether the perceived motion was induced by a translation or by a rotation. Motions that are parallel to the image plane (ie: motions on the x and y-axes), as explained in [24], can cause confusion in the observer as to whether a rotation or a translation has occurred. In certain circumstances, a rotation about an axis in the negative direction will appear similar to a translation along the other axis of the image plane. These ambiguous motions, although not the

only ones, can be seen in Table 6.1.

axis of rotation	rotation direction	axis of translation	translation direction
x	negative	y	positive
x	positive	y	negative
y	negative	x	positive
y	positive	x	negative

Table 6.1: Ambiguous motion

The degree of robustness of an estimation is dictated by the size of the field of view (the larger the better), and the ratio of the magnitude of the translation to the distance from tracked features (again, the larger the better) [24]. In other words, the effects of translation are usually inversely proportional to the distance of the camera from the scene [4]. When a human tries to interpret the camera motion from the flow field in figure 6.1(a), it is unlikely that he/she will be able to tell that the motion is the result of a pure rotation, it is more likely that it would be characterized in the same way as 6.1(b), as a pure translation on the x-axis. Egomotion estimation algorithms often suffer from the same shortcoming. In fact, the further the camera from a scene on the image plane of a rotating camera, the more like a translation it will appear. The exception to this rule is rotation about the z-axis, as seen in figure 6.1. A viewer would easily be able to guess the motion that the camera is undergoing at the time of the capture. Likewise, estimations of translation along the z-axis tend to be the best of the three axes.

This problem is further exacerbated by the fact that these ambiguous types of

(a) Rotation about the y-axis.

(b) Translation on the x-axis.

Figure 6.1: Two frames from different sequences demonstrating how rotational and translational motion can be ambiguous.

motions can occur concurrently. For instance, the camera may be translating along the x-axis while simultaneously rotating about the negative y-axis. In such a circumstance, it becomes even more difficult to infer what type of motion is truly causing the degree of the displacement. It is clear that the camera is moving the in the positive x direction, but the degree of the magnitude of the flow that may be attributed is indeed ambiguous. After all, only the translation *direction* has been obtained, a unit vector that may represent a translational magnitude that can be anything from

Figure 6.2: Rotation about the z-axis.

extremely small to very large.

Likewise, say that the camera has again translated along the positive x-axis, but it has rotated about the positive y-axis. In the circumstance where the magnitudes of the translational flow vectors are similar to those of the rotational flow vectors, it will appear that there has been nearly no movement. This is contingent on the distances of the features from the camera lens.

6.2 Final thoughts and future work

The modern INS is both affordable and reliable, and when coupled with a GPS it makes for a better estimator of 3d motion than one could ever hope for in a camera-based solution. Furthermore, in a single camera-based solution, it is only possible to extract the direction of translation (a unit vector), not its magnitude. Depth and rotation can be obtained, but some other sensor is required if magnitude of translation is required.

However, that does not mean that the ego-estimation problem is unimportant. If

one wishes to determine the distance of an approaching object without resorting to radar or some other active sensor, then rigid motion must be solved for in order to extract depth. Furthermore, vision-based navigation has been a very useful tool in simultaneous localization and mapping (SLAM). SLAM deals with the problem of building a map of an unknown environment while navigating the environment with said map [25].

The implemented estimator presently does very little to prune statistical outliers. If a flow vector exceeds the maximal prespecified length, then it is not used in the $3d$ motion calculations. In order to obtain a more robust solution, there will be more effort put into using estimates based on prior motion to eliminate outliers. For example, if a Kalman filter indicates that the camera should moving in direction $\hat{t}$ with a rotation of $\hat{\omega}$, flow vectors that vary from the expected values by more than a predefined threshold will be pruned. Furthermore, SIFT (Scale-Invariant Feature Transform) will be used instead of Shi and Tomasi's [26] method for identifying features, as the former is generally unaffected by changes in illumination, rotation and scale [27].

In the preceding thesis, all real-world images were captured using a standard 3.2 megapixel camera. It is not so much the quality of the sensor that may have caused issues so much as the consumer-grade lens. Indeed there is some barrel distortion that must be corrected for, and in future a calibrated camera will be used exclusively. It was not realized at the outset how ambiguous optical flow data really can be, and how subtle changes in the field will completely throw off motion estimates. So not only is it important to use a better feature tracker for avoiding excessive outliers, it is also important to have subpixel accuracy of matched features, which was not completed

here. Essentially, the implemented estimator gives a fairly good estimation of heading and rotation at times, but the results can be highly noisy and at times even wrong (due to a lack of convergence in Levenberg-Marquardt). The problems faced during experimentation due high levels of inaccuracy were extremely frustrating, as it was presumed that there was a problem with the estimator. However, the literature suggests that such is the nature of optical flow applications. Thus, it would be interesting to characterize the type of motion and focus on, for example, the motion of features that are closer to the camera, which will tell more about the nature of the motion. Likewise, more weight could be put on certain areas that characterize translation versus rotation better.

It is a good idea to generate ideal flow fields for a certain type of rigid motion. Even when working with synthetic images, there are still too many unknowns and inaccuracies to be sure whether or not an the translational or rotational estimators are performing properly. Adding noise gradually and subsequently working with synthetic images makes much more sense.

Finally, Bruss and Horn's egomotion estimation method will be utilized over Heeger and Jepson's. By utilizing a nonlinear equation solver, this paper has essentially emulated the same method, but with $n - 6$ constraints instead of n. Furthermore, it is much slower because the SVD has to be performed each time one finds the orthogonal complement of the column space of a matrix, and also because the Jacobian has to be approximated. In other words, the implementation will become much faster and more accurate, the universal metrics toward which all Engineers work.

Bibliography

[1] J.P. Ostrowski E. Altug and Robert Mahony, "Control of a quadrotor helicopter using visual feedback", *Proceedings of the 2002 IEEE International Conference on Robotics and Automation*, pp. 72–77, 2002.

[2] D.J. Heeger and A.D. Jepson, "Subspace methods for recovering rigid motion i: algorithm and implementation", *Int. J. Comput. Vision*, vol. 7, no. 2, pp. 95–117, 1992.

[3] A.R. Bruss and B.K.P. Horn, "Passive navigation", *Computer Vision, Graphics and Image Processing*, vol. 21, no. 1, pp. 3–20, 1983.

[4] C. Tomasi and J. Shi, "Direction of heading from image deformations", in *CVPR93*, 1993, pp. 422–427.

[5] T. Tian, C. Tomasi, and D. Heeger, "Comparison of approaches to egomotion computation", 1996.

[6] B.K. Horn and B.G. Schunck, "Determining optical flow", pp. 389 407, 1992.

[7] S.M. Smith, "Reviews of optic flow, motion segmentation, edge finding and corner finding", Tech. Rep., Oxford University, 1997.

[8] M.I.A. Lourakis, "levmar: Levenberg-marquardt nonlinear least squares algorithms in C/C++", [web page] http://www.ics.forth.gr/ lourakis/levmar, Jul. 2004, [Accessed on Jul. 24, 2007].

[9] D. Stavens, "Introduction to opencv", [web page] http://ai.stanford.edu/ dstavens/cs223b, Jan. 2005, [Accessed on Jul. 24, 2007].

[10] I.T. Young, J.J. Gerbrands, and L.J. van Vliet, "Image processing fundamentals".

[11] A. Fusiello, E. Trucco, T. Tommasini, and V. Roberto, "Improving feature tracking with robust statistics", *Pattern Analysis and Applications*, vol. 2, pp. 312–320, 1999.

[12] E.C. Hildreth, "Computations underlying the measurement of visual motion", *Artificial Intelligence*, vol. 23, pp. 309–354, 1984.

[13] J. Bouguet, "Pyramidal implementation of the Lucas Kanade feature tracker", 2000.

