LYAPUNOV-BASED RANGE AND MOTION IDENTIFICATION FOR AFFINE AND NON-AFFINE 3D VISION SYSTEMS

By

SUMIT GUPTA

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2006

To my parents, my sister, and Amrita.

ACKNOWLEDGMENTS

I express my most sincere appreciation to my supervisory commitee chair, mentor, and friend, Dr. Warren E. Dixon. His contribution to my current and ensuing career cannot be overemphasized. I thank him for the education, advice, and for giving me opportunities that would otherwise not be possible. A special thanks to Dr. Rick Lind for his technical insight and encouragement. I express my appreciation and gratitude to Dr. Carl Crane, Dr. Thomas Burks, and Dr. Dapeng Oliver Wu, for lending their knowledge and support. It is a great priviledge to work with such far-thinking and inspirational individuals. All that I have learnt and accomplished during the course of my thesis would not have been possible without their dedication.

I especially thank Dr. Nick Gans for his invaluable guidance and support during the last semester of my research. I thank all of my colleagues who helped during my thesis research: Siddhartha Mehta, Guoqiang Hu, Darren Aiken, Sanjay Solanki, and Vilas Chitrakaran.

Most importantly I would like to express my deepest appreciation to my parents, my sister, and Amrita. Their love, understanding, patience and personal sacrifice made this dissertation possible.

TABLE OF CONTENTS

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

LYAPUNOV-BASED RANGE AND MOTION IDENTIFICATION FOR AFFINE AND NON-AFFINE 3D VISION SYSTEMS

By

Sumit Gupta

May 2006

Chair: Dr. Warren E. Dixon
Major Department: Mechanical and Aerospace Engineering

Many applications require interpreting Euclidean motion of features of a 3-dimensional (3D) object through 2D images. We examined determination of the Euclidean coordinates of the features of a 3D object undergoing general affine motion for a central paracatadioptric imaging system. The nonlinear estimator we examined asymptotically determines the Euclidean motion and range information from a single camera, provided that some observability conditions are satisfied, and that the Euclidean motion parameters are known. I proposed techniques which were developed through a Lyapunov-based design and stability analysis. Simulation results show the performance of the state estimator.

However, unlike image systems based on a planar image surface (or spherical or ellipsoidal surfaces), the dynamic system resulting from projecting a general 3D imaging surface is not guaranteed to maintain an affine form. Because of the nonaffine form, existing range-identification observers cannot be directly used. Therefore we developed a nonlinear state estimator that can be applied to a nonaffine vision system to determine the range and Euclidean coordinates of an object feature without using linear methods.

As with many vision-based estimation and control strategies, our results were based on feature point tracking. Feature point's tracking is an essential capability of many coordinated guidance, navigation, and control applications. The hardware-in-the-loop simulation (HILS) facility at the University of Florida provides a rapid prototyping testbed for simulating such applications. The facility centers around a virtual environment to investigate visualization and computational technologies and assists in simulating vision-based feedback controllers. We described the HILS testbed, examined the theoretical concepts behind the visual servoing algorithms, and discussed solutions to problems faced while developing the control loop.

CHAPTER 1
INTRODUCTION

Autonomous vehicle/robotic guidance, navigation and control applications typically require interpretating of the Euclidean motion of features of a 3-dimensional (3D) object through 2D images that are projected from the 3D feature. Determining the Euclidean motion of an object through the image projection is challenging, because the distance from the camera to the object along the focal length (i.e., the range to the object) is an unmeasurable time-varying signal that appears nonlinearly in the projected image-space. Although the problem of determining the Euclidean coordinates of a moving object from its 2D images is inherently nonlinear, typical results are based on linearization based methods such as extended Kalman filtering ([5], [10]). Classical structure from motion (SFM) results, ([18], [31]), are examples of such linearization-based methods.

Several researchers have recently investigated the problem of determining the unknown states (i.e., the Euclidean coordinates) when the motion parameters (feature velocities) are known. For example, a discontinuous observer was developed [17] to exponentially identify range information of features from successive images of a camera where the object model is based on known skew-symmetric affine motion parameters. Motivated to generalize the object motion beyond the skew-symmetric form [17], Chen and Kano developed a new discontinuous observer [6] that exponentially forces the state observation error to be uniformly ultimately bounded (UUB) for known motion parameters. In comparison to the UUB result [6], a continuous observer was constructed [8] to asymptotically identify the range information for a general affine system with known motion parameters (i.e., result [8] eliminated the skew-symmetric assumption and yielded an asymptotic result

with a continuous observer). Ma et al. ([20], [21]) developed a state estimation strategy for affine systems with known motion parameters where only a single homogeneous observation point is provided (i.e., a single image coordinate).

All of the aforementioned results assumed that a conventional planar imaging surface were used. Using a planar imaging surface is restrictive for some applications because of limitations in the field of view (FOV). To improve the FOV, researchers proposed using spherical, elliptical, or paraboloid imaging surfaces [22]; rotating imaging systems [1]; fish-eye lenses [32]; catadioptric lenses [25]; or a cluster of cameras [28]. A vision system retains an affine form [22] for some of these imaging surfaces (e.g., planar, spherical, or ellipsoidal). That is, the projection of the affine Euclidean motion of an object onto some imaging surfaces yields an affine dynamic system. In previous results, the affine form of the vision system was transformed into the nonlinear system expressed by:

$$\begin{aligned} \dot{x}_1 &= \omega^T(x_1, u)x_2 + \phi(x_1, u) \\ \dot{x}_2 &= g(x_1, x_2, u) \\ y &= x_1 \end{aligned} \tag{1–1}$$

where $\omega(x_1, u)$ and $g(x_1, x_2, u)$ are nonlinear functions of their arguments. Once the vision system is written as in equation (1–1), the identifier based observer (IBO) proposed by Jankovic and Ghosh [17] can be applied directly to determine the object range and motion [22].

A catadioptric system combines reflective (catoptric) and refractive (dioptric) elements (i.e., a camera and a mirror) [16]. Catadioptric systems with a single effective viewpoint are classified as central catadioptric systems. Central catadioptric systems are desirable because they yield pure perspective images [12]. Baker and Nayar [3] derived the complete class of single-lens single-mirror catadioptric

systems (e.g., paraboloid mirror under orthographic projection) that satisfy the single-viewpoint constraint.

Catadioptric systems provide a larger FOV in a manner that is better than alternative technologies. For example, a rotating camera system has a reduced effective bandwidth, has moving parts, and requires extra care to be taken to eliminate blur as the acquired images are stitched together to construct a panoramic scene. For many applications, the cost of a cluster of cameras is inhibitive when compared to a catadioptric system with a similar FOV. Moreover, the viewpoints of all the cameras must coincide for a cluster of cameras to generate pure perspective images (which is a nontrivial calibration obstacle).

Catadioptric systems also exhibit several limitations. In general, the coordinates of an object are projected onto a mirror and then onto a camera lens. For cameras using a lens that yields a perspective projection, alignment of lens and mirror must be calibrated for the distance between them. Paracatadioptric systems are a special kind of central catadioptric system constructed with a paraboloid mirror and an orthographic lens. Using the orthographic lens reduces the alignment requirements; and hence, simplifies calibration of the system, and computation of pure-perspective images [24]. Compared to other technologies that extend the FOV, another limitation of catadioptric systems is that using a curved mirror warps the image. This distorted image mapping presents a challenging obstacle for reconstructing the Euclidean coordinates of observed feature points.

Our contribution was to develop a nonlinear estimator, [8], to extract range information and Euclidean coordinates from a paracatadioptric system. We used a Lyapunov-based analysis to prove that the 3D Euclidean coordinates of an object moving with general affine motion are asymptotically identified.

However, a projected image from a general 3D imaging surface is not guaranteed to maintain an affine form. Motivated by the benefits of an improved FOV,

Ma et al. suggested that range identification could be achieved using a linear approximation-based observer for a paraboloid imaging system whose focal point and vertex coincide. The resulting state estimation of the original nonlinear system is produced from a sequence of approximate linear, time-varying observers [22]. Gupta et al. [14] constructed a nonlinear observer, based on [8], to asymptotically identify the range for the system considered [22]. The IBO [22] (or most of the existing nonlinear observers) can not be directly applied for projections on a paraboloid imaging system. Ma et al. [22] suggested that range identification could be achieved for a paraboloid image surface using a linear approximation-based observer where the state estimation of the original nonlinear system is carried out by a sequence of approximate linear time-varying observers [21].

I developed a nonlinear estimator that can be used to identify range and the Euclidean coordinates from a nonaffine imaging system without the use of linear approximations [22]. The structure of the nonlinear observer is based on a previous observer [8]; however, new observability conditions are imposed due to the nonaffine form of the projection resulting from the paraboloid image surface. A Lyapunov-based analysis is used to prove the range and the 3D coordinates of an object moving with general affine Euclidean motion (and nonaffine image dynamics) are asymptotically identified. Numerical simulation results are provided that illustrate the performance of the estimator.

We considered the affine Euclidean motion of an object feature [8]

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{1–2}$$

where $x(t) = [x_1(t), x_2(t), x_3(t)]^T \in \mathbb{R}^3$ denote the unmeasurable Euclidean coordinates of an object feature along the X, Y, and Z axes of an inertial reference

frame, respectively, and the Z axis is colinear with the optical axis of the camera. In (1–2), the parameters $a_{i,j}(t) \in \mathbb{R}$ and $b_i(t)$ $\forall i, j = 1, 2, 3$ denote known motion parameters [6], [30]. The affine motion dynamics introduced in (1–2) are expressed in a general form that describes an object motion that undergoes a rotation, translation, and linear deformation [30]. It is assumed that the known motion parameters $a_{i,j}(t)$ and $b_i(t)$ $\forall i, j = 1, 2, 3$ introduced in (1–2) are bounded functions of time and are second order differentiable.

As with many vision-based estimation and control strategies, the results in Chapter Two and Three are based on feature point tracking. Feature point tracking is a standard task of computer vision with numerous applications in navigation, motion understanding, surveillance, scene monitoring, and video database management. Hence, there is a definite need for a rapid prototyping testbed where visual servoing techniques can be simulated. The hardware-in-the-loop-simulation (HILS) facility at the University of Florida provides such a testbed to simulate vision based control strategies. The complete control loop at the HILS facility is described in the following steps:

1. A virtual reality simulator renders a virtual environment on the projector screens.

2. Images are captured by a camera viewing the screens and passed into an image processing workstation through a analog to digital converter using a firewire.

3. Image processing and computer vision algorithms identify feature points from the images obtained, determine the relationship between current and reference frames and accordingly generate control commands.

4. Sockets are used to communicate the generated control commands into the virtual reality simulator in order to control motion of the virtual environment.

In my thesis, I also describe the testbed and the theoretical concepts behind image processing algorithms, and discuss solutions to the problems faced while developing the control loop.

CHAPTER 2
LYAPUNOV-BASED RANGE AND MOTION IDENTIFICATION OF A PARACATADIOPTRIC VISION SYSTEM

2.1 Introduction

Autonomous vehicle/robotic guidance, navigation and control applications typically require the interpretation of the Euclidean motion of features of a 3-dimensional (3D) object through 2D images that are projected from the 3D feature. The research described in this chapter is focused on determining the Euclidean coordinates of features of a 3D object moving with affine motion dynamics using central catadioptric systems. A catadioptric system combines reflective (catoptric) and refractive (dioptric) elements (i.e. a camera and a mirror) [16]. Catadioptric systems with a single effective viewpoint are classified as central catadioptric systems. Central catadioptric systems are desirable because they yield pure perspective images [12]. Specifically, in this result we consider a paracatadioptric system (i.e., paraboloid mirror combined with an orthographic lens) derived by Baker and Nayar in [3].

The contribution of the current result is the development of a nonlinear estimator, based on [8], to extract the range information and the Euclidean coordinates from a central paracatadioptric system. Specifically, a Lyapunov-based analysis is used to prove that the 3D Euclidean coordinates of an object moving with general affine motion are asymptotically identified. Numerical simulation results are provided to illustrate the performance of the observer.

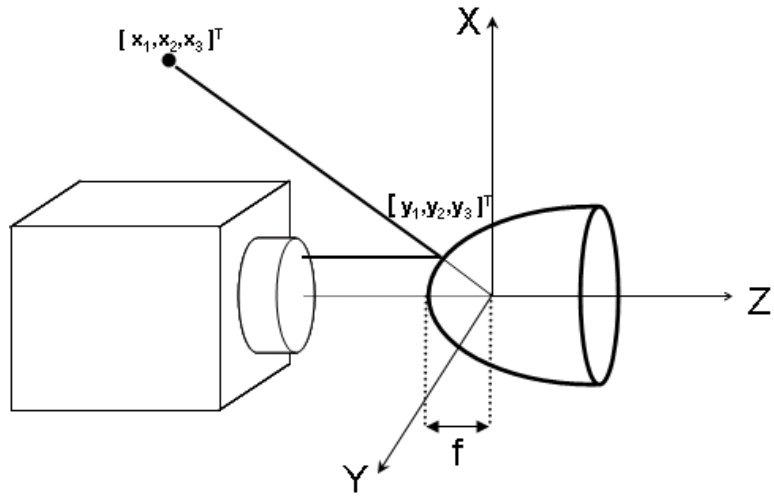

Figure 2–1: Euclidean point projected onto paraboloid mirror and then reflected to an orthographic camera.

2.2 Projection

The projection of a point $x(t)$ onto a paraboloid mirror with its focus at the origin (Figure. 2–1) can be described as follows [13]:

$$y \triangleq \begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix}^T = \frac{2f}{L} \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}^T \tag{2–1}$$

where $f \in \mathbb{R}$ denotes the constant known distance between the focal point and the vertex of the paraboloid, and $L(x) \in \mathbb{R}$ is defined as

$$L = -x_3 + \sqrt{x_1^2 + x_2^2 + x_3^2}. \tag{2–2}$$

Because the projection from the paraboloid mirror to the camera is orthographic in nature (i.e., reflected light rays are parallel to the optical axis), $y_1(t)$ and $y_2(t)$ correspond to the measured pixel coordinates of the camera. Also, because the paraboloid is rotationally symmetric, $y_3(t)$ is computed from the measured pixel coordinates as

$$y_3 = \frac{(y_1^2 + y_2^2)}{4f} - f. \tag{2–3}$$

To facilitate subsequent development, the auxiliary signal $y_4(t) \in \mathbb{R}$ is defined as

$$y_4 = \frac{2f}{L} \tag{2–4}$$

and contains the unknown range information. Substituting (2–4) into (2–1), I obtain the following system

$$y = y_4 x. \tag{2–5}$$

Taking the time derivative of (2–5) and utilizing (1–2), the following can be determined:

$$\dot{y}_1 = a_{11}y_1 + a_{12}y_2 + a_{13}y_3 + \frac{(a_{31}y_1^2 + a_{32}y_1y_2 + a_{33}y_1y_3)}{2f} + g_1 \tag{2–6}$$

$$\dot{y}_2 = a_{21}y_1 + a_{22}y_2 + a_{23}y_3 + \frac{(a_{31}y_1y_2 + a_{32}y_2^2 + a_{33}y_2y_3)}{2f} + g_2 \tag{2–7}$$

$$\dot{y}_3 = a_{31}y_1 + a_{32}y_2 + a_{33}y_3 + \frac{(a_{31}y_1y_3 + a_{32}y_2y_3 + a_{33}y_3^2)}{2f} + g_3 \tag{2–8}$$

where (2–6) through (2–8) signify the projected dynamics of the object feature onto the paracatadioptric system.

In (2–6) through (2–8), the unmeasurable signals $g(t) \triangleq [g_1(t),\, g_2(t),\, g_3(t)]^T \in \mathbb{R}$ are defined as

$$g = y_4 b + \Omega_0 y \tag{2–9}$$

where $\Omega_0(t) \in \mathbb{R}$ is defined as

$$\Omega_0 = \frac{b_3}{L} - \frac{y_1\dot{x}_1 + y_2\dot{x}_2 + y_3\dot{x}_3}{2f(L + x_3)}. \tag{2–10}$$

To facilitate the later development, the dynamics in (2–6) through (2–8) can be rewritten as

$$\begin{bmatrix} \dot{y}_1 & \dot{y}_2 & \dot{y}_3 \end{bmatrix}^T = \Omega_1 + g \tag{2–11}$$

where $\Omega_1(t) \in \mathbb{R}^3$ denotes a matrix of measurable and known signals.

Assumption 2-1: In contrast to the systems examined in [14] and [22], the development in this result is based on the more general (and more practical) assumption that the focal point is not at the vertex of the paraboloid. Moreover, the focal point is not a vanishing point (i.e. $f \in \mathcal{L}_\infty$).

Assumption 2-2: The image-space feature coordinates $y_1(t)$, $y_2(t)$ are bounded functions of time; hence, from (2–3) $y_3(t) \in \mathcal{L}_\infty$.

Assumption 2-3: The object feature is not a vanishing point (i.e. $L \in \mathcal{L}_\infty$ therefore $y_4(t) \neq 0$). I assume that $L \neq 0$ (i.e. $x_1, x_2 \neq 0$ simultaneously); hence, the object feature does not intersect the optical axis of the imaging system. Since $L \neq 0$, (2–4) can be used to conclude that $y_4(t) \in \mathcal{L}_\infty$.

Assumption 2-4: The object must translate in at least one direction (i.e. $b_1, b_2, b_3 \neq 0$ simultaneously).

From (3–9), the signal $y_4(t)$ containing the range information can be defined as

$$y_4^2 = \frac{(y_2 g_1 - y_1 g_2)^2 + (y_3 g_1 - y_1 g_3)^2 + (y_3 g_2 - y_2 g_3)^2}{(y_2 b_1 - y_1 b_2)^2 + (y_3 b_1 - y_1 b_3)^2 + (y_3 b_2 - y_2 b_3)^2}. \tag{2–12}$$

Remark 2-1: Assumptions 2-1 through 2-4 are standard assumptions that are practically properties of the physical system rather than assumptions.

Remark 2-2: Based on Assumptions 2-1 through 2-4, the expressions given in (2–6) through (3–31) can be used to determine that $\dot{y}(t)$, $\Omega_1(t)$, and $g(t) \in \mathcal{L}_\infty$. Given that these signals are bounded, Assumptions 2-1 through 2-4 and the development in the appendix can be used to prove that

$$\|g(\cdot)\| \leq \zeta_1 \qquad \|\dot{g}(\cdot)\| \leq \zeta_2 \qquad \|\ddot{g}(\cdot)\| \leq \zeta_3 \tag{2–13}$$

where ζ_1, ζ_2 and $\zeta_3 \in \mathbb{R}$ denote known positive constants.

2.3 Range and Motion Identification

2.3.1 Objective

The objective of this result is to extract the Euclidean coordinate information of the object feature from its projection onto the paracatadioptric system. From (2–5) and the fact that $y_1(t)$, $y_2(t)$ and $y_3(t)$ are measurable, if $y_4(t)$ could be identified then the complete Euclidean coordinates of the feature can be determined. To achieve this objective, an estimator is constructed based on the unmeasurable image-space dynamics for $y(t)$. To quantify the objective, a measurable estimation error, denoted by $e(t) \triangleq [e_1(t),\, e_2(t),\, e_3(t)]^T \in \mathbb{R}^3$, is defined as follows:

$$e = y - \hat{y} \tag{2–14}$$

where $\hat{y}(t) \triangleq [\hat{y}_1(t),\, \hat{y}_2(t),\, \hat{y}_3(t)]^T \in \mathbb{R}^3$ denotes a subsequently designed estimate. An unmeasurable[1] filtered estimation error, denoted by $r(t) \triangleq [e_1(t),\, e_2(t),\, e_3(t)]^T \in \mathbb{R}^3$, is also defined as

$$r = \dot{e} + \alpha e \tag{2–15}$$

where $\alpha \in \mathbb{R}^{3\times 3}$ denotes a diagonal matrix of positive constant gains α_1, α_2, $\alpha_3 \in \mathbb{R}$. The error systems in (2–14) and (2–15) are defined based on the goal to prove that the projected dynamics given in (2–6) through (2–8) can be identified (i.e., that $g(t)$ can be identified). If $g(t)$ can be identified, the fact that $y(t)$ are measurable can be used along with (2–12) to compute $y_4(t)$.

[1] The filtered estimation signal is unmeasurable due to a dependence on the unmeasurable terms $g_1(t)$, $g_2(t)$, $g_3(t)$.

2.3.2 Estimator Design and Error System

Based on (3–31) and the subsequent analysis, the following estimation signals are defined:

$$\begin{bmatrix} \dot{\hat{y}}_1 & \dot{\hat{y}}_2 & \dot{\hat{y}}_3 \end{bmatrix}^T = \Omega_1 + \hat{g} \tag{2–16}$$

where $\hat{g}(t) \triangleq [\hat{g}_1(t),\, \hat{g}_2(t),\, \hat{g}_3(t)]^T \in \mathbb{R}^3$ denotes a subsequently designed estimate for $g(t)$. The following error dynamics are obtained after taking the time derivative of $e(t)$ and utilizing (3–31) and (2–16):

$$\dot{e} = g - \hat{g}. \tag{2–17}$$

Based on the structure of (2–15) and (2–17), $\dot{\hat{g}}(t)$ is designed as follows [8]:

$$\dot{\hat{g}} = -(k_s + \alpha)\hat{g} + \gamma sgn(e) + \alpha k_s e \tag{2–18}$$

where k_s, $\gamma \in \mathbb{R}^{3\times 3}$ denote diagonal matrices of positive constant estimation gains, and the notation $sgn(\cdot)$ is used to indicate a vector with the standard signum function applied to each element of the argument. After using (2–15), (2–17) and (2–18), the following expression can be obtained:

$$\dot{r} = \eta - k_s r - \gamma sgn(e) \tag{2–19}$$

where $\eta(t) \triangleq \begin{bmatrix} \eta_1 & \eta_2 & \eta_3 \end{bmatrix}^T \in \mathbb{R}^3$ is defined as

$$\eta = \dot{g} + (k_s + \alpha)\, g. \tag{2–20}$$

Remark 2-3: Based on (2–13) and (2–20), the following inequalities can be developed:

$$|\eta(\cdot)| \leq \zeta_4 \qquad |\dot{\eta}(\cdot)| \leq \zeta_5. \tag{2–21}$$

where ζ_4 and $\zeta_5 \in \mathbb{R}$ denote known positive constants.

Remark 2-4: The structure of the estimator in (2–18) contains discontinuous terms; however, as discussed in [8], the overall structure of the estimator is continuous (i.e., $\hat{g}(t)$ is continuous).

Remark 2-5: Considering (2–12), the unmeasurable signal $y_4(t)$ can be identified if $\hat{g}(t)$ approaches $g(t)$ as $t \to \infty$ (i.e., $\hat{y}_1(t)$, $\hat{y}_2(t)$ and $\hat{y}_3(t)$ approach $y_1(t)$, $y_2(t)$ and $y_3(t)$ as $t \to \infty$) since the parameters $b_i(t)$ $\forall i = 1, 2, 3$ are assumed to be known, and $y_1(t)$, $y_2(t)$ and $y_3(t)$ are measurable. After $y_4(t)$ is identified, (2–5) can be used to extract the 3D Euclidean coordinates of the object feature (i.e. determine the range information). To prove that $\hat{g}(t)$ approaches $g(t)$ as $t \to \infty$, the subsequent development will focus on proving that $\|\dot{e}(t)\| \to 0$ and $\|e(t)\| \to 0$ as $t \to \infty$ based on (2–14) and (2–17).

2.4 Analysis

The following theorem and associated proof can be used to conclude that the observer design of (2–16) and (2–18) can be used to identify the unmeasurable signal $y_4(t)$.

Theorem 2-1: For the paracatadioptric system in consideration, the unmeasurable signal $y_4(t)$ (and hence, the Euclidean coordinates of the object feature) can be asymptotically determined from the estimator in (2–16) and (2–18) provided the elements of the constant diagonal matrix γ introduced in (2–18) are selected according to the sufficient condition

$$\gamma_i \geq \zeta_4 + \frac{1}{\alpha_i}\zeta_5 \tag{2–22}$$

$\forall i = 1, 2, 3$ and ζ_4, ζ_5 are defined in (2–21).

Proof: Consider a non-negative function $V(t) \in \mathbb{R}$ as follows (i.e., a Lyapunov function candidate):

$$V = \frac{1}{2} r^T r \ . \tag{2–23}$$

After taking the time derivative of (2–23) and substituting for the error system dynamics given in (2–19), the following expression can be obtained:

$$\dot{V} = -r^T k_s r + (\dot{e} + \alpha e)^T (\eta - \gamma sgn(e)) . \tag{2–24}$$

After integrating (2–24) and exploiting the fact that

$$\xi_i \cdot sgn(\xi_i) = |\xi_i| \ \ \forall \xi_i \in \mathbb{R},$$

the following inequality can be obtained:

$$V(t) \leq V(t_0) - \int_{t_0}^{t} \left(r^T(\sigma) k_s r(\sigma)\right) d\sigma + \sum_{i=1}^{3} \alpha_i \int_{t_0}^{t} |e_i(\sigma)| \left(|\eta_i(\sigma)| - \gamma_i\right) d\sigma + \chi_i \tag{2–25}$$

where the auxiliary terms $\chi_i(t) \in \mathbb{R}$ are defined as

$$\chi_i = \int_{t_0}^{t} \dot{e}_i(\sigma) \eta_i(\sigma) d\sigma - \gamma_i \int_{t_0}^{t} \dot{e}_i(\sigma) sgn(e_i(\sigma)) d\sigma \tag{2–26}$$

$\forall i = 1, 2, 3$. The integral expression in (2–26) can be evaluated as

$$\begin{aligned} \chi_i &= e_i(\sigma) \eta_i(\sigma)|_{t_0}^{t} - \int_{t_0}^{t} e_i(\sigma) \dot{\eta}_i(\sigma) d\sigma - \gamma_i |e_i(\sigma)| \ |_{t_0}^{t} \\ &= e_i(t) \eta_i(t) - \int_{t_0}^{t} e_i(\sigma) \dot{\eta}_i(\sigma) d\sigma - \gamma_i |e_i(t)| - e_i(t_0) \eta_i(t_0) + \gamma_i |e_i(t_0)| \end{aligned} \tag{2–27}$$