[14] K. Prazdny, "On the information in optical flows.", *Computer Vision, Graphics, and Image Processing*, vol. 22, no. 2, pp. 239–259, 1983.

[15] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins Press, Baltimore, MD, USA, second edition, 1989.

[16] J.S. Hourigan and L.V. McIndoo, "The singular value decomposition introduction", [web page]

http://online.redwoods.cc.ca.us/instruct/darnold/laproj/Fall98/JodLynn/report2.pdf, Accessed on Nov. 3, 2007.

[17] T. Zhang and C. Tomasi, "Fast, robust, and consistent camera motion estimation.", in *CVPR*. 1999, pp. 1164–1170, IEEE Computer Society.

[18] A. Tayebian and S. McGilvray, "Attitude stabilization of a vtol quadrotor aircraft", *IEEE Transactions on Control Systems Technology*, vol. 14, pp. 562 571, 2006.

[19] M. Chen and Mihai Huzmezan, "A simulation model and hinf loop shaping control of a quadrotor unmanned air vehicle.", in *Modelling, Simulation, and Optimization*, M. H. Hamza, Ed. 2003, pp. 320–325, IASTED/ACTA Press.

[20] J.P. Ostrowski E. Altug and C.J. Taylor, "Control of a quadrotor helicopter using dual camera visual feedback", *Int. J. Rob. Res.*, vol. 24, no. 5, pp. 329–341, 2005.

[21] T. Wescott, "Pid without a phd", *Embedded Systems Programming*, October 2000, http://www.embedded.com/2000/0010/0010feat3.htm.

[22] Persistence of Vision Pty. Ltd., "Persistence of vision raytracer (version 3.6)", [software] http://www.povray.org/download/, 2004, Accessed on Nov. 10, 2007.

[23] "Matlab function reference", [web page] http://www.mathworks.com/access/helpdesk/help/techdoc/ref/ode113.html, [Accessed on Nov. 6, 2007].

[24] K. Daniilidis and H.-H. Nagel, "The coupling of rotation and translation in motion estimation of planar surfaces", in *it IEEE Conf. on Computer Vision and Pattern Recognition 1993*, New York, NY, June 15-17, 1993, pp. 188–193.

[25] S. Riisgaard and M.R. Blas, "Slam for dummies: a tutorial to simultaneous localization and mapping", [web page] http://ocw.mit.edu/OcwWeb/Aeronautics-and-Astronautics/16-412JSpring-2005/Projects, 2005, [Accessed on Nov. 3, 2007].

[26] J. Shi and C. Tomasi, "Good features to track", in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR '94)*, Seattle, Jun, 1994.

[27] D. Lowe, "Distinctive image features from scale-invariant keypoints", in *International Journal of Computer Vision*, 2003, vol. 20, pp. 91–110.

Appendices

Appendix A

Egomotion Estimation Code

A.1 Camera3DState.cpp

```
#include <cv.h>
#include "Camera3DState.h"
/**
 * class: Camera3DState
 * brief: Responsible for storing the current rotational displacement of
 * the camera in 3d space.
 */
Camera3DState::Camera3DState( ) {
   r_x = 0.0;
   r_y = 0.0;
   r_z = 0.0;
}
/**
 * A 3x1 opencv matrix is inputted instead for the update
```

```
 */
void Camera3DState::updateRotation( CvMat* R ) {

   r_x = r_x + cvmGet( R, 0, 0 );
   r_y = r_y + cvmGet( R, 1, 0 );
   r_z = r_z + cvmGet( R, 2, 0 );
}

/**
  * It is simply assumed that as long as p and info are not null
  * that they are indeed of the correct dimensions
  * pre: p has 3 elements and info has 6
  * post: the per-frame displacements have been added
  */
void Camera3DState::updateRotation( double* p ) { //, double* info ) {
     r_x = r_x + p[0];
     r_y = r_y + p[1];
     r_z = r_z + p[2];
}
/**
 * The cumulative rotation about the x-axis from t=0 to the current time
 */
double Camera3DState::getRotationAboutXAxis( ) {
  return r_x;
}
/**
 * The cumulative rotation about the y-axis from t=0 to the current time
 */
```

```
double Camera3DState::getRotationAboutYAxis( ) {
   return r_y;
}
/**
 * The cumulative rotation about the z-axis from t=0 to the current time
 */
double Camera3DState::getRotationAboutZAxis( ) {
   return r_z;
}
```

A.2 CMatrixBuilder.cpp

```
#include "CMatrixBuilder.h"

CMatrixBuilder::CMatrixBuilder( CvPoint2D32f* FrameOneFeatures, CvPoint2D32f* FrameTwoFeatures,
      char* FoundFeaturesMap, int NumPoints ) {

  f = 1.0;//634.0;
  frameOneFeatures = FrameOneFeatures;
  frameTwoFeatures = FrameTwoFeatures;
  foundFeaturesMap = FoundFeaturesMap;
  numPoints = NumPoints;

  // calculate the number of found points
  int i = 0;
  numFeaturesProcessed = 0;
  while( i < numPoints ) {
    if( foundFeaturesMap[i] == 1 ) {
```

```
            numFeaturesProcessed++;
        }
        i++;
    }

    allocateMatrices( );
}

CMatrixBuilder::~CMatrixBuilder( ) {
    cvReleaseMat( &A_T );
    cvReleaseMat( &B );
    cvReleaseMat( &C );
}
CvMat* CMatrixBuilder::getC( ) {
    return C;
}
/*
 * Returns the number of rows of C
 */
int CMatrixBuilder::getCRowSize( ) {
    return 2*numFeaturesProcessed;
}
/*
 * Return the number of columns of C
 */
int CMatrixBuilder::getCColumnSize( ) {
    return numFeaturesProcessed+3;
}
```

```
void CMatrixBuilder::allocateMatrices( ) {
   // make matrices of the correct dimensions
   A_T = cvCreateMat( 2*numFeaturesProcessed, numFeaturesProcessed, CV_32F );
   B = cvCreateMat( 2*numFeaturesProcessed, 3, CV_32F );
   C = cvCreateMat( 2*numFeaturesProcessed,numFeaturesProcessed+3,CV_32F );
   //Vt = cvCreateMat(1, 2*numFeaturesProcessed, CV_32F );

   // initialize all elements of A(T) to 0
   for( int i=0; i<2*numFeaturesProcessed; i++ ) {
      for( int j=0; j<numFeaturesProcessed; j++ ) {
         CV_MAT_ELEM( *A_T, float, i, j ) = 0;
      }
   }
}

void CMatrixBuilder::fillA_T( CvMat* T ) {
      CvPoint2D32f* p1 = frameOneFeatures;

      // initialize A_T and all of its static elements
   CvMat* A_xy = cvCreateMat( 2,3, CV_32F );
   /*
    * changed on May 8th to negative f
    */
   CV_MAT_ELEM( *A_xy, float, 0, 0 ) = -f;
   CV_MAT_ELEM( *A_xy, float, 0, 1 ) = 0;
   CV_MAT_ELEM( *A_xy, float, 1, 0 ) = 0;
   CV_MAT_ELEM( *A_xy, float, 1, 1 ) = -f;
```

```
int rowIndex = 0;
numFeaturesProcessed = 0;
int i=0;

CvMat* res = cvCreateMat( 2, 1, CV_32F ); // res is temporary storage for what goes into A(T)
while( i < numPoints ) {
  if( foundFeaturesMap[i] == 1 ) { //opticalFlowFoundFeature[i] == 1 ) {
    CV_MAT_ELEM( *A_xy, float, 0, 2 ) = p1[i].x;
    CV_MAT_ELEM( *A_xy, float, 1, 2 ) = p1[i].y;

    cvmMul( A_xy,T,res ); //ewknflaw

    CV_MAT_ELEM( *A_T, float, rowIndex, numFeaturesProcessed ) =
        cvmGet( res, 0, 0 );
    CV_MAT_ELEM( *A_T, float, rowIndex+1,numFeaturesProcessed ) =
        cvmGet( res, 1, 0 );

    rowIndex = rowIndex+2;
    numFeaturesProcessed++;
  }
  i++;
  }
  cvReleaseMat( &res );
  cvReleaseMat( &A_xy );
}
```

```
void CMatrixBuilder::fillB( ) { //CvPoint2D32f* p1 ) {
        CvPoint2D32f* p1 = frameOneFeatures;

    // initialize B and its static elements - the x & y's will be
    // updated in the loop
    int i=0;
    int rowIndex = 0;
    float x;
    float y;
    while( i < numFeaturesProcessed ) {                                    110
                x = p1[i].x;
                y = p1[i].y;
                // fill all columns of the first row
      // THIS WAS CHANGED TO X*Y ON MAY 8TH
      CV_MAT_ELEM( *B, float, rowIndex, 0 ) = ( x*y ) / f;
      CV_MAT_ELEM( *B, float, rowIndex, 1 ) = -1* ( f + ( x*x / f ) );
      CV_MAT_ELEM( *B, float, rowIndex, 2 ) = y;
      // fill all columns of second row
      CV_MAT_ELEM( *B, float, rowIndex+1, 0 ) = f + ( y*y / f );
      CV_MAT_ELEM( *B, float, rowIndex+1, 1 ) = -1 * ( x*y / f);           120
      CV_MAT_ELEM( *B, float, rowIndex+1, 2 ) = -1 * x;

      // update indices
      i++;
      rowIndex += 2;
    }
}
```

```
/**
 * FillC( )                                                          130
 * The columns of C are made up of A_T and B
 */
void CMatrixBuilder::fillC( ) {
        // fill the left side of C with elements of A
   for( int i=0; i<2*numFeaturesProcessed; i++ ) {
      for( int j=0; j<numFeaturesProcessed; j++ ) {
         CV_MAT_ELEM( *C, float, i, j ) = cvmGet( A_T, i, j );
      }
   }
   // fill the right side of C with elements of B                    140
   for( int i=0; i<2*numFeaturesProcessed; i++ ) {
      for( int j=0; j<3; j++ ) {
         // B is offset exactly the width of A in C
         CV_MAT_ELEM( *C, float, i, numFeaturesProcessed+j ) = cvmGet( B, i, j );
      }
   }
}
```