$\forall i = 1, 2, 3$. Substituting (2–27) into (2–25) and performing some algebraic manipulation yields

$$V(t) \leq V(t_0) - \int_{t_0}^{t} \left(r^T(\sigma) k_s r(\sigma)\right) d\sigma + \chi_4 + \zeta_0$$

where the auxiliary terms $\chi_4(t)$, $\zeta_0 \in \mathbb{R}$ are defined as

$$\chi_4 = \sum_{i=1}^{3} \alpha_i \int_{t_0}^{t} |e_i(\sigma)| \left(|\eta_i(\sigma)| + \frac{1}{\alpha_i} |\dot{\eta}_i(\sigma)| - \gamma_i\right) d\sigma + \sum_{i=1}^{3} |e_i(t)| \left(|\eta_i(t)| - \gamma_i\right)$$

$$\zeta_0 = \sum_{i=1}^{3} - e_i(t_0) \eta_i(t_0) + \gamma_i |e_i(t_0)| .$$

Provided γ_i $\forall i = 1, 2, 3$ are selected according to the inequality introduced in (2–22), $\chi_4(t)$ will always be negative or zero; hence, the following upper bound can be developed:

$$V(t) \leq V(t_0) - \int_{t_0}^{t} \left(r^T(\sigma)\, k_s r(\sigma)\right) d\sigma + \zeta_0 \ . \tag{2–28}$$

From (3–27) and (3–38), the following inequalities can be determined:

$$V(t_0) + \zeta_0 \geq V(t) \geq 0;$$

hence, $r(t) \in \mathcal{L}_\infty$. The expression in (3–38) can be used to determine that

$$\int_{t_0}^{t} \left(r^T(\sigma)\, k_s r(\sigma)\right) d\sigma \leq V(t_0) + \zeta_0 < \infty \ . \tag{2–29}$$

By definition, (2–29) can now be used to prove that $r(t) \in \mathcal{L}_2$. From the fact that $r(t) \in \mathcal{L}_\infty$, (2–14) and (2–15) can be used to prove that $e(t)$, $\dot{e}(t)$, $\hat{y}(t)$, and $\dot{\hat{y}}(t) \in \mathcal{L}_\infty$. The expressions in (2–16) and (2–18) can be used to determine that $\hat{g}(t)$ and $\dot{\hat{g}}(t) \in \mathcal{L}_\infty$. Based on (2–13), the expressions in (2–19) and (2–20) can be used to prove that $\eta(t)$, $\dot{\eta}(t)$, $\dot{r}(t) \in \mathcal{L}_\infty$. Based on the fact that $r(t)$, $\dot{r}(t) \in \mathcal{L}_\infty$ and that $r(t) \in \mathcal{L}_2$, Barbalat's Lemma [27] can be used to prove that $\|r(t)\| \to 0$ as $t \to \infty$; hence, standard linear analysis can be used to prove that $\|e(t)\| \to 0$ and $\|\dot{e}(t)\| \to 0$ as $t \to \infty$. Based on the fact that $\|e(t)\| \to 0$ and $\|\dot{e}(t)\| \to 0$ as $t \to \infty$, the expression given in (2–14) can be used to determine that $\hat{y}_1(t)$, $\hat{y}_2(t)$ and $\hat{y}_3(t)$ approach $y_1(t)$, $y_2(t)$ and $y_3(t)$ as $t \to \infty$, respectively. Therefore, the expression in (2–17) can be used to determine that $\hat{g}(t)$ approaches $g(x)$ as $t \to \infty$. The result that $\hat{g}(t)$ approaches $g(x)$ as $t \to \infty$, the fact that the parameters $b_i(t)$ $\forall i = 1, 2, 3$ are assumed to be known, and the fact that the image-space signals $y_1(t)$, $y_2(t)$ and $y_3(t)$ are measurable can be used to identify the unknown signal $y_4(t)$ from (2–12). Once $y_4(t)$ is identified, the complete Euclidean coordinates of the object feature can be determined using (2–5).

2.5 Numerical Simulation

In this section, numerical simulation results are provided to illustrate the performance of the range identification observer for the paracatadioptric system. The object feature is assigned the following affine motion dynamics [8]:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix} = \begin{bmatrix} -0.2 & 0.4 & -0.6 \\ 0.1 & -0.2 & 0.3 \\ 0.3 & -0.4 & 0.4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 0.5 & 0.25 & 0.3 \end{bmatrix}^T$$

with the initial Euclidean coordinates:

$$\begin{bmatrix} x_1(0) & x_2(0) & x_3(0) \end{bmatrix}^T = \begin{bmatrix} 0.4 & 0.6 & 1 \end{bmatrix}^T.$$

By arbitrarily letting $f = \frac{1}{2}$, the expressions in (2–1), (2–2), and (2–4) can be used to determine that

$$\hat{y}_1(t_0) = y_1(t_0) = 1.72$$
$$\hat{y}_2(t_0) = y_2(t_0) = 2.58$$
$$\hat{y}_3(t_0) = y_3(t_0) = 4.29$$
$$y_4(t_0) = 4.29.$$

The estimates for $g(x)$ were initialized as follows:

$$\hat{g}_1(t_0) = 1 \quad \hat{g}_2(t_0) = 1 \quad \hat{g}_3(t_0) = 1.$$

After adjusting the observer gains as

$$k_s = diag\{50, 50, 50\} \qquad \alpha = diag\{15, 15, 15\}$$
$$\gamma = diag\{1, 1, 1\} \times 10^{-5}$$

the resulting mismatch between $g_1(x)$ and $\hat{g}_1(t)$, $g_2(x)$ and $\hat{g}_2(t)$, and $g_3(x)$ and $\hat{g}_3(t)$ (i.e., $\dot{e}_1(t)$, $\dot{e}_2(t)$, and $\dot{e}_3(t)$ respectively) is depicted in Figure. 2–2. The mismatch between $y_4(t)$ and $\hat{y}_4(t)$ is provided in Figure. 2–3. The $y_4(t)$ term is

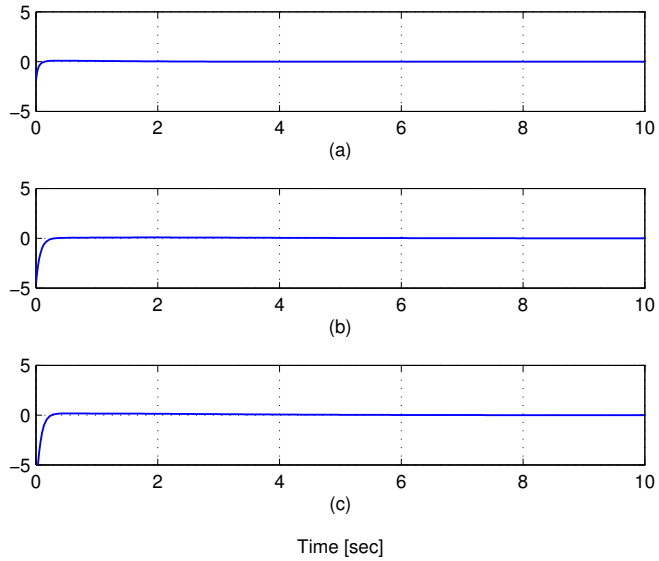

Figure 2–2: Estimation error of auxiliary signals (a) $\dot{e}_1(t)$ (b) $\dot{e}_2(t)$ and (c) $\dot{e}_3(t)$ in [pixels/sec].

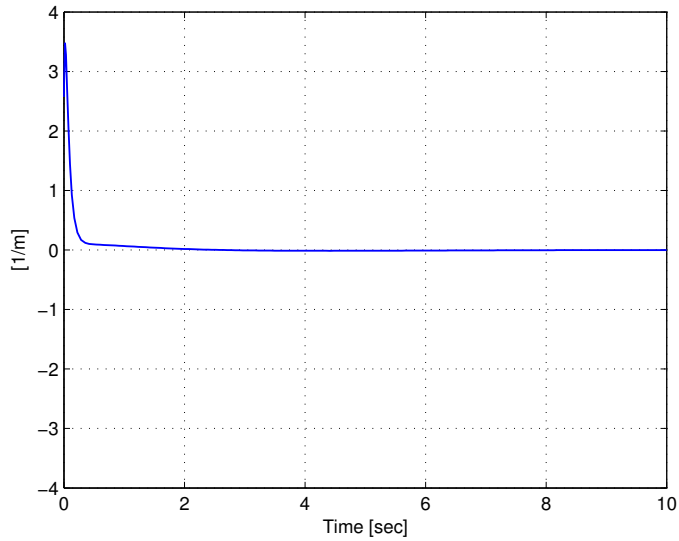

Figure 2–3: Mismatch between $y_4(t)$ and $\hat{y}_4(t)$.

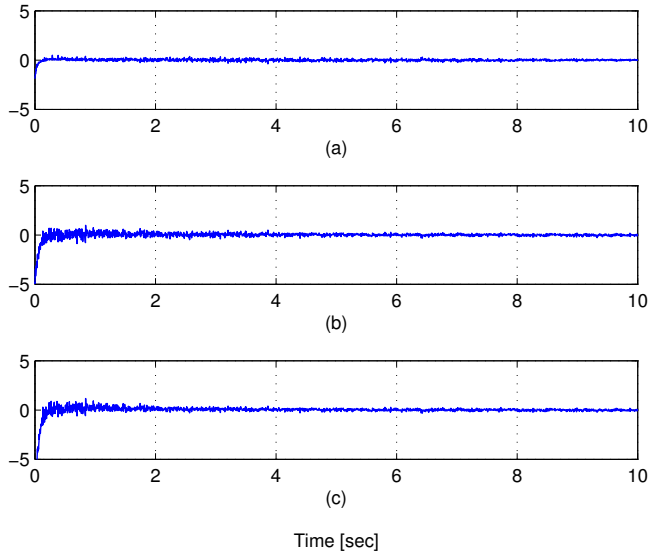

Figure 2–4: Estimation error of auxiliary signals in the presence of noise (a) $\dot{e}_1(t)$ (b) $\dot{e}_2(t)$ and (c) $\dot{e}_3(t)$ in [pixels/sec].

obtained from numerical integration of the $\dot{y}_4(t)$ term, while the estimated value is obtained by replacing $g(x)$ with $\hat{g}(t)$ in (3–19).

Additive-white-gaussian-noise (AWGN) was injected into the measurable image-space signals $y_1(t)$, $y_2(t)$ via the awgn() function in MATLAB, while maintaining a constant signal-to-noise-ratio of 20. Without changing any of the other simulation parameters, the mismatch between $g_1(x)$ and $\hat{g}_1(t)$, $g_2(x)$ and $\hat{g}_2(t)$, and $g_3(x)$ and $\hat{g}_3(t)$ (i.e., $\dot{e}_1(t)$, $\dot{e}_2(t)$, and $\dot{e}_3(t)$ respectively) is provided in Figure. 2–4, while the mismatch between $y_4(t)$ and $\hat{y}_4(t)$ is provided in Figure. 2–5.

The results depicted in Figures. 2–2 through 2–5 indicate that the proposed observer can be used to identify range, and hence, the Euclidean coordinates of an object feature moving with affine motion dynamics projected onto a paracatadioptric system provided the observability conditions are satisfied. These results are comparable to the results obtained in [8] for a planar image surface.

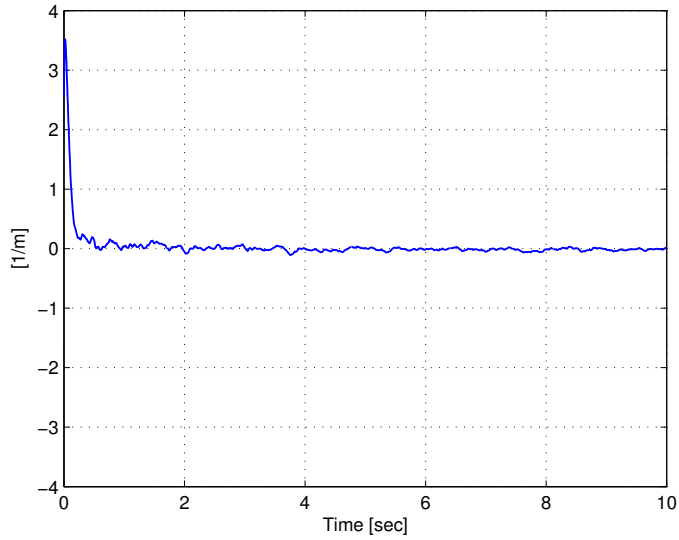

Figure 2–5: Mismatch between $y_4(t)$ and $\hat{y}_4(t)$ in the presence of noise.