sectionDataStruct.h

```
#include <cv.h>
#ifndef DATASTRUCT_H
#define DATASTRUCT_H
/**
 * This class stores optical flow data and feature points for
 * calculation of egomotion estimation.
 *
```

```
   * param: T When passed to the translation estimator, it
   * is an intial guess of the translation.  When passed to
   * the rotation estimator, it is the returned estimate from
   * the nonlinear solver.
   * param: R When passed to the rotation estimator, it is an
   * initial guess of the rotation.  Afterwards, it is a
   * refined estimate, suitable for passing to a controller.
   */
struct DataStruct {
   CvPoint2D32f* frameOneFeatures;
   CvPoint2D32f* frameTwoFeatures;
   char* featuresMap;
   int numPoints;
   int numFeaturesProcessed;
   CvMat* T;
   CvMat* R;
};
#endif
```

A.3 LeastSquares.cpp

```
#include "LeastSquares.h"
#include <iostream>
#include <math.h>

/**
 * The initial guess for T is already inside of cPerp,
 * although it is included as an argument because
```

```
 * the residual must be subtracted each time and cPerp
 * must be rebuilt.
 */
LeastSquares::LeastSquares( ) {

}
/**
 * param: M A 1xnumCols vector for which we must
 * find the norm
 */
double LeastSquares::evaluateColVecL2Norm( CvMat* M, int numCols ) {
   double sumOfSquares = 0;
   for( int i=0; i<numCols; i++ ) {
      // add the square of the current element
      sumOfSquares += ( cvmGet( M, 0, i ) * cvmGet( M, 0, i ) );
   }
   return sqrt( sumOfSquares );
}
/**
 * param: M a numRows x 1 vectors for which we must find the norm
 */
double LeastSquares::evaluateRowVecL2Norm( CvMat* M, int numRows ) {
   double sumOfSquares = 0;
   for( int i=0; i<numRows; i++ ) {
      sumOfSquares += ( cvmGet( M, i, 0 ) * cvmGet( M, i, 0 ) );
   }
   return sqrt( sumOfSquares );
}
```

A.4 NumberConverter.cpp

```
#include <string>
#include <iostream>
#include <math.h>
#include "NumberConverter.h"

/**
 * Will turn the inputted vector into one of unit length
 */
void NumberConverter::normalize3dVector( CvMat* vec ) {
   double len = get3dVectorLength( vec );
   CV_MAT_ELEM( *vec, float, 0, 0 ) = cvmGet( vec, 0, 0 ) / len;
   CV_MAT_ELEM( *vec, float, 1, 0 ) = cvmGet( vec, 1, 0 ) / len;
   CV_MAT_ELEM( *vec, float, 2, 0 ) = cvmGet( vec, 2, 0 ) / len;
}

double NumberConverter::get3dVectorLength( CvMat* vec ) {
   double x = cvmGet( vec, 0, 0 );
   double y = cvmGet( vec, 1, 0 );
   double z = cvmGet( vec, 2, 0 );

   double len = sqrt( (x*x) + (y*y) + (z*z) );

   return len;
}
```

```
double NumberConverter::degreesToRadians( double degrees ) {
  return degrees * ( 3.14159265 / 180.0 );
}

double NumberConverter::radiansToDegrees( double radians ) {
  return radians * ( 180.0 / 3.14159265 );
}
/**
 * Both T and Tout must be preallocated 3x1 CvMats.
 * param: tOut tOut[0][0] = rho, tOut[1][0] = phi, tOut[2][0] = theta
 */
void NumberConverter::cartesianToSpherical( CvMat* T, CvMat* tOut ) {
  assert( T != NULL );
  assert( tOut != NULL );
  double x = cvmGet( T, 0, 0 );
  double y = cvmGet( T, 1, 0 );
  double z = cvmGet( T, 2, 0 );

  // convert cartesian coords to spherical ones
  double S = sqrt( (x*x) + (y*y ) );
  double rho = sqrt( (x*x) + (y*y) + (z*z) );
  if( (rho >= 1.01 ) || (rho <= -1.01 ) ) {
    std::cerr << "rho calculation is messed up" << std::endl;
  }
  double phi = radiansToDegrees( acos( z ) ); // actually z / rho, but rho = 1.0
  double theta;
  if( x >= 0.0 ) {
    theta = radiansToDegrees( asin( y / S ) );
```

```
    }
    else {
      theta = radiansToDegrees( 3.14159265 - asin( y / S ) );
    }
    CV_MAT_ELEM( *tOut, float, 0, 0 ) = rho;
    CV_MAT_ELEM( *tOut, float, 1, 0 ) = phi;
    CV_MAT_ELEM( *tOut, float, 2, 0 ) = theta;                    60
}
/**
 * Both must be previously-initialized 3x1 matrices
 * param: T T[0][0] = rho, T[1][0] = phi, T[2][0] = theta
 * param: tOut tOut[0][0] = x, tOut[1][0] = y, tOut[2][0] = z
 */
void NumberConverter::sphericalToCartesian( CvMat* T, CvMat* tOut ) {
    double rho = cvmGet( T, 0, 0 );
    double phi = cvmGet( T, 1, 0 );
    double theta = cvmGet( T, 2, 0 );                             70

    sphericalToCartesian( rho, phi, theta, tOut );
}
void NumberConverter::sphericalToCartesian( double rho, double phi, double theta, CvMat* tOut ) {

    CV_MAT_ELEM( *tOut, float, 0, 0 ) =
        rho * sin( degreesToRadians( phi ) ) * cos( degreesToRadians( theta ) ); // x
    CV_MAT_ELEM( *tOut, float, 1, 0 ) =
        rho * sin( degreesToRadians( phi ) ) * sin( degreesToRadians( theta ) ); // y
    CV_MAT_ELEM( *tOut, float, 2, 0 ) =                           80
        rho * cos( degreesToRadians( phi ) ); // z
```

```
}

/**
 * Taken/modified from the CodeProject article entitled "Reliable Floating Point Equality Comparison".
 * http://www.codeproject.com/tips/FloatingPointEquality.asp
 */
bool NumberConverter::AlmostEqual(double nVal1, double nVal2, double nEpsilon=0.000001 ) {
   bool bRet = (((nVal2 - nEpsilon) < nVal1) && (nVal1 < (nVal2 + nEpsilon)));
   return bRet;
}
/**
 *
 * param: nEpsilon Want the elements of m1 and m2 to be equal to at least this level
 * of precision.  For example, if you want elements to be equal to 3 decimal places,
 * input nEpsilon = 0.0001
 */
bool NumberConverter::AlmostEqual( CvMat* m1, CvMat* m2, double nEpsilon=0.000001 ) {
   CvSize m1Size = cvGetSize( m1 );
   CvSize m2Size = cvGetSize( m2 );

   if( m1Size.width != m2Size.width ) {
      return false;
   }
   if( m1Size.height != m2Size.height ) {
      return false;
   }
   // have determined that they're the same size, so
```

```
    // now let's cycle through the elements and make sure that
    // they're equal
    for( int i=0; i<m1Size.height; i++ ) {
      for( int j=0; j<m1Size.width; j++ ) {
        if( !AlmostEqual( cvmGet( m1, i, j ), cvmGet( m2, i, j ), nEpsilon ) ) {
          return false;
        }
      }
    }
    // all elements have evaluated true when tested with AlmostEquals
    return true;
}
```