2.6 Conclusion

The range and the Euclidean motion of an object feature undergoing general affine motion are determined for a central paracatadioptric system via a nonlinear estimator. The nonlinear estimator was proven, via Lyapunov-based analysis, and numerically demonstrated to asymptotically determine the range and Euclidean coordinates of features of a 3D object moving with known motion parameters, for a paracatadioptric system

CHAPTER 3
LYAPUNOV-BASED RANGE AND MOTION IDENTIFICATION OF A NONAFFINE PERSPECTIVE DYNAMIC SYSTEM

3.1 Introduction

Consider a 3-dimensional (3D) object undergoing affine motion in Euclidean space. A vision system retains the affine form for some imaging surfaces (e.g., planar, spherical, or ellipsoidal). However, projection of the affine Euclidean motion of an object onto some imaging surfaces is not guaranteed to yield an affine form in general. L. Ma in [22], states that the IBO (or most of the existing nonlinear observers) can not be directly applied for projections that do not maintain an affine form. In [22], Ma et.al. suggested that range identification could be achieved for a nonaffine projection from a paraboloid image surface using a linear approximation-based observer where the state estimation of the original nonlinear system is carried out by a sequence of approximate linear time-varying observers originally proposed in [21]. The vision system under consideration, as described in [22] assumes that the focal point and the vertex of the paraboloid imaging surface coincide to produce the nonaffine transformation. However, in this result, the physical construction issue is disregarded, and focus is on the range and motion identification problem via observation on a paraboloid imaging surface.

The contribution of the result in this chapter is the development of a nonlinear estimator that can be used to identify range information and Euclidean coordinates from a nonaffine projection resulting from the paraboloid image system described in [22], without the use of linear approximations. A Lyapunov-based analysis is used to prove the range and the 3D coordinates of an object moving with general affine Euclidean motion (and nonaffine image dynamics) are asymptotically identified.

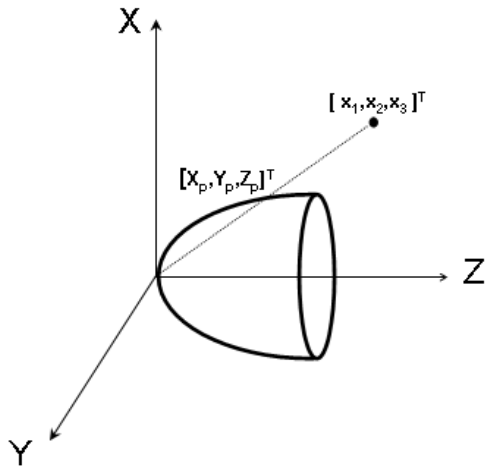

Figure 3–1: A paraboloid imaging surface centered at $[0,0,0]^T$.

Numerical simulation results are provided that illustrate the performance of the estimator.

3.2 Nonaffine Projection

The projection of the Euclidean coordinates onto a paraboloid imaging surface (see Figure. 3–1) is defined as follows:

$$X_p = x_1 x_3 / L, \quad Y_p = x_2 x_3 / L, \quad Z_p = x_3^2 / L \tag{3–1}$$

where $X_p(t)$, $Y_p(t)$, $Z_p(t) \in \mathbb{R}$ denote the projected coordinates of $x_1(t)$, $x_2(t)$, $x_3(t)$, and $L \in \mathbb{R}$ is defined as

$$L = x_1^2 + x_2^2. \tag{3–2}$$

The image-space coordinates, denoted by $y(t) \in \mathbb{R}^4$, can then be defined as follows:

$$\begin{bmatrix} y_1 & y_2 & y_3 & y_4 \end{bmatrix}^T = \begin{bmatrix} X_p & Y_p & Z_p & 1/L \end{bmatrix}^T \tag{3–3}$$

where $y_1(t)$ and $y_2(t)$ are the measured pixel coordinates from the camera, $y_3(t)$ is computed from the measured pixel coordinates as

$$y_3 = y_1^2 + y_2^2, \tag{3–4}$$

and $y_4(t)$ contains the unknown range information [22]. The projection onto the paraboloid can be expressed by the following nonaffine perspective dynamic system (PDS) [22]:

$$\begin{aligned}\dot{y}_1 &= a_{11}y_1 + a_{12}y_2 + a_{13}y_3 + a_{31}\frac{y_1^2}{y_3} + a_{32}\frac{y_1y_2}{y_3} + a_{33}y_1 - 2a_{11}\frac{y_1^3}{y_3} \\ &- 2(a_{12}+a_{21})\frac{y_1^2y_2}{y_3} - 2a_{22}\frac{y_1y_2^2}{y_3} - 2a_{13}y_1^2 - 2a_{23}y_1y_2 + f_1\end{aligned} \tag{3–5}$$

$$\begin{aligned}\dot{y}_2 &= a_{21}y_1 + a_{22}y_2 + a_{23}y_3 + a_{31}\frac{y_1y_2}{y_3} + a_{32}\frac{y_2^2}{y_3} + a_{33}y_2 - 2a_{11}\frac{y_1^2y_2}{y_3} \\ &- 2(a_{12}+a_{21})\frac{y_1y_2^2}{y_3} - 2a_{22}\frac{y_2^3}{y_3} - 2a_{13}y_1y_2 - 2a_{23}y_2^2 + f_2\end{aligned} \tag{3–6}$$

$$\begin{aligned}\dot{y}_3 &= 2a_{31}y_1 + 2a_{32}y_2 + 2a_{33}y_3 - 2a_{11}y_1^2 - 2a_{22}y_2^2 \\ &- 2(a_{12}+a_{21})y_1y_2 - 2a_{13}y_1y_3 - 2a_{23}y_2y_3 + f_3\end{aligned} \tag{3–7}$$

$$\begin{aligned}\dot{y}_4 &= -\frac{2a_{11}y_1^2y_4}{y_3} - 2a_{23}y_2y_4 - 2b_1y_4y_5 - \frac{2a_{22}y_2^2y_4}{y_3} \\ &- 2a_{13}y_1y_4 - 2(a_{12}+a_{21})\frac{y_1y_2y_4}{y_3} - 2b_2y_4y_6\end{aligned} \tag{3–8}$$

where $f_1(y_1, y_5, y_6, y_7),\ f_2(y_2, y_5, y_6, y_7),\ f_3(y_3, y_5, y_6, y_7) \in \mathbb{R}$ are unmeasurable signals defined as

$$f \triangleq \begin{bmatrix} f_1 & f_2 & f_3 \end{bmatrix}^T = \Omega_1 \begin{bmatrix} y_5 & y_6 & y_7 \end{bmatrix}^T. \tag{3–9}$$

In (3–9), $\Omega_1(y_1, y_2, y_3) \in \mathbb{R}^{3\times 3}$ denotes the following matrix of known and measurable signals:

$$\Omega_1 \triangleq \begin{bmatrix} (b_3 - 2b_1y_1) & -2b_2y_1 & b_1 \\ -2b_1y_2 & (b_3 - 2b_2y_2) & b_2 \\ -2b_1y_3 & -2b_2y_3 & 2b_3 \end{bmatrix}, \tag{3–10}$$

and the unmeasurable auxiliary signals $y_5(y_1, y_3, y_4)$, $y_6(y_2, y_3, y_4)$, $y_7(y_3, y_4) \in \mathbb{R}$ are defined as

$$\begin{bmatrix} y_5 & y_6 & y_7 \end{bmatrix}^T = \frac{1}{L}\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}^T \tag{3–11}$$
$$= \begin{bmatrix} y_1\sqrt{\frac{y_4}{y_3}} & y_2\sqrt{\frac{y_4}{y_3}} & \sqrt{y_3 y_4} \end{bmatrix}^T.$$

To facilitate the subsequent development, the PDS in (3–5) through (3–7) can be rewritten as

$$\begin{bmatrix} \dot{y}_1 & \dot{y}_2 & \dot{y}_3 \end{bmatrix}^T = \Omega_2 + f \tag{3–12}$$

where $\Omega_2(y_1, y_2, y_3) \in \mathbb{R}^3$ denotes a vector of measurable and known signals.

Assumption 3-1: The image-space feature coordinates $y_1(t)$, $y_2(t)$ and $y_3(t)$ are bounded functions of time.

Assumption 3-2: The object feature motion avoids the degenerate case where the feature intersects the image plane (i.e., $x_3(t) \neq 0$).

Assumption 3-3: The object feature does not collide with the imaging surface (i.e., $L > Z_p$). Hence, from (3–1) and (3–3) $y_4(t) \in \mathcal{L}_\infty$. Moreover, the object feature is not a vanishing point since $\mathcal{L} \in \mathcal{L}_\infty$.

Assumption 3-4: The matrix $\Omega_1(y_1, y_2, y_3)$ introduced in (3–10) is invertible provided

$$b_3 \neq 0 \tag{3–13}$$

and

$$b_3^2 - 2b_2b_3y_2 + b_2^2y_3 - 2b_1b_3y_1 + b_1^2y_3 \neq 0. \tag{3–14}$$

After utilizing (3–4), the condition in (3–14) can be written as

$$(y_1 - k_1)^2 + (y_2 - k_2)^2 \neq 0 \tag{3–15}$$

where $k_1(t)$ and $k_2(t) \in \mathbb{R}$ are auxiliary terms defined as

$$k_1 = \frac{b_1b_3}{b_1^2 + b_2^2} \qquad k_2 = \frac{b_2b_3}{b_1^2 + b_2^2} \tag{3–16}$$

Geometrically, the observability condition in (3–15) indicates that the projection of the object feature cannot intersect the point (k_1, k_2). For the special case when $b_1 = b_2 = 0$, (3–14) reduces to (3–13). Based on (3–15) and (3–16) the observability condition in (3–13) is equivalent to the physical property given in Assumption 3-3. That is if $b_3 = 0$, then (3–1) through (3–4) indicate that $x_3(t) = 0$.

Remark 3-1: Assumptions 3-1 through 3-4 are standard assumptions that are practical properties of the physical system rather than assumptions.

Remark 3-2: Based on Assumptions 3-1 through 3-4, the expressions given in (3–5) through (3–8) and (3–19) can be used to determine that $\dot{y}_i(t)$, $\Omega_1(y_1, y_2, y_3)$, and $f(y_1, y_2, y_3, y_5, y_6, y_7) \in \mathcal{L}_\infty$ $\forall i = 1, 2, ...4$. Given that these signals are bounded, Assumptions 3-1 through 3-4 can be used to determine that $\dot{f}(\cdot)$ and $\ddot{f}(\cdot) \in \mathcal{L}_\infty$.

3.3 Range and Motion Identification

3.3.1 Objective

The objective in this paper is to identify the unmeasurable state $y_4(t)$ of the PDS described by (3–5) through (3–12). From (3–11) and the fact that $y_1(t)$, $y_2(t)$ and $y_3(t)$ are measurable, if $y_4(t)$ could be identified then the complete Euclidean coordinates of the feature can be determined. To achieve this objective, an estimator is constructed based on the unmeasurable image-space dynamics for $y(t)$. To quantify the objective, a measurable estimation error, denoted by $e(t) \in \mathbb{R}^3$, is defined as follows:

$$e \triangleq \begin{bmatrix} e_1 & e_2 & e_3 \end{bmatrix}^T = \begin{bmatrix} y_1 - \hat{y}_1 & y_2 - \hat{y}_2 & y_3 - \hat{y}_3 \end{bmatrix}^T \tag{3–17}$$

where $\hat{y}_i(t) \in \mathbb{R}\ \forall i = 1, 2, 3$ denotes a subsequently designed estimator. An unmeasurable[1] filtered estimation error, denoted by $r(t) \in \mathbb{R}^3$, is also defined as

$$r \triangleq \begin{bmatrix} r_1 & r_2 & r_3 \end{bmatrix}^T = \dot{e} + \alpha e \tag{3–18}$$

where $\alpha \in \mathbb{R}^{3\times 3}$ denotes a diagonal matrix of positive constant gains.

The error systems in (3–17) and (3–18) are defined based on the goal to prove that the PDS in (3–5) through (3–7) can be identified (i.e., that $f(y_1, y_2, y_3, y_5, y_6, y_7)$ can be identified). If $f(y_1, y_2, y_3, y_5, y_6, y_7)$ can be identified, and assuming the conditions in (3–13) and (3–14) are satisfied, then (3–9) can be used to identify $y_5(y_1, y_3, y_4)$, $y_6(y_2, y_3, y_4)$, and $y_7(y_3, y_4)$. If $y_5(y_1, y_3, y_4)$, $y_6(y_2, y_3, y_4)$, and $y_7(y_3, y_4)$ can be identified, then (3–11) can be used to compute $y_4(t)$ as follows:

$$y_4 = \frac{y_3(y_5^2 + y_6^2 + y_7^2)}{y_1^2 + y_2^2 + y_3^2}. \tag{3–19}$$

3.3.2 Estimator Design and Error System

Based on the PDS in (3–12), the estimation objective, and the subsequent analysis, the following estimation signals are defined

$$\begin{bmatrix} \dot{\hat{y}}_1 & \dot{\hat{y}}_2 & \dot{\hat{y}}_3 \end{bmatrix}^T = \Omega_2 + \hat{f}(t) \tag{3–20}$$

where $\hat{f}(t) \triangleq [\hat{f}_1(t),\ \hat{f}_2(t),\ \hat{f}_3(t)]^T \in \mathbb{R}^3$ denotes a subsequently designed estimate for $f(y_1, y_2, y_3, y_5, y_6, y_7)$ introduced in (3–9). The following error dynamics are obtained after taking the time derivative of $e(t)$ and utilizing (3–12) and (3–20):

$$\dot{e} = f - \hat{f}. \tag{3–21}$$

[1] The filtered estimation signal is unmeasurable due to a dependence on the unmeasurable terms $f_1(y_1,\ y_5,\ y_6,\ y_7)$, $f_2(y_2, y_5, y_6, y_7)$, $f_3(y_3, y_5, y_6, y_7)$.

The time-derivative of (3–18) can be expressed as

$$\dot{r} = \dot{f} - \dot{\hat{f}} + \alpha \left(f - \hat{f} \right) \tag{3–22}$$

after utilizing (3–21) and the time derivative of (3–21). Based on the structure of (3–22) and the subsequent analysis, $\hat{f}(t)$ is designed as follows [8]:

$$\dot{\hat{f}} = -(k_s + \alpha)\hat{f} + \gamma sgn(e) + \alpha k_s e \tag{3–23}$$

where k_s, $\gamma \in \mathbb{R}^{3\times 3}$ denote diagonal matrices of positive constant estimation gains, and the notation $sgn(\cdot)$ is used to indicate a vector with the standard signum function applied to each element of the argument. After substituting (3–23) into (3–22) and then adding and subtracting the terms $k_s f(y_1, y_2, y_3, y_5, y_6, y_7)$, the following expression can be obtained:

$$\dot{r} = \eta - k_s r - \gamma sgn(e) \tag{3–24}$$

where $\eta(t) \triangleq \begin{bmatrix} \eta_1 & \eta_2 & \eta_3 \end{bmatrix}^T \in \mathbb{R}^3$ is defined as

$$\eta \triangleq \dot{f} + (k_s + \alpha) f. \tag{3–25}$$

Remark 3-3: The structure of the estimator in (3–23) contains discontinuous terms; however, as discussed in [8], the overall structure of the estimator is continuous (i.e., $\hat{f}(t)$ is continuous).

Remark 3-4: Once $y_4(t)$ is identified, the complete 3D Euclidean coordinates of the object feature can be determined using (3–3) and (3–11). Provided the observability conditions given in (3–13) and (3–14) are satisfied $y_4(t)$ can be identified if $\hat{f}(t)$ approaches $f(t)$ as $t \to \infty$ (i.e., $\hat{y}_1(t)$, $\hat{y}_2(t)$ and $\hat{y}_3(t)$ approach $y_1(t)$, $y_2(t)$ and $y_3(t)$ as $t \to \infty$) since the parameters $b_i(t)$ $\forall i = 1, 2, 3$ are assumed to be known, and $y_1(t)$, $y_2(t)$ and $y_3(t)$ are measurable. To prove that $\hat{f}(t)$

approaches $f(t)$ as $t \to \infty$, the subsequent development will focus on proving that $\|\dot{e}(t)\| \to 0$ and $\|e(t)\| \to 0$ as $t \to \infty$ based on (3–17) and (3–21).

3.4 Analysis

The following theorem and associated proof can be used to conclude that the observer design of (3–20) and (3–23) can be used to identify the unmeasurable state $y_4(t)$ (i.e., the object feature range and motion can be determined) provided the observability conditions in (3–13) and (3–14) are satisfied.

Theorem 3-1: Given the PDS in (3–5) through (3–11), the unmeasurable state $y_4(t)$ (and hence, the Euclidean coordinates of the object feature) can be asymptotically determined from the estimator in (3–20) and (3–23) provided the elements of the constant diagonal matrix γ introduced in (3–23) are selected according to the sufficient conditions

$$\begin{gathered} \gamma_1 \geq |\eta_1| + |\dot{\eta}_1| \qquad \gamma_2 \geq |\eta_2| + |\dot{\eta}_2| \\ \gamma_3 \geq |\eta_3| + |\dot{\eta}_3| \end{gathered} \tag{3–26}$$

where $\eta(t)$ is defined in (3–25), and the observability conditions introduced in (3–13) and (3–14) are satisfied.

Proof: Consider a non-negative function $V(t) \in \mathbb{R}$ as follows (i.e., a Lyapunov function candidate):

$$V \triangleq \frac{1}{2} r^T r \ . \tag{3–27}$$

After taking the time derivative of (3–27) and substituting for the error system dynamics given in (3–24), the following expression can be obtained:

$$\dot{V} = -r^T k_s r + (\dot{e} + \alpha e)^T (\eta - \gamma sgn(e)) \, . \tag{3–28}$$

After integrating (3–28) and exploiting the fact that

$$\xi_i \cdot sgn(\xi_i) = |\xi_i| \ \ \forall \xi_i \in \mathbb{R},$$

the following inequality can be obtained:

$$V(t) \leq V(t_0) - \int_{t_0}^{t} \left(r^T(\sigma) k_s r(\sigma)\right) d\sigma + \alpha_1 \int_{t_0}^{t} |e_1(\sigma)| \left(|\eta_1(\sigma)| - \gamma_1\right) d\sigma + \chi_1 \quad (3\text{–}29)$$
$$+ \alpha_2 \int_{t_0}^{t} |e_2(\sigma)| \left(|\eta_2(\sigma)| - \gamma_2\right) d\sigma + \chi_2 + \alpha_3 \int_{t_0}^{t} |e_3(\sigma)| \left(|\eta_3(\sigma)| - \gamma_3\right) d\sigma + \chi_3$$

where the auxiliary terms $\chi_1(t)$, $\chi_2(t)$, $\chi_3(t) \in \mathbb{R}$ are defined as

$$\chi_1 \triangleq \int_{t_0}^{t} \dot{e}_1(\sigma) \eta_1(\sigma) d\sigma - \gamma_1 \int_{t_0}^{t} \dot{e}_1(\sigma) sgn(e_1(\sigma)) d\sigma \quad (3\text{–}30)$$

$$\chi_2 \triangleq \int_{t_0}^{t} \dot{e}_2(\sigma) \eta_2(\sigma) d\sigma - \gamma_2 \int_{t_0}^{t} \dot{e}_2(\sigma) sgn(e_2(\sigma)) d\sigma \quad (3\text{–}31)$$

$$\chi_3 \triangleq \int_{t_0}^{t} \dot{e}_3(\sigma) \eta_3(\sigma) d\sigma - \gamma_3 \int_{t_0}^{t} \dot{e}_3(\sigma) sgn(e_3(\sigma)) d\sigma. \quad (3\text{–}32)$$

The integral expressions in (3–30) through (3–32) can be evaluated as

$$\chi_1 = e_1(\sigma) \eta_1(\sigma)_{t_0}^{t} d\sigma - \int_{t_0}^{t} e_1(\sigma) \dot{\eta}_1(\sigma) d\sigma - \gamma_1 |e_1(\sigma)|_{t_0}^{t} \quad (3\text{–}33)$$
$$= e_1(t) \eta_1(t) - \int_{t_0}^{t} e_1(\sigma) \dot{\eta}_1(\sigma) d\sigma - \gamma_1 |e_1(t)| - e_1(t_0) \eta_1(t_0) + \gamma_1 |e_1(t_0)|$$

$$\chi_2 = e_2(t) \eta_2(t) - \int_{t_0}^{t} e_2(\sigma) \dot{\eta}_2(\sigma) d\sigma - \gamma_2 |e_2(t)| - e_2(t_0) \eta_2(t_0) + \gamma_2 |e_2(t_0)| \quad (3\text{–}34)$$

$$\chi_3 = e_3(t) \eta_3(t) - \int_{t_0}^{t} e_3(\sigma) \dot{\eta}_3(\sigma) d\sigma - \gamma_3 |e_3(t)| - e_3(t_0) \eta_3(t_0) + \gamma_3 |e_3(t_0)|. \quad (3\text{–}35)$$

Substituting (3–33) through (3–35) into (3–29) and performing some algebraic manipulation yields

$$V(t) \leq V(t_0) - \int_{t_0}^{t} \left(r^T(\sigma) k_s r(\sigma)\right) d\sigma + \chi_4 + \zeta_0$$

where the auxiliary terms $\chi_4(t), \zeta_0 \in \mathbb{R}$ are defined as

$$\chi_4 \triangleq \alpha_1 \int_{t_0}^{t} |e_1(\sigma)| \left(|\eta_1(\sigma)| + |\dot{\eta}_1(\sigma)| - \gamma_1\right) d\sigma \quad (3\text{–}36)$$
$$+ \alpha_2 \int_{t_0}^{t} |e_2(\sigma)| \left(|\eta_2(\sigma)| + |\dot{\eta}_2(\sigma)| - \gamma_2\right) d\sigma + \alpha_3 \int_{t_0}^{t} |e_3(\sigma)| \left(|\eta_3(\sigma)| + |\dot{\eta}_3(\sigma)| - \gamma_3\right) d\sigma$$
$$+ |e_1(t)| \left(|\eta_1(t)| - \gamma_1\right) + |e_2(t)| \left(|\eta_2(t)| - \gamma_2\right) + |e_3(t)| \left(|\eta_3(t)| - \gamma_3\right)$$

$$\zeta_0 \triangleq -e_1(t_0)\,\eta_1(t_0) + \gamma_1\,|e_1(t_0)| \tag{3–37}$$
$$-\,e_2(t_0)\,\eta_2(t_0) + \gamma_2\,|e_2(t_0)| - e_3(t_0)\,\eta_3(t_0) + \gamma_3\,|e_3(t_0)|\,.$$

Provided the diagonal entries of γ are selected according to the inequalities introduced in (3–26), $\chi_4(t)$ will always be negative or zero; hence, the following upper bound can be developed:

$$V(t) \leq V(t_0) - \int_{t_0}^{t} \left(r^T(\sigma)\,k_s r(\sigma)\right) d\sigma + \zeta_0\ . \tag{3–38}$$

From (3–27) and (3–38), the following inequalities can be determined:

$$V(t_0) + \zeta_0 \geq V(t) \geq 0;$$

hence, $r(t) \in \mathcal{L}_\infty$. The expression in (3–38) can be used to determine that

$$\int_{t_0}^{t} \left(r^T(\sigma)\,k_s r(\sigma)\right) d\sigma \leq V(t_0) + \zeta_0 < \infty\ . \tag{3–39}$$

By definition, (3–39) can now be used to prove that $r(t) \in \mathcal{L}_2$. From the fact that $r(t) \in \mathcal{L}_\infty$, (3–17) and (3–18) can be used to prove that $e(t)$, $\dot{e}(t)$, $\hat{y}(t)$, and $\dot{\hat{y}}(t) \in \mathcal{L}_\infty$. The expressions in (3–20) and (3–23) can be used to determine that $\hat{f}(t)$ and $\dot{\hat{f}}(t) \in \mathcal{L}_\infty$. Assumptions 3-1 through 3-4 can be used to determine that $\dot{f}(\cdot)$ and $\ddot{f}(\cdot) \in \mathcal{L}_\infty$. Based on the facts that $f(\cdot)$, $\dot{f}(\cdot) \in \mathcal{L}_\infty$, the expressions in (3–24) and (3–25) can be used to prove that $\eta(t)$, $\dot{\eta}(t)$, $\dot{r}(t) \in \mathcal{L}_\infty$. Based on the fact that $r(t)$, $\dot{r}(t) \in \mathcal{L}_\infty$ and that $r(t) \in \mathcal{L}_2$, Barbalat's Lemma [27] can be used to prove that $\|r(t)\| \to 0$ as $t \to \infty$; hence, Lemma 1.6 of [7] can be used to prove that $\|e(t)\| \to 0$ and $\|\dot{e}(t)\| \to 0$ as $t \to \infty$.

Based on the fact that $\|e(t)\| \to 0$ and $\|\dot{e}(t)\| \to 0$ as $t \to \infty$, the expression given in (3–17) can be used to determine that $\hat{y}_1(t)$, $\hat{y}_2(t)$ and $\hat{y}_3(t)$ approach $y_1(t)$, $y_2(t)$ and $y_3(t)$ as $t \to \infty$, respectively. Therefore, the expression in (3–21) can be used to determine that $\hat{f}$ approaches f as $t \to \infty$. If the observability conditions

given in (3–13) and (3–14) are satisfied, then the result that $\hat{f}(t)$ approaches $f(t)$ as $t \to \infty$, the fact that the parameters $b_i(t)$ $\forall i = 1, 2, 3$ are assumed to be known, and the fact that the image-space signals $y_1(t)$, $y_2(t)$ and $y_3(t)$ are measurable can be used to identify the unknown Euclidean parameter $y_4(t)$ from (3–19). Once $y_4(t)$ is identified, the complete Euclidean coordinates of the object feature can be determined using (3–9) through (3–11).