A.5 FlowMain.cpp

```
/*
 * FlowMain.cpp
 * Modified version of David Stavens' optical flow generator source code.
 */
#include <iostream>
#include <stdio.h>
#include <cv.h>
#include <highgui.h>
#include <math.h>
#include "DataStruct.h"
#include "RotationEstimator.h"
#include "TranslationEstimator.h"
#include "Camera3DState.h"
```

```
#include "NumberConverter.h"
static const double pi = 3.14159265358979323846;

inline static double square(int a){
    return a * a;
}

inline static void allocateOnDemand( IplImage **img, CvSize size, int depth, int channels ) {
    if ( *img != NULL ) return;

    *img = cvCreateImage( size, depth, channels );
    if ( *img == NULL ) {
        fprintf(stderr, "Error: Couldn't allocate image. Out of memory?\n");
        exit(-1);
    }
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "usage: %s image1.jpg image2.jpg\n", argv[0]);
        return -1;
    }
    /* Create an object that decodes the input video stream. */
    IplImage *frame = cvLoadImage( argv[1], 1 );
    /* Read the video's frame size out of the AVI. */
    CvSize frame_size = cvGetSize( frame );

    /* Create a windows called "Optical Flow" for visualizing the output.
```

```
 * Have the window automatically change its size to match the output.
 */
cvNamedWindow("Optical Flow", CV_WINDOW_AUTOSIZE);
DataStruct egomotionEstData;
CvMat* transGuess = cvCreateMat( 3, 1, CV_32F );
CV_MAT_ELEM( *transGuess, float, 0, 0 ) = 0.98; // translationEst[0];
CV_MAT_ELEM( *transGuess, float, 1, 0 ) = 0.1; // translationEst[1];
CV_MAT_ELEM( *transGuess, float, 2, 0 ) = 0.1; // translationEst[2];

CvMat* rotGuess = cvCreateMat( 3, 1, CV_32F );
CV_MAT_ELEM( *rotGuess, float, 0, 0 ) = 0.01;
CV_MAT_ELEM( *rotGuess, float, 1, 0 ) = 0.01;
CV_MAT_ELEM( *rotGuess, float, 2, 0 ) = 0.01;
egomotionEstData.T = transGuess;
egomotionEstData.R = rotGuess;

static IplImage *frame1 = NULL, *frame1_1C = NULL, *frame2_1C = NULL, *eig_image = NULL,
*temp_image = NULL, *pyramid1 = NULL, *pyramid2 = NULL;
allocateOnDemand( &frame1_1C, frame_size, IPL_DEPTH_8U, 1 );
cvConvertImage(frame, frame1_1C, 0);

/* We'll make a full color backup of this frame so that we can draw on it.
 * (It's not the best idea to draw on the static memory space of cvQueryFrame().)
 */

allocateOnDemand( &frame1, frame_size, IPL_DEPTH_8U, 3 );
cvConvertImage(frame, frame1, 0);
```

```
/* Get the second frame of video. Same principles as the first. Note that
 * this frame is saved and used in the next iteration of the loop as well.
 */

frame = cvLoadImage( argv[2], 1 );
//frame = cvQueryFrame( input_video );

if (frame == NULL) {
  return 0;
}
allocateOnDemand( &frame2_1C, frame_size, IPL_DEPTH_8U, 1 );
cvConvertImage(frame, frame2_1C, 0);

/* Shi and Tomasi Feature Tracking! */
/* Preparation: Allocate the necessary storage. */
allocateOnDemand( &eig_image, frame_size, IPL_DEPTH_32F, 1 );
allocateOnDemand( &temp_image, frame_size, IPL_DEPTH_32F, 1 );
/* Preparation: This array will contain the features found in frame 1. */
CvPoint2D32f frame1_features[400];
/* Preparation: BEFORE the function call this variable is the array size
 * (or the maximum number of features to find). AFTER the function call
 * this variable is the number of features actually found.
 */
int number_of_features;
/* I'm hardcoding this at 400. But you should make this a #define so that you can
 * change the number of features you use for an accuracy/speed tradeoff analysis.
 */
number_of_features = 400;
```

```
/* Actually run the Shi and Tomasi algorithm!!
 * "frame1_1C" is the input image.
 * "eig_image" and "temp_image" are just workspace for the algorithm.
 * The first ".01" specifies the minimum quality of the features (based on the eigenvalues).
 * The second ".01" specifies the minimum Euclidean distance between features.
 * "NULL" means use the entire input image. You could point to a part of the image.
 * WHEN THE ALGORITHM RETURNS:
 * "frame1_features" will contain the feature points.
 * "number_of_features" will be set to a value <= 400 indicating the number of feature points found.
 */
cvGoodFeaturesToTrack(frame1_1C, eig_image, temp_image,
    frame1_features, &number_of_features, .01, 40 );

/* Pyramidal Lucas Kanade Optical Flow! */

/* This array will contain the locations of the points from frame 1 in frame 2. */
CvPoint2D32f frame2_features[400];

/* The i-th element of this array will be non-zero if and only if the i-th feature of
 * frame 1 was found in frame 2.
 */
char optical_flow_found_feature[400];

/* The i-th element of this array is the error in the optical flow for the i-th feature
 * of frame1 as found in frame 2. If the i-th feature was not found (see the array above)
 * I think the i-th entry in this array is undefined.
 */
float optical_flow_feature_error[400];
```

```
/* This is the window size to use to avoid the aperture problem (see slide "Optical Flow: Overview"). */
CvSize optical_flow_window = cvSize(3,3);

/* This termination criteria tells the algorithm to stop when it has either done 20 iterations or when
 * epsilon is better than .3.  You can play with these parameters for speed vs. accuracy but these values
 * work pretty well in many situations.
 */
CvTermCriteria optical_flow_termination_criteria
   = cvTermCriteria( CV_TERMCRIT_ITER | CV_TERMCRIT_EPS, 20, .3 );
allocateOnDemand( &pyramid1, frame_size, IPL_DEPTH_8U, 1 );
allocateOnDemand( &pyramid2, frame_size, IPL_DEPTH_8U, 1 );
cvCalcOpticalFlowPyrLK(frame1_1C, frame2_1C, pyramid1, pyramid2, frame1_features, frame2_features,
      number_of_features, optical_flow_window, 5, optical_flow_found_feature,
      optical_flow_feature_error, optical_flow_termination_criteria, 0 );

int numFeaturesProcessed = 0; // count the number of features processed
/* For fun (and debugging :)), let's draw the flow field. */
for(int i = 0; i < number_of_features; i++){
   /* If Pyramidal Lucas Kanade didn't really find the feature, skip it. */
   if ( optical_flow_found_feature[i] == 0 ) continue;
   int line_thickness;
   line_thickness = 1;
   /* CV_RGB(red, green, blue) is the red, green, and blue components
    * of the color you want, each out of 255.
    */
   CvScalar line_color;
   line_color = CV_RGB(255,0,0);
```

```
    CvPoint p,q;
    p.x = (int) frame1_features[i].x;
    p.y = (int) frame1_features[i].y;
    q.x = (int) frame2_features[i].x;
    q.y = (int) frame2_features[i].y;
    double angle;                                                    160
    angle = atan2( (double) p.y - q.y, (double) p.x - q.x );
    double hypotenuse;
    hypotenuse = sqrt( square(p.y - q.y) + square(p.x - q.x) );
    /* Here we lengthen the arrow by a factor of three. */
    q.x = (int) (p.x - 3 * hypotenuse * cos(angle));
    q.y = (int) (p.y - 3 * hypotenuse * sin(angle));
    cvLine( frame1, p, q, line_color, line_thickness, CV_AA, 0 );
    p.x = (int) (q.x + 9 * cos(angle + pi / 4));
    p.y = (int) (q.y + 9 * sin(angle + pi / 4));
    cvLine( frame1, p, q, line_color, line_thickness, CV_AA, 0 );    170
    p.x = (int) (q.x + 9 * cos(angle - pi / 4));
    p.y = (int) (q.y + 9 * sin(angle - pi / 4));
    cvLine( frame1, p, q, line_color, line_thickness, CV_AA, 0 );
    numFeaturesProcessed++;
}

/* Now display the image we drew on. Recall that "Optical Flow" is the name of
 * the window we created above.
 */
cvShowImage("Optical Flow", frame1);                                 180
/* And wait for the user to press a key (so the user has time to look at the image).
```

```
 * If the argument is 0 then it waits forever otherwise it waits that number of milliseconds.
 * The return value is the key the user pressed.
 */
int key_pressed;
key_pressed = cvWaitKey(0);

/*
 * Egomotion estimation!
 */
NumberConverter numConv;
egomotionEstData.frameOneFeatures = frame1_features;
egomotionEstData.frameTwoFeatures = frame2_features;
egomotionEstData.featuresMap = optical_flow_found_feature;
egomotionEstData.numPoints = number_of_features;
egomotionEstData.numFeaturesProcessed = numFeaturesProcessed;
TranslationEstimator transEstimator;

// iterative solver
transEstimator.getTranslation( &egomotionEstData );
std::cout << "iterative solution: " << cvmGet( egomotionEstData.T, 0, 0 )
   << " " << cvmGet( egomotionEstData.T, 1, 0 ) << " "
   << cvmGet( egomotionEstData.T, 2, 0 ) << std::endl;
// nonlinear solver
transEstimator.estimateTranslation( &egomotionEstData );
numConv.normalize3dVector( egomotionEstData.T );
std::cout << "nonlinear solver normalized trans: " <<
   cvmGet( egomotionEstData.T, 0, 0 ) << " "
   << cvmGet( egomotionEstData.T, 1, 0 ) <<
```

```
        " " << cvmGet( egomotionEstData.T, 2, 0 ) << std::endl;

    RotationEstimator rotationEstimator;
    rotationEstimator.estimateRotation( &egomotionEstData );
}
```

A.6 OrthogonalComplement.cpp

```
#include "OrthogonalComplement.h"

using namespace std;
OrthogonalComplement::~OrthogonalComplement( ) {
    cvReleaseMat( &Ut );
    cvReleaseMat( &W );
    cvReleaseMat( &Vt );
    cvReleaseMat( &cPerp );
}
OrthogonalComplement::OrthogonalComplement( CvMat* M, int r, int c ) {
    rows = r;
    cols = c;
    C = M;
    Ut = cvCreateMat( rows, rows, CV_32F );
    W = cvCreateMat( rows, cols, CV_32F );
    Vt = cvCreateMat( cols, cols, CV_32F );
    cPerp = cvCreateMat( rows, rows-cols, CV_32F );
}
void OrthogonalComplement::evalSVDecomposition( ) {
    //ofstream myfile;
```

```
    //myfile.open( "SVD.txt" );
    //myfile << "Writing this to a file.\n";
    cvSVD( C, W, Ut, Vt, CV_SVD_U_T|CV_SVD_V_T );
    // we will avoid the data logging for now, because
    // it is far too time-consuming
    /*for( int i=0; i<rows; i++ ) {
      for( int j=0; j<cols; j++ ) {
        myfile << " " << cvmGet( W, i, j ) << " ";
      }
      myfile << std::endl;
    }
    myfile.close( );*/
}
CvMat* OrthogonalComplement::getOrthogonalComplement( ) {
    // we have to transpose U^t first (it is stored as U)
    CvMat* U = cvCreateMat( rows, rows, CV_32F );
    // transpose Ut -> U
    cvTranspose( Ut, U );
    for( int i=0; i<rows; i++ ) {
      for( int j=0; j<(rows-cols); j++ ) {
        CV_MAT_ELEM( *cPerp, float, i, j ) = cvmGet( U, i, j+cols );
      }
    }
    int rank1 = getDiagonalMatrixRankBruteForce( W,rows,cols );
    int rank2 = getDiagonalMatrixRankElegant( W,rows,cols );
    if( rank1 != rank2 ) {
      std::cout << "Error: the ranks are not equal" << std::endl;
    }
```

```
    if( cols != rank1 ) {
      std::cout << "the rank of C is " << rank1 <<
        ", but the number of columns is " << cols << std::endl;
    }
    return cPerp;
}
/*
 * Only a singular, diagonal matrix may be passed as an
 * argument to this method
 */
int OrthogonalComplement::getDiagonalMatrixRankBruteForce( CvMat* W, int rows, int cols ) {
    int rank = 0;
    bool nonZeroFound;
    // search row by row until a zero row is found
    for( int i=0; i<rows; i++ ) {
      // prior to examining any elements in row
      nonZeroFound = false;
      for( int j=0; j<cols; j++ ) {
        if( cvmGet( W,i,j ) != 0 ) {
          nonZeroFound = true;
        }
      }
      if( nonZeroFound ) {
        rank++;
      }
      nonZeroFound = false;
    }
    return rank;
```

```
}
int OrthogonalComplement::getDiagonalMatrixRankElegant( CvMat* W, int rows, int cols ) {
    int rank = 0;
    for( int i=0; i<cols; i++ ) {
        if( cvmGet( W,i,i ) != 0 ) {
            rank++;
        }
    }
    return rank;
}
```