3.5 Numerical Simulation

In this section, numerical simulation results are provided to illustrate the performance of the range identification observer given a paraboloid imaging surface. The object feature is governed by the following affine motion dynamics:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix} = \begin{bmatrix} -0.2 & 0.4 & -0.6 \\ 0.1 & -0.2 & 0.3 \\ 0.3 & -0.4 & 0.4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 0.5 & 0.25 & 0.3 \end{bmatrix}^T$$

with the following initial Euclidean coordinates

$$\begin{bmatrix} x_1(0) & x_2(0) & x_3(0) \end{bmatrix}^T = \begin{bmatrix} 0.4 & 0.6 & 1 \end{bmatrix}^T.$$

Based on (3–1), the relationship in (3–1) through (3–3) can be used to determine the following initial conditions in the image-space

$$\hat{y}_1(t_0) = y_1(t_0) = 0.769$$
$$\hat{y}_2(t_0) = y_2(t_0) = 1.150$$
$$\hat{y}_3(t_0) = y_3(t_0) = 1.920$$
$$y_4(t_0) = 1.920$$

The estimates for $f(t)$ were initialized as follows:

$$\hat{f}_1(t_0) = 1 \quad \hat{f}_2(t_0) = 1 \quad \hat{f}_3(t_0) = 1.$$

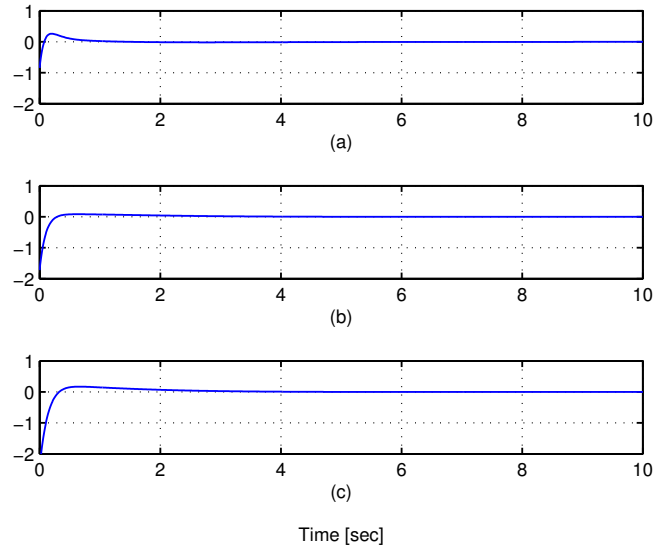

Figure 3–2: Estimation error of auxiliary signals (a) $\dot{e}_1(t)$ (b) $\dot{e}_2(t)$ and (c) $\dot{e}_3(t)$ in [pixels/sec].

After adjusting the observer gains as

$$k_s = diag\{10, 10, 10\} \qquad \alpha = diag\{8, 8, 8\}$$
$$\gamma = diag\{1, 1, 1\} \times 10^{-5}$$

the resulting mismatch between $f_1(t)$ and $\hat{f}_1(t)$, $f_2(t)$ and $\hat{f}_2(t)$, and $f_3(t)$ and $\hat{f}_3(t)$ (i.e., $\dot{e}_1(t)$, $\dot{e}_2(t)$, and $\dot{e}_3(t)$ respectively) is provided in Figure. 3–2. The mismatch between $y_4(t)$ and $\hat{y}_4(t)$ is provided in Figure. 3–3. The $y_4(t)$ term is obtained from numerical integration of the $\dot{y}_4(t)$ term, while the estimated value is obtained by replacing $f(t)$ with $\hat{f}(t)$ in (3–9), solving for $y_5(t)$, $y_6(t)$, and $y_7(t)$, and then utilizing equation (3–19).

Additive-white-gaussian-noise (AWGN) was injected into the measurable image-space signals $y_1(t)$, $y_2(t)$ via the awgn() function in MATLAB, while maintaining a constant signal-to-noise-ratio of 20. Without changing any of the other simulation parameters, the mismatch between $f_1(t)$ and $\hat{f}_1(t)$, $f_2(t)$ and $\hat{f}_2(t)$, and $f_3(t)$ and $\hat{f}_3(t)$ (i.e., $\dot{e}_1(t)$, $\dot{e}_2(t)$, and $\dot{e}_3(t)$ respectively) is provided in Figure. 3–4, while the mismatch between $y_4(t)$ and $\hat{y}_4(t)$ is provided in Figure. 3–5.

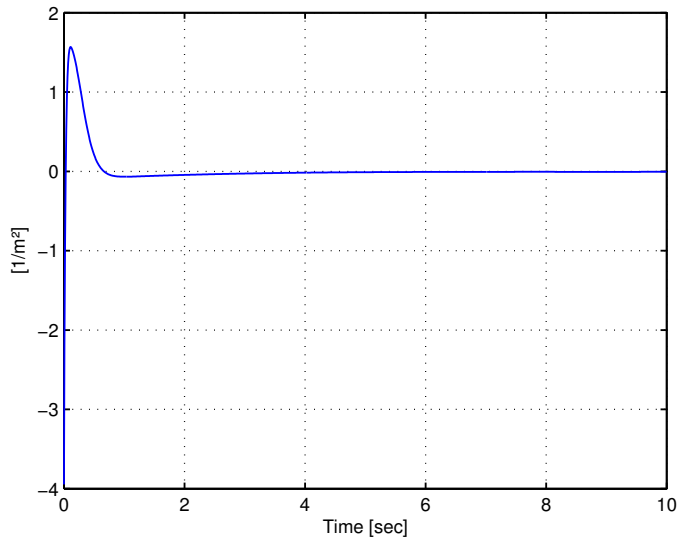

Figure 3–3: Mismatch between $y_4(t)$ and $\hat{y}_4(t)$.

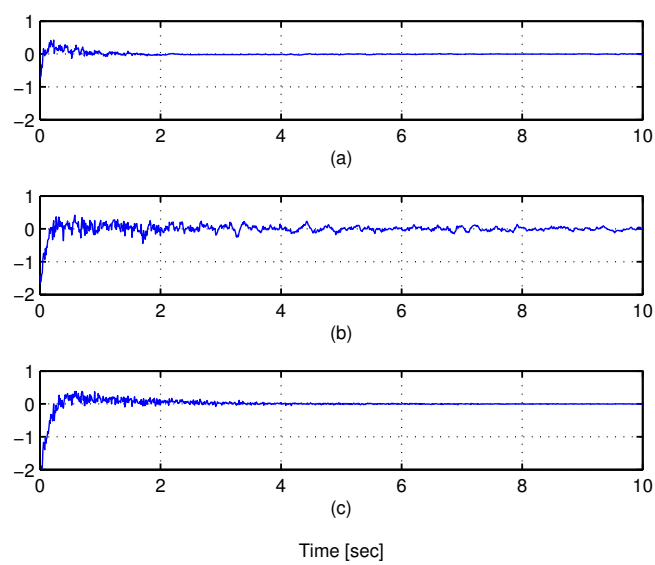

Figure 3–4: Estimation error of auxiliary signals in the presence of noise (a) $\dot{e}_1(t)$ (b) $\dot{e}_2(t)$ and (c) $\dot{e}_3(t)$ in [pixels/sec].

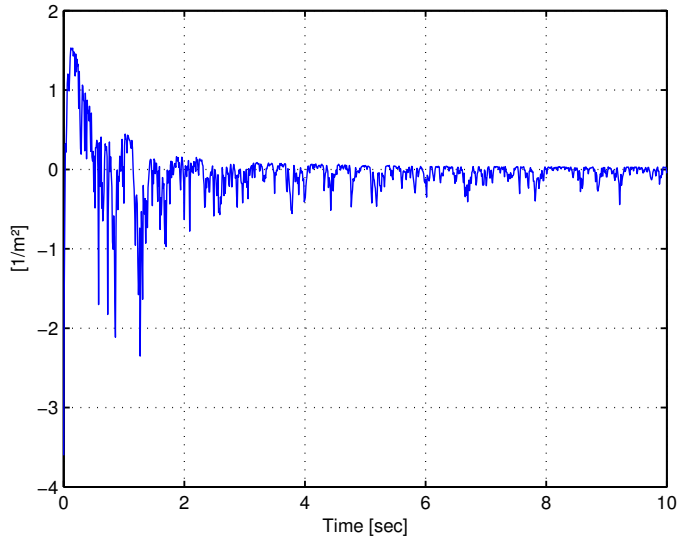

Figure 3–5: Mismatch between $y_4(t)$ and $\hat{y}_4(t)$ in the presence of noise.

The results depicted in Figure. 3–2 through Figure. 3–5 indicate that the proposed observer can be used to identify the range and hence, the Euclidean coordinates of an object feature moving with affine motion dynamics and a nonaffine PDS, provided the observability conditions are satisfied. These results are comparable to the results obtained in [8] which applied the range identification observer to a planar imaging surface.

3.6 Conclusion

The range and the Euclidean coordinates of an object undergoing general affine motion are determined for a paraboloid imaging system. Unlike image systems that are based on a planar image surface (or spherical or ellipsoidal surface), the perspective dynamic system resulting from projected image of a general 3D imaging surface is not guaranteed to maintain an affine form as shown in [22].

A nonlinear state estimator is developed for the nonaffine projection resulting from the paraboloid image surface whose focal point and the vertex coincide (see Figure. 3–1), without the use of linear approximations. The nonlinear estimator

was proven and numerically demonstrated to asymptotically determine the range information from a single camera provided some observability conditions are satisfied and that the Euclidean motion parameters are known. Although the system discussed disregards the physical construction issue, this result indicates that the technique applicable to catadioptric systems (discussed in the previous chapter) is also applicable to general 3D vision systems that do not maintain an affine form.

CHAPTER 4
HARDWARE IN THE LOOP

4.1 Introduction

The hardware-in-the-loop-simulation (HILS) facility at the University of Florida provides a rapid prototyping testbed to simulate vision based guidance, navigation and control applications. The development of the HILS involved a three fold process - hardware setup, software development and integration of their components.

The hardware required for the HILS facility can be subdivided into the following five main components: (1) virtual reality simulator; (2) modular display; (3) supporting workstation; (4) interface; and (5) image processing workstation. Much of the software developed in the HILS is based on image processing and computer vision methods, primarily feature point identification and multi-view photogrammetry technique.

The contributions discussed in this chapter include the real-time identification of features and the implementation of a multi-view photogrammetry technique to facilitate visual servo control applications. An overview of the complete control loop at the HILS facility is illustrated in Figure. 4–1 and described in the following steps:

1. The virtual reality simulator renders the visual environment database on the projector screens (modular display).

2. Images are captured by a camera and passed to the image processing workstation through an analog to digital converter using a firewire.

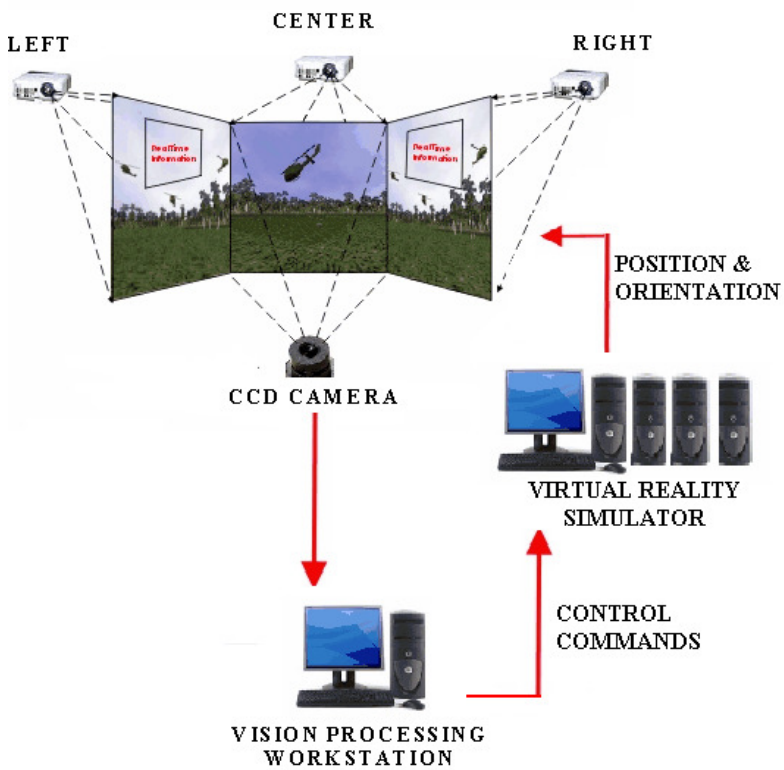

Figure 4–1: Overview of HILS facility.

3. Image processing and computer vision algorithms identify feature points from the images obtained, determine the relationship between current and reference frames and accordingly generate control commands.

4. Sockets are used to communicate the generated control commands into the virtual reality simulator in order to control motion of the virtual environment.

This chapter is organized in the following manner. In Section 4.2, a description of the hardware setup of the HILS facility is presented. In Section 4.3, the image processing algorithm for real-time feature detection and tracking is discussed. Implementation of a multi-view photogrammetry technique is illustrated in Section 4.4. In Section 4.5, the use of sockets for communicating control information between workstations is expressed, and concluding remarks are provided in Section 4.6.

4.2 Hardware Setup

The hardware for the virtual environment consists of the following five main components: (1) virtual reality simulator; (2) modular display; (3) supporting workstation; (4) interface; and (5) image processing workstation. Each component is necessary to construct a virtual environment for supporting visualization and computational technologies.

The first component is the virtual reality simulator which generates the data associated with the virtual environment. The virtual reality simulator utilizes MultiGen-Pardigm-Vega Prime software package, an extensible COTS tool available for the creation and deployment of visual simulation, urban simulation, and general visualization applications. Such generation includes construction of the complete 3D space of the virtual environment. Most importantly, the simulator accounts for the position and orientation to properly describe the image being viewed. The HILS facility is setup to generate the urban visual environment of University of Florida in real-time. The setup includes four computers for rendering the image on the projector screens. One computer is the master (server), and the remaining three act as slaves and are responsible for generating the virtual scene on each of the three projector screens.

The second component is the modular display. The display consists of a set of three screens as depicted in Figure. 4–2, and projectors that render images of the virtual database onto the screens. Each screen presents a different aspect of the virtual environment. The third component is the supporting workstation as illustrated in Figure. 4–3. This workstation consists of machines which allow access to the virtual reality simulator and the displays. The machines are standard off-the-shelf computers running Windows operating systems.

The fourth component is the interface. This interface allows the virtual environment to communicate with external components associated with other facilities.

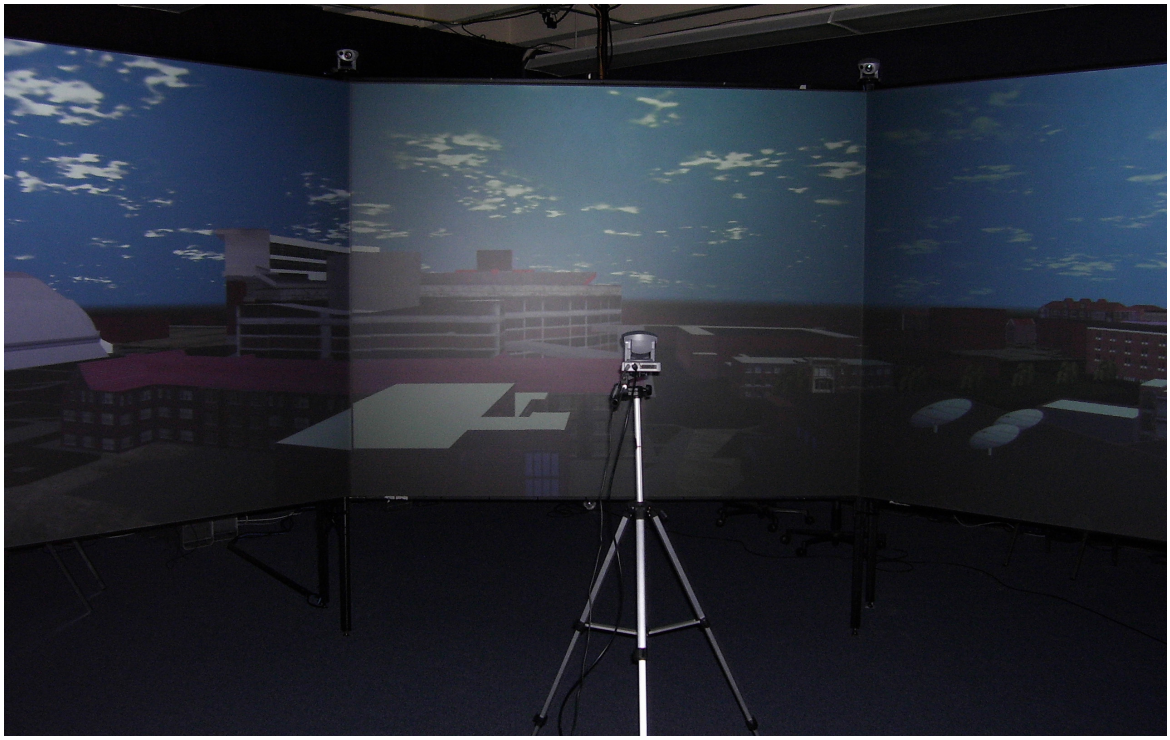

Figure 4–2: Modular display in the HILS facilty at the University of Florida.

Purpose of the interface is to exchange data with other computational facilities and virtual environments so a massive virtual laboratory can be constructed.

The fifth segment consists of the image processing equipment such as the camera setup, image processing workstation and their interfacing equipment. A separate machine running Windows is used to run the image processing algorithms. The camera viewing the display screens is connected to the image processing workstation using an analog to digital converter and a firewire for high speed video transmission.

4.3 Image Processing

Many vision-based estimation and control strategies require real-time detection and tracking of features. The most important segment of the image processing algorithm (see Appendix C) is real-time identification of features required for coordinated guidance, navigation and control in complex 3D surroundings such as urban environments. The algorithm is developed in C/C++ using Visual Studio 6.0

Figure 4–3: Supporting workstation in the HILS facility at the University of Florida.

and Visual Studio .NET. Intel's Open Source Computer Vision Library is utilized for real-time implementation of most of the image processing functions. Also, GNU Scientific Library (GSL) is used, which is a numerical library for C and C++ programming. GSL provides a wide range of mathematical routines for most of the computation involved in the algorithms.

4.3.1 Feature Point Detection and Tracking

As stated in [26], no feature-based vision system can work unless good features can be identified and tracked from frame to frame. Two important issues need to be addressed; a method for selecting good and reliable feature points; and a method to track the feature points frame to frame. These issues are discussed in detail in [29] and [26], where the Kanade-Lucas-Tomasi (KLT) tracker is promoted as a solution.

The algorithm developed for the HILS facility (see Appendix C) is based on the KLT tracker that selects features which are optimal for tracking. The basic principle of the KLT is that a good feature is one that can be tracked well, so tracking should not be separated from feature extraction. As stated in [29], a good feature is a textured patch with high intensity variation in both x and y directions, such as a corner. By representing the intensity function in x and y directions by g_x and g_y, respectively, we can define the local intensity variation matrix as

$$Z = \begin{bmatrix} g_x^2 & g_x g_y \\ g_x g_y & g_y^2 \end{bmatrix}. \tag{4–1}$$

A patch defined by a 7 x 7 pixels window is accepted as a candidate feature if in the center of the window both eigenvalues of Z, exceed a predefined threshold. Two large eigenvalues can represent corners, salt and pepper textures, or any other pattern that can be tracked reliably. Two small eigenvalues mean a roughly constant intensity profile within a window. A large and a small eigenvalue correspond to a

unidirectional texture pattern. The intensity variations in a window are bounded by the maximum allowable pixel value, so the greater eigenvalue cannot be arbitrarily large. In conclusion, if the two eigenvalues of Z are λ_1 and λ_2, we accept a window if

$$min(\lambda_1, \lambda_2) > \lambda \tag{4–2}$$

where λ is a predefined threshold.

As described in [26], feature point tracking is a standard task of computer vision with numerous applications in navigation, motion understanding, surveillance, scene monitoring, and video database management. In an image sequence, moving objects are represented by their feature points detected prior to tracking or during tracking. As the scene moves, the patterns of image intensities changes in a complex way. Denoting images by I, these changes can be described as image motion:

$$I(x, y, t + \tau) = I(x - \zeta(x, y, t, \tau), y - \eta(x, y, t, \tau)). \tag{4–3}$$

Thus a later image taken at time $t + \tau$ (where τ represents a small time interval) can be obtained by moving every point in the current image, taken at time t, by a suitable amount. The amount of motion $\delta = (\zeta, \eta)$ is called the displacement of the point at $m = (x, y)$. The displacement vector δ is a function of image position m. Tomasi in [26], states that pure translation is the best motion model for tracking because it exhibits reliability and accuracy in comparing features between the reference and current image, hence

$$\delta = d$$

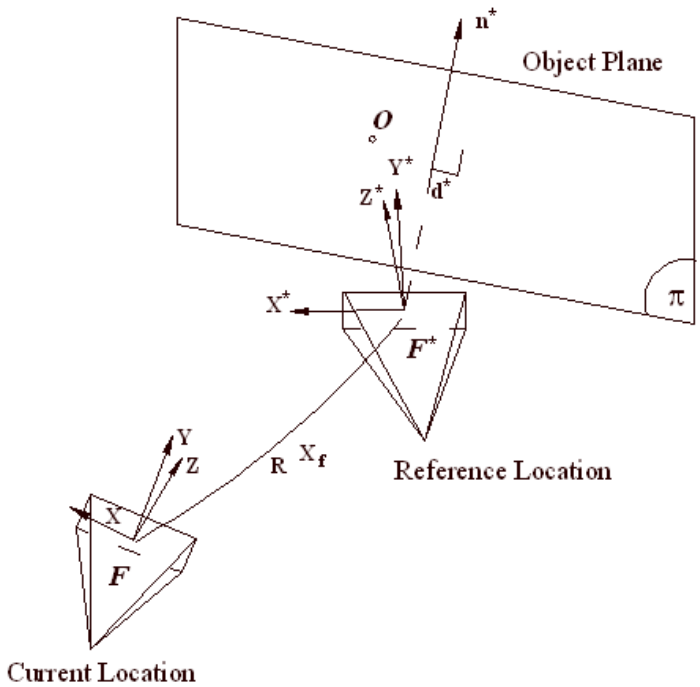

Figure 4–4: Moving camera looking at a reference plane.

where d is the translation of the feature's window center between successive frames. Thus, the knowledge of translation d allows reliable and optimal tracking of the feature points windows.

In the algorithm developed at the HILS facility (see Appendix C), the method employed for tracking is based on the pyramidal implementation of the KLT Tracker described in [4].

4.4 Multi-view Photogrammetry

The feature points data obtained during motion through the virtual environment is used to determine the relationship between the current and a constant reference position as shown in Figure. 4–4. This relationship is obtained by determining the rotation and translation between corresponding feature points on the current and reference image position. The rotation and translation components relating corresponding points of the reference and current image is obtained by first constructing the Euclidean homography matrix. Various techniques can then

be used (e.g., see [10], [33]) to decompose the Euclidean homography matrix into rotational and translational components.

4.4.1 Euclidean Reconstruction

Consider the orthogonal coordinate systems, denoted $\mathcal{F}$ and $\mathcal{F}^*$ that are depicted in Figure. 4–4. The coordinate system $\mathcal{F}$ is attached to a moving camera. A reference plane π on the object is defined by four target points O_i $\forall i = 1, 2, 3, 4$ where the three dimensional ($3D$) coordinates of O_i expressed in terms of $\mathcal{F}$ and $\mathcal{F}^*$ are defined as elements of $\bar{m}_i(t)$ and $\bar{m}_i^* \in \mathbb{R}^3$ and represented by

$$\bar{m}_i(t) \triangleq \begin{bmatrix} x_i(t) & y_i(t) & z_i(t) \end{bmatrix}^T \tag{4–4}$$

$$\bar{m}_i^* \triangleq \begin{bmatrix} x_i^* & y_i^* & z_i^* \end{bmatrix}^T. \tag{4–5}$$

The Euclidean-space is projected onto the image-space, so the normalized coordinates of the targets points $\bar{m}_i(t)$ and $\bar{m}_i^*$ are defined as

$$m_i = \frac{\bar{m}_i}{z_i} = \begin{bmatrix} \frac{x_i}{z_i} & \frac{y_i}{z_i} & 1 \end{bmatrix}^T \tag{4–6}$$

$$m_i^* = \frac{\bar{m}_i^*}{z_i^*} = \begin{bmatrix} \frac{x_i^*}{z_i^*} & \frac{y_i^*}{z_i^*} & 1 \end{bmatrix}^T \tag{4–7}$$

under the standard assumption that $z_i(t)$ and $z_i^* > \varepsilon$, where ε denotes an arbitrarily small positive scalar constant.

Each target point has pixel coordinates that are acquired from the moving camera, expressed in terms of $\mathcal{F}$, denoted by $u_i(t), v_i(t) \in \mathbb{R}$, and are defined as elements of $p_i(t) \in \mathbb{R}^3$ as follows:

$$p_i \triangleq \begin{bmatrix} u_i & v_i & 1 \end{bmatrix}^T. \tag{4–8}$$

The pixel coordinates of the target points at the reference position is expressed in terms of $\mathcal{F}^*$ (denoted by $u_i^*, v_i^* \in \mathbb{R}$) and are defined as elements of $p_i^* \in \mathbb{R}^3$ as

follows:

$$p_i^* \triangleq \begin{bmatrix} u_i^* & v_i^* & 1 \end{bmatrix}^T . \tag{4–9}$$

The pixel coordinates $p_i(t)$ and p_i^* are related by the following global invertible transformation (i.e., the pinhole model) to the normalized task-space coordinates $m_i(t)$ and m_i^* respectively:

$$\begin{aligned} p_i &= Am_i \\ p_i^* &= Am_i^* \end{aligned} \tag{4–10}$$

where A is the intrinsic camera calibration matrix.