A.7 RotationEstimator.cpp

```
#include "RotationEstimator.h"
#include <cv.h>
#include <iostream>
#include "libs/lm.h"
#include <math.h>
#include "NumberConverter.h"
#define PI 3.14159265
/*
 * precondition : V must be a 2x1 vector
 */
void buildV( CvMat* V, CvPoint2D32f* p1, CvPoint2D32f* p2 ) {
    CV_MAT_ELEM( *V, float, 0, 0 ) = p1->x - p2->x;
    CV_MAT_ELEM( *V, float, 1, 0 ) = p1->y - p2->y;
}
```

```
/*
 * precondition : a 2x1 vector was inputted
 */
double get2dVectorLength( CvMat* vec ) {
    double x = cvmGet( vec, 0, 0 );
    double y = cvmGet( vec, 1, 0 );
    double len = sqrt( (x*x) + (y*y) );

    return len;
}

/*
 * Will turn the inputted vector into one of unit length
 */
void normalize2dVector( CvMat* vec ) {

    // get the length of the vector
    double len = get2dVectorLength( vec );
    CV_MAT_ELEM( *vec, float, 0, 0 ) = cvmGet( vec, 0, 0 ) / len;
    CV_MAT_ELEM( *vec, float, 1, 0 ) = cvmGet( vec, 1, 0 ) / len;

    //double length = get2dVectorLength( vec );
    //std::cout << "length of normalized vec: " << length << std::endl;
}
/*
 * Puts a vector orthogonal to the inputVec into the outputVec
 * precondition : 2 2x1 (preallocated) vectors have been inputted
 * postcondition : an orthogonal vector of unit length has been returned
```

```
 */
void getOrthogonal2dUnitVector( CvMat* inputVec, CvMat* outputVec ) {

    CV_MAT_ELEM( *outputVec, float, 0, 0 ) = -1 * cvmGet( inputVec, 1, 0 );
    CV_MAT_ELEM( *outputVec, float, 1, 0 ) = cvmGet( inputVec, 0, 0 );

    normalize2dVector( outputVec );                                          50
}

void buildB( CvMat* B, CvPoint2D32f* p1 ) {
    double x = p1->x;
    double y = p1->y;
    double f = 634.0;
    //std::cout << "x: " << x << ", y: " << y << std::endl;
    CV_MAT_ELEM( *B, float, 0, 0 ) = ( x*y ) / f;
    CV_MAT_ELEM( *B, float, 0, 1 ) = -1* ( f + ( x*x / f ) );
    CV_MAT_ELEM( *B, float, 0, 2 ) = y;                                      60

    CV_MAT_ELEM( *B, float, 1, 0 ) = f + ( y*y / f );
    CV_MAT_ELEM( *B, float, 1, 1 ) = -1 * ( x*y / f);
    CV_MAT_ELEM( *B, float, 1, 2 ) = -1 * x;

}

void buildA( CvMat* A, CvPoint2D32f* p1 ) {
    double f = 634.0;
                                                                             70
    CV_MAT_ELEM( *A, float, 0, 0 ) = -f;
```

```
    CV_MAT_ELEM( *A, float, 0, 1 ) = 0;
    CV_MAT_ELEM( *A, float, 0, 2 ) = p1->x;

    CV_MAT_ELEM( *A, float, 1, 0 ) = 0;
    CV_MAT_ELEM( *A, float, 1, 1 ) = -f;
    CV_MAT_ELEM( *A, float, 1, 2 ) = p1->y;

}

/*
 * Method called by levmar
 *
 */
void rotationEstNonlinear( double* p, double* x, int m, int n, void *data ) {
        assert( p != NULL );
    assert( x != NULL );
    assert( data != NULL );

    CvMat* Ri = cvCreateMat( 3, 1, CV_32F );

    CV_MAT_ELEM( *Ri, float, 0, 0 ) = p[0];
    CV_MAT_ELEM( *Ri, float, 1, 0 ) = p[1];
    CV_MAT_ELEM( *Ri, float, 2, 0 ) = p[2];

    CvMat* A = cvCreateMat( 2, 3, CV_32F );
        CvMat* B = cvCreateMat( 2, 3, CV_32F );
    CvMat* AT = cvCreateMat( 2, 1, CV_32F ); // product of A and T matrices
        CvMat* d = cvCreateMat( 2, 1, CV_32F );
```

```
    CvMat* dt = cvCreateMat( 1, 2, CV_32F ); // transpose of d
CvMat* dtB = cvCreateMat( 1, 3, CV_32F );
CvMat* dtBOmega = cvCreateMat( 1, 1, CV_32F );
CvMat* V = cvCreateMat( 2, 1, CV_32F );
CvMat* dtV = cvCreateMat( 1, 1, CV_32F );

DataStruct *dptr;
dptr=(struct DataStruct *)data;
int numProcessed = 0;
for( int i=0; i<dptr->numPoints; i++ ) {
  if( dptr->featuresMap[i] == 1 ) {
    // build matrix for current set of points
    // we want to pass a pointer to feature i - hence the '&' outside the
    // accessing of the actual point
    buildA( A, &(dptr->frameOneFeatures[i]) );
    buildB( B, &(dptr->frameOneFeatures[i]) );
    //std::cout << "gets here" << std::endl;
    //cvmMul( A, dptr->T, AT );
    cvMatMul( A, dptr->T, AT );
    //std::cout << "gets here" << std::endl;
    // put the othogonal vector to AT into d
    getOrthogonal2dUnitVector( AT, d );

    // fill the transpose of d
    cvTranspose( d, dt );

    cvMatMul( dt, B, dtB );
    //std::cout << "past the creation of dB" << std::endl;
```

```
            // get the first term of the least squares expression
            cvMatMul( dtB, Ri, dtBOmega );
            //std::cout << "past creation of dBOmega" << std::endl;
            // we want to pass a pointer to feature i
            buildV( V, &(dptr->frameOneFeatures[i]), &(dptr->frameTwoFeatures[i]) );
            cvmMul( dt, V, dtV );
            //std::cout << "past creation of dtV" << std::endl;
            x[numProcessed] = cvmGet( dtBOmega, 0, 0 ) - cvmGet( dtV, 0, 0 );
            numProcessed++;
        }
    }
    cvReleaseMat( &A );
    cvReleaseMat( &B );
    cvReleaseMat( &AT );
    cvReleaseMat( &d );
    cvReleaseMat( &dt );
    cvReleaseMat( &dtB );
    cvReleaseMat( &dtBOmega );
    cvReleaseMat( &V );
    cvReleaseMat( &dtV );
}
/**
 * Responsible for estimating the rotation between 2 frames using a
 * nonlinear equation solver. The estimated translation, T, between
 * the two frames must be found prior to estimating the rotation.
 * param: myData data necessary for calculating the 3d rotation (including an estimate
 * of the 3d translation between the two frames for which we are currently estimating the
 * rotation).
```

```
*/
void RotationEstimator::estimateRotation( DataStruct* myData ) {

  double opts[LM_OPTS_SZ];
  opts[0]=LM_INIT_MU; opts[1]=1E-15; opts[2]=1E-15; //opts[3]=1E-20;                    160
  opts[3]=1E-15;//1E-4;
  opts[4]=LM_DIFF_DELTA;

  std::cout << "before cvmGets on myData->R" << std::endl;
  // set the initial guess
  p[0] = cvmGet( myData->R, 0, 0 ); // x rotation
  p[1] = cvmGet( myData->R, 1, 0 ); // y rotation
  p[2] = cvmGet( myData->R, 2, 0 ); // z rotation

  std::cout << "entering estimator, the rotation, p=[ " << p[0] << ", "              170
     << p[1] << ", " << p[2] << "]" << std::endl;

  double x[myData->numFeaturesProcessed];
  for( int i=0; i < myData->numFeaturesProcessed; i++ ) {
    x[i] = 0;
  }
  NumberConverter numConv;
  double lowerBound = numConv.degreesToRadians( -10.0 );
  double upperBound = numConv.degreesToRadians( 10.0 );
                                                                                      180
  std::cout << "Lower Bound: " << lowerBound << ", Upper Bound: " << upperBound << std::endl;

  double lb[3], ub[3];
```

```
    lb[0] = lowerBound; lb[1] = lowerBound; lb[2] = lowerBound;
    ub[0] = upperBound; ub[1] = upperBound; ub[2] = upperBound;


    int ret = dlevmar_bc_dif( rotationEstNonlinear, p, x, 3, myData->numFeaturesProcessed,
          lb, ub, 1000, opts, info, NULL, NULL, (void *)myData ); //&myData );
          // once the nonlinear solver has been completed, the estimate is stored in p
    // assign it to myData.R
    CV_MAT_ELEM( *(myData->R), float, 0, 0 ) = p[0];
    CV_MAT_ELEM( *(myData->R), float, 1, 0 ) = p[1];
    CV_MAT_ELEM( *(myData->R), float, 2, 0 ) = p[2];

    std::cout <<"exiting estimator, the rotation p=[ " <<
       p[0] << ", " << p[1] << ", " << p[2] << "]" << std::endl;

    printf("Levenberg-Marquardt returned %d in %g iter, reason %g\nSolution: ",
          ret, info[5], info[6]);
          for(int i=0; i<3; ++i) {
       printf("%.7g ", p[i]);
    }
          printf("\n\nMinimization info:\n");
          for(int i=0; i<LM_INFO_SZ; ++i) {
       printf("%g ", info[i]);
    }
          printf("\n");
}
/*
 * This is the output info of the levmar solver, and basically contains information about the
```

```
 * convergence (or lack thereof) of the data
 */
double* RotationEstimator::getNonlinearSolverInfo( ) {
    return info;
}
/*
 * Make the pointer NULL so it correctly throws an exception
 * when one attempts to access data that is no longer present
 */                                                              220
RotationEstimator::~RotationEstimator( ) {
    //delete [] p;
    //delete [] info;
    //cvReleaseMat( &orthVec );
}
```

A.8 TranslationEstimator.cpp

```
#include "OrthogonalComplement.h"
#include "TranslationEstimator.h"
#include "NumberConverter.h"
#include "LeastSquares.h"
#include "CMatrixBuilder.h"
#include "UnitSphere.h"
#include "UnitSpherePatch.h"
#include <float.h>
#include <iostream>
#include "libs/lm.h"                                             10
#include <cv.h>
```

```
/**
 * class: TranslationEstimator.cpp
 * brief: Given an initial guess at the translation between two images, it will
 * use a nonlinear solver to give an accurate estimate.
 */

CvMat* buildVt( DataStruct* dptr ) {
  char* opticalFlowFoundFeature = dptr->featuresMap;
  CvPoint2D32f* p1 = dptr->frameOneFeatures;
  CvPoint2D32f* p2 = dptr->frameTwoFeatures;
  int numPoints = dptr->numPoints;
  CvMat* Vt = cvCreateMat(1, 2*dptr->numFeaturesProcessed, CV_32F );
  int i = 0;
  int index = 0;
  while( i < numPoints ) {
    if( opticalFlowFoundFeature[i] == 1 ) {
      CV_MAT_ELEM( *Vt, float, 0, 2*index ) = p1[i].x - p2[i].x;
      CV_MAT_ELEM( *Vt, float, 0, (2*index)+1 ) = p1[i].y - p2[i].y;
      index++;
    }
    i++;
  }
  return Vt;
}
/**
 * fn: translationEstNonlinear Nonlinear translation estimator. It is called indirectly using the levmar library.
 * param: p estimate of the 3d translation
 * param: x a vector of zeroes (we want the estimate that renders as close to zero as possible)
```

```
 * param: m dimension of p                                                    40
 * param: n dimension of x
 * param: data pointer to an instance of DataStruct containing the flow information needed for calculation
 */
void translationEstNonlinear( double* p, double* x, int m, int n, void *data ) {
  CvMat* T = cvCreateMat( 3, 1, CV_32F );
  assert( data != NULL ); // a mistake I frequently make

      // take guess and convert it to usable format
  CV_MAT_ELEM( *T, float, 0, 0 ) = p[0];
  CV_MAT_ELEM( *T, float, 1, 0 ) = p[1];                                     50
  CV_MAT_ELEM( *T, float, 2, 0 ) = p[2];

      struct DataStruct *dptr;
      dptr=(struct DataStruct *)data;
      CMatrixBuilder cBuilder( dptr->frameOneFeatures, dptr->frameTwoFeatures,
                               dptr->featuresMap, dptr->numPoints );
      cBuilder.fillA_T( T );
      cBuilder.fillB( );
      cBuilder.fillC( );
                                                                             60
      CvMat* C = cBuilder.getC( );
      OrthogonalComplement orthComp( C, cBuilder.getCRowSize( ), cBuilder.getCColumnSize( ) );
      orthComp.evalSVDecomposition( );
      CvMat* CPerp = orthComp.getOrthogonalComplement( );

/// Vt is the transpose of the column vector comprised of the x & y
/// components of the elements of the optical flow field
```

```
    CvMat* Vt = buildVt( dptr );

                                                                    70
        /// where the result of Vt * CPerp is stored
    /// the column count is based on the shape of CPerp as a
    /// result of taking the right side of the U matrix after
    /// the SV Decomposition
    /// note: numFeaturesProcessed - 3, because it is actually
    /// numRows - numCols
        CvMat* VtCPerp = cvCreateMat( 1,
                    (2 * dptr->numFeaturesProcessed) - dptr->numFeaturesProcessed - 3, CV_32F );
    //std::cout << "gets here" << std::endl;
        cvmMul( Vt, CPerp, VtCPerp );                               80
    //cvMatMul( Vt, CPerp,
    for( int i=0; i< ( (2 * dptr->numFeaturesProcessed) - dptr->numFeaturesProcessed - 3); i++ ) {
      x[i] = cvmGet( VtCPerp, 0, i );
    }
    dptr->T = T;
}
/**
 * Iterate through all candidates on the unit sphere and see which one
 * has the minimum residual
 */                                                                 90
CvMat* TranslationEstimator::getTranslation( DataStruct *myData ) {
    assert( myData != NULL ); // a mistake I frequently make

    double minResid = DBL_MAX;
    int minResidPos = -1;
```

```
CvMat* T;
UnitSphere unitSphere;
LeastSquares leastSquares;
std::ofstream myfile;                                                    100
myfile.open( "minResidTrans.txt" );
for( int i=0; i<unitSphere.getNumUnitVectors( ); i++ ) {
   T = unitSphere.getUnitVectorAt( i );
   CMatrixBuilder cBuilder( myData->frameOneFeatures, myData->frameTwoFeatures,
         myData->featuresMap, myData->numPoints );

   cBuilder.fillA_T( T );
   cBuilder.fillB( );
   cBuilder.fillC( );
   CvMat* C = cBuilder.getC( );                                          110
   OrthogonalComplement orthComp( C, cBuilder.getCRowSize( ), cBuilder.getCColumnSize( ) );
   orthComp.evalSVDecomposition( );
   CvMat* CPerp = orthComp.getOrthogonalComplement( );

   // Vt is the transpose of the column vector comprised of the x & y
   // components of the elements of the optical flow field
   CvMat* Vt = buildVt( myData );

   // where the result of Vt * CPerp is stored
   // the column count is based on the shape of CPerp as a              120
   // result of taking the right side of the U matrix after
   // the SV Decomposition
   // note: numFeaturesProcessed - 3, because it is actually
```

```
        // numRows - numCols
        int numVtCPerpCols =
          (2 * myData->numFeaturesProcessed) - myData->numFeaturesProcessed - 3;
        CvMat* VtCPerp = cvCreateMat( 1, numVtCPerpCols, CV_32F );
        //cvmMul( Vt, CPerp, VtCPerp );
        cvMatMul( Vt, CPerp, VtCPerp );
        double lsq = leastSquares.evaluateColVecL2Norm( VtCPerp, numVtCPerpCols );
        double resid = lsq / numVtCPerpCols;
        //std::cout << "evaluating " << std::endl;

        myfile << " " << cvmGet( unitSphere.getSphericalCoordsAt( i ), 0, 0 ) << " " <<
          cvmGet( unitSphere.getSphericalCoordsAt( i ), 1, 0 ) << " " << resid;
        myfile << std::endl;
        if( resid < minResid ) {
          minResid = resid;
          minResidPos = i;
          std::cout << "the new min is t=[ " << cvmGet( unitSphere.getUnitVectorAt( i ), 0, 0 )
            << " " << cvmGet( unitSphere.getUnitVectorAt( i ), 1, 0 ) <<
            " " << cvmGet( unitSphere.getUnitVectorAt( i ), 2, 0 )
            << " ], and the residual is " << minResid << std::endl;
        }
        cvReleaseMat( &Vt );
        cvReleaseMat( &VtCPerp );
      }
      //cvReleaseMat( &T );
      myfile.close( );
      return unitSphere.getUnitVectorAt( minResidPos );
    }
```

```
/**
 * Method that calls the private nonlinear rotation solver. The method takes the previous final estimate
 * of the translation as the starting point for a new estimate of the translation.
 * param: myData pointer to the struct that contains all of the necessary data to calculate the
 * rotational portion of the egomotion. It has an estimate of the translative component (myData->T),
 * and an initial guess for the rotational component (myData->R).
 * see: translationEstNonlinear( double* p, double* x, int m, int n, void *data )
 */
void TranslationEstimator::estimateTranslation( DataStruct *myData ) {
  std::cout << "in translation estimator" << std::endl;
  int ret;
  double opts[ LM_OPTS_SZ ];
  double info[ LM_INFO_SZ ]; // output variable that indicates convergence, etc
  opts[0]= LM_INIT_MU; //opts[1]=1E-15; opts[2]=1E-15; opts[3]=1E-20;
  opts[1]=1E-15;
  opts[2]=1E-15;
  opts[3]=1E-20; //4;
  opts[4]= LM_DIFF_DELTA; // relevant only if the finite difference jacobian version is used

  int m = 3; // dim of p vector
      int n = (2*myData->numFeaturesProcessed) - myData->numFeaturesProcessed - 3;

      double p[m];
      double x[n];

 // by
```

```
    p[0] = cvmGet( myData->T, 0, 0 );
    p[1] = cvmGet( myData->T, 1, 0 );
    p[2] = cvmGet( myData->T, 2, 0 );

for( int i=0; i<n; i++ ) {
  x[i] = 0.0;
}
// stick boundaries on this sucka
double lb[3],ub[3];
lb[0] = -1.0; lb[1] = -1.0; lb[2] = -1.0;
ub[0] = 1.0; ub[1] = 1.0; ub[2] = 1.0;
ret = dlevmar_bc_dif( translationEstNonlinear, p, x, m, n,
    lb, ub, 1000, opts, info,  NULL, NULL, (void *)myData );

// now that it has been solved, be sure to update the translation
// estimate in the DataStruct
CV_MAT_ELEM( *(myData->T), float, 0, 0 ) = p[0];
CV_MAT_ELEM( *(myData->T), float, 1, 0 ) = p[1];
CV_MAT_ELEM( *(myData->T), float, 2, 0 ) = p[2];

printf("Levenberg-Marquardt returned %d in %g iter, reason %g\nSolution: ",
    ret, info[5], info[6]);
for(int i=0; i<m; ++i) {
  printf("%.7g ", p[i]);
}
printf("\n\nMinimization info:\n");
for(int i=0; i<LM_INFO_SZ; ++i) {
  printf("%g ", info[i]);
```

```
    }
    printf("\n");
}
```