The constant distance from the origin of F^* to the object plane π along the unit normal n^* is denoted by $d^* \in R$ and is defined as

$$d^* \triangleq n^{*T}\bar{m}_i^* . \tag{4–11}$$

The coordinate frames $\mathcal{F}$ and $\mathcal{F}^*$ depicted in Figure. 4–4 are attached to the camera and denote the actual and reference locations of the camera. From the geometry between the coordinate frames, $\bar{m}_i^*$ can be related to $\bar{m}_i(t)$ as follows

$$\bar{m}_i = x_f + R\bar{m}_i^*. \tag{4–12}$$

In (4–12), $R(t) \in SO(3)$ denotes the rotation between $\mathcal{F}$ and $\mathcal{F}^*$, and $x_f(t) \in \mathbb{R}^3$ denotes the translation vector from $\mathcal{F}$ to $\mathcal{F}^*$ expressed in the coordinate frame $\mathcal{F}$. By utilizing (4–6), (4–7), and (4–11), the expressions in (4–12) can be written as follows

$$m_i = \alpha_i \underbrace{\left(R + x_h n^{*T}\right)}_{H} m_i^*. \tag{4–13}$$

In (4–13), $x_h(t) \in \mathbb{R}^3$ denotes the following scaled translation vector

$$x_h = \frac{x_f}{d^*} \tag{4–14}$$

and $\alpha_i(t)$ denotes the depth ratio defined as

$$\alpha_i = \frac{z_i^*}{z_i}. \tag{4–15}$$

After substituting (4–10) into (4–13), the following relationships can be developed

$$p_i = \alpha_i \underbrace{\left(AHA^{-1}\right)}_{G} p_i^* \tag{4–16}$$

where $G(t) = [g_{ij}(t)], \forall i, j = 1, 2, 3 \in \mathbb{R}^{3\times 3}$ denotes a projective homography matrix. After normalizing $G(t)$ by $g_{33}(t)$, which is assumed to be non-zero without loss of generality, the projective relationship in (4–16) can be expressed as follows:

$$p_i = \alpha_i g_{33} Gn p_i^* \tag{4–17}$$

where $Gn \in \mathbb{R}^3$ denotes the normalized projective homography. From (4–17), a set of 12 linearly independent equations given by the 4 target point pairs $(p_i^*, p_i(t))$ with 3 independent equations per target pair can be used to determine $G(t)$ and $\alpha_i(t)g_{33}(t)$. Based on the fact that intrinsic camera calibration matrix A is assumed to be known, (4–16) and (4–17) can be used to determine $g_{33}(t)H(t)$. Various techniques can then be used (e.g., see [10, 33]) to decompose the product $g_{33}(t)H(t)$ into rotational and translational components. Specifically, the scale factor $g_{33}(t)$, the rotation vector $R(t)$, the unit normal vector n^*, and the scaled translation vector denoted by $x_h(t)$ can all be computed from the decomposition of the product $g_{33}(t)H(t)$. Since the product $\alpha_i(t)g_{33}(t)$ can be computed from (4–17), and $g_{33}(t)$ can be determined through the decomposition of the product $g_{33}(t)H(t)$, the depth ration $\alpha_i(t)$ can also be computed.

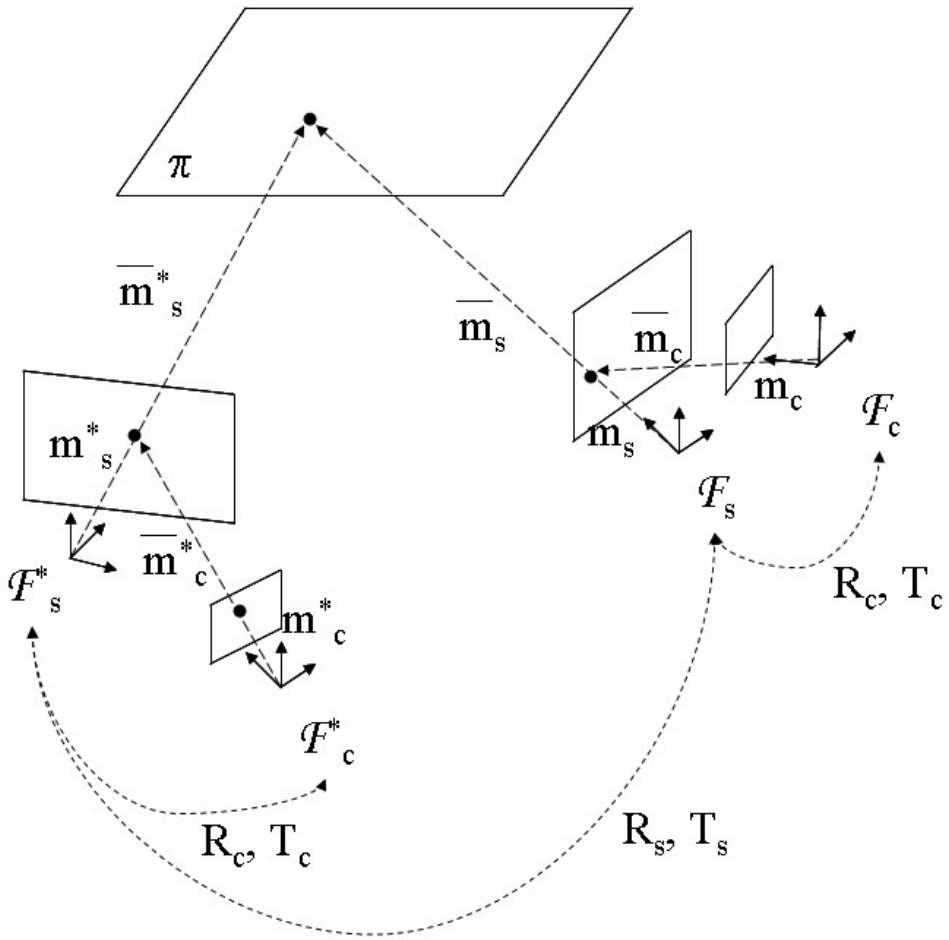

Figure 4–5: Camera, projector plane and virtual scene geometry.

4.4.2 Camera, Screen, and Virtual Scene Geometry

4.4.2.1 Problem Statement

In the HILS facility, the camera is viewing a screen onto which a virtual scene is projected. The camera processes images of the screen in order to control the motion of the virtual scene using the multi-view geometry technique discussed in the previous section. However the multi-view photogrammetry technique cannot be directly applied for the HILS system. This is due to the fact that a homography exists between the *on screen* current image and the *on screen* goal image, but what we are given is the *camera views* of the on screen images instead. Thus, there exists an additional homography between the points on the screen and the points on the camera image. Determining this additional homography allows us to recover the homography between the current and goal on screen images.

The HILS is modelled as if the virtual environment is a true 3D scene which the physical camera is viewing. However, the camera does not look at the 3D scene directly. The camera views points of 3D objects that are projected onto a 2D plane. Geometry between the projector, camera and virtual scene is illustrated in Figure. 4–5. The camera is rigidly connected to the projection plane and any change of scene on the projector plane can be modelled as the projector plane moving through the virtual environment.

4.4.2.2 Virtual Scene Geometry

The virtual environment is projected onto a virtual image plane which is reproduced on a physical display screen. We model this projection of the virtual world onto the screen by the pinhole projection model with some focal point f_s. Reference frame $\mathcal{F}_s^*$ is attached at f_s which is typical for camera projection as seen in Figure. 4–5. We refer to $\mathcal{F}_s^*$ as the *goal frame*. A point p_s lies on the virtual camera's image plane π_s having coordinates $\bar{m}_s^*$ for the virtual camera's goal frame. Motion of the virtual scene results in the virtual camera undergoing a virtual

rotation R_s and translation T_s to a new pose $\mathcal{F}_s$. Hence p_s now has coordinates $\bar{m}_s$ in the virtual camera's new frame $\mathcal{F}_s$.

The relationship between the goal and new frame is expressed as

$$\bar{m}_s = R_s \bar{m}_s^* + T_s. \tag{4–18}$$

Denoting d_s^* as the distance between the virtual camera and π, and n_s^* as the unit vector normal to π in the camera frame, we obtain the following relationship:

$$d_s^* = n_s^{*T} \bar{m}_s. \tag{4–19}$$

After combining (4–18) and (4–19) we can express $\bar{m}_s$ as

$$\bar{m}_s = R_s \bar{m}_s^* + \frac{T_s}{d_s^*}(n_s^{*T} \bar{m}_s^*) \tag{4–20}$$

$$\bar{m}_s = (R_s + \frac{T_s}{d_s^*} n_s^{*T}) \bar{m}_s^*. \tag{4–21}$$

We substitute T_{sd}^* and normalize $\bar{m}_s^*$ and $\bar{m}_s$ by the following equations:

$$m_s = \frac{\bar{m}_s}{z_s}; \quad m_s^* = \frac{\bar{m}_s^*}{z_s^*}\ ; \quad T_{sd}^* = \frac{T_s}{d_s^*}. \tag{4–22}$$

Using (4–21) and (4–22), we can rewrite $\bar{m}_s$ as:

$$m_s = \frac{\bar{z}_s^*}{z_s}(R_s + T_{sd}^* n_s^{*T}) m_s^*$$

$$m_s = \frac{\bar{z}_s^*}{z_s} H_s m_s^* \tag{4–23}$$

where $H_s = (R_s^* + T_{sd}^* n_s^{*T})$. H_s is the *virtual Euclidean homography matrix* which can be determined using (4–23). Hence, the virtual screens rotation and translation information (i.e., R_s, T_{sd}^* and n_s^*) can be recovered through the Faugeras decomposition method described in [9].

4.4.2.3 Camera Geometry

A physical camera is looking at a screen onto which a 2D image is projected from 3D virtual scene; hence, there exists an additional homography between the points on the screen and the points on the camera image. Recovering this homography allows us to recover the homography between the current and goal virtual scene images. As seen in Figure. 4–5, a reference frame $\mathcal{F}_c$ is attached to the camera at the focal point. A point on the screen represented by p_s has coordinates $\bar{m}_c^*$ in the real camera's reference frame. A homography exists between any two planes that provides a bijective map. Thus, there is a constant homography between the real camera image plane and the screen. The *camera-to-screen Euclidean homography* matrix is represented by $H_{cs} \in \mathbb{R}^{3\times 3}$ is and it maps $\bar{m}_c$ to $\bar{m}_s$ by the following equations:

$$\bar{m}_c^* = H_{cs}\bar{m}_s^* \quad \text{and} \quad \bar{m}_c = H_{cs}\bar{m}_s. \tag{4–24}$$

The normalized coordinates of p_s in the camera frame are defined as

$$m_c = \frac{\overline{m}_c}{z_c} \tag{4–25}$$

where z_c is the 3D Euclidean coordinate of p_s in the camera frame, under the standard assumption that $z_c > \varepsilon$, where ε denotes an arbitrarily small positive scalar constant. After substituting (4–22) through (4–24) in (4–25) we can rewrite m_c as

$$m_c = z_c H_{cs}\frac{z_s^*}{z_s}H_s m_s^* \tag{4–26}$$

$$= z_c H_{cs}\frac{z_s^*}{z_s}H_s\left(\frac{1}{z_c^*}H_{cs}^{-1}\right)m_c^* \tag{4–27}$$

$$= \frac{z_c}{z_c^*}\frac{z_s^*}{z_s}H_{cs}H_s H_{cs}^{-1}m_c^* \tag{4–28}$$

$$= \frac{z_c}{z_c^*}\frac{z_s^*}{z_s}H_c m_c^* n \tag{4–29}$$

where H_c is the *camera-to-camera Euclidean homography* matrix, which maps points in the goal camera frame to points in the current camera frame.

4.4.2.4 Camera to Screen Geometry

The Euclidean coordinates m_s and m_c cannot be determined a priori. Rather, we can determine the pixel coordinates, denoted by p_s and p_c, of points on the screen and on the camera image, respectively. The pixel coordinates are related to m_s and m_c as follows:

$$p_s = A_s m_s \tag{4–30}$$

$$p_c = A_c m_c, \tag{4–31}$$

where A_c and A_s are intrinsic camera calibration matrices of the real and virtual camera, respectively. The real camera calibration matrix A_c can be determined through typical camera calibration techniques whereas the virtual camera calibration matrix, denoted by A_s, can be determined from the settings of the virtual reality simulator.

Equations (4–23) and (4–24) are altered as

$$p_s = G_s p_s^* \text{ and } p_c = G_{cs} p_s, \quad p_c^* = G_{cs} p_s^* \tag{4–32}$$

where G_s is the *virtual projective homography* matrix, and G_{cs} is the *camera-to-screen projective homography* matrix. These matrices can be expressed as

$$G_s = A_s H_s A_s^{-1} \text{ and } G_{cs} = A_c H_{cs} A_s^{-1}. \tag{4–33}$$

When solving for G_s or G_{cs} , we generally find normalized solutions

$$G_{sN} = \frac{G_s}{g_{s33}} \text{ and } G_{csN} = \frac{G_{cs}}{g_{cs33}} \tag{4–34}$$

where g_{s33} and g_{cs33} are the bottom right element of the G_s and G_{cs} matrices, respectively.

After combining (4–28), (4–30) and (4–31), we get

$$p_c = \frac{z_c}{z_c^*}\frac{z_s^*}{z_s} A_c H_{cs} H_s H_{cs}^{-1} A_c^{-1} p_c^*. \tag{4–35}$$

After substituting (4–33) and (4–34) in (4–35), we can rewrite p_c as

$$p_c = \frac{z_c}{z_c^*}\frac{z_s^*}{z_s}(g_{cs33} G_{csN} A_s) H_s (A_s^{-1} g_{cs33}^{-1} G_{csN}^{-1}) p_c^* \tag{4–36}$$

$$= \underbrace{\frac{z_c}{z_c^*}}_{\alpha_c} \underbrace{\frac{z_s^* g_{s33}}{z_s}}_{\alpha_s} \underbrace{G_{csN} G_{sN} G_{csN}^{-1}}_{G_{csN}} p_c^* \tag{4–37}$$

$$= \alpha_c \alpha_s G_{cN} p_c^* \tag{4–38}$$

where G_{cN