A.9 UnitSphere.cpp

```
#include <iostream>
#include <math.h>
#include "UnitSphere.h"
#include "NumberConverter.h"

UnitSphere::UnitSphere( ) {
    //numUnitVectors = 61 * 121;
    numUnitVectors = 3600;
    generateAllUnitVectors( );
}
void UnitSphere::initialTranslationGuess( CvMat* T ) {
    NumberConverter numConv;
    CvMat* tSpherical = cvCreateMat( 3, 1, CV_32F );
    CvMat* tNew = cvCreateMat( 3, 1, CV_32F );
    numConv.cartesianToSpherical( T, tSpherical );

    numConv.sphericalToCartesian( tSpherical, tNew );

/*
 * generateAllUnitVectors( )
```

```
 * We generate all possible unit vectors using
 * spherical coordinates (rho = 1, sweet!), and
 * then convert them to cartesian ones
 *
 */
void UnitSphere::generateAllUnitVectors( ) {
  double rho = 1.0;
  int i = 0; // i is the index for the array
  //for( int theta=-180; theta<=180; theta++ ) {                    30
  NumberConverter numConv;
  for( int theta=0; theta<180; theta+=3 ) { // <180 because 0 & 180 refer to the same vector
    for( int phi=270; phi<360; phi+=3 ) {

      // the next 3 lines are for returning the spherical coords for graphing purposes
      S[i] = cvCreateMat( 2, 1, CV_32F );
      CV_MAT_ELEM( *S[i], float, 0, 0 ) = phi;
      CV_MAT_ELEM( *S[i], float, 1, 0 ) = theta;

      T[i] = cvCreateMat( 3, 1, CV_32F );                         40
      CV_MAT_ELEM( *T[i], float, 0, 0 ) =
        rho * sin( numConv.degreesToRadians( phi ) ) *
        cos( numConv.degreesToRadians( theta ) ); // x
      CV_MAT_ELEM( *T[i], float, 1, 0 ) =
        rho * sin( numConv.degreesToRadians( phi ) ) *
        sin( numConv.degreesToRadians( theta ) ); // y
      CV_MAT_ELEM( *T[i], float, 2, 0 ) = rho *
        cos( numConv.degreesToRadians( phi ) ); // z
```

```
        i++;
      }
      for( int phi=0; phi<90; phi+=3 ) { // 90 refers to the same vector as 270
        S[i] = cvCreateMat( 2, 1, CV_32F );
                    CV_MAT_ELEM( *S[i], float, 0, 0 ) = phi;
                    CV_MAT_ELEM( *S[i], float, 1, 0 ) = theta;

        T[i] = cvCreateMat( 3, 1, CV_32F );
        CV_MAT_ELEM( *T[i], float, 0, 0 ) = rho *
          sin( numConv.degreesToRadians( phi ) ) *
          cos( numConv.degreesToRadians( theta ) ); // x
        CV_MAT_ELEM( *T[i], float, 1, 0 ) = rho *
          sin( numConv.degreesToRadians( phi ) ) *
          sin( numConv.degreesToRadians( theta ) ); // y
        CV_MAT_ELEM( *T[i], float, 2, 0 ) = rho *
          cos( numConv.degreesToRadians( phi ) ); // z
        i++;
      }
    }
    std::cout << "number of points: " << i << std::endl;
}
void UnitSphere::printUnitVectorAt( int i ) {
    std::cout << "the new min is t=[ " << cvmGet( getUnitVectorAt( i ), 0, 0 )
      << " " << cvmGet( getUnitVectorAt( i ), 1, 0 ) << " " <<
      cvmGet( getUnitVectorAt( i ), 2, 0 ) << std::endl;
}
/*
 * getNumUnitVectors( )
```

```
 * Get the number of unit vectors stored in the sequential list
 */
int UnitSphere::getNumUnitVectors( ) {                                    80
   return numUnitVectors;
}
/*
 * All unit vectors are stored in a sequential list.
 * This method is how they are accessed.
 */
CvMat* UnitSphere::getUnitVectorAt( int i ) {
      return T[i];
}
/**                                                                       90
 * The 0th element in S[i] is phi, while the 1st is
 * theta
 */
CvMat* UnitSphere::getSphericalCoordsAt( int i ) {
   return S[i];
}
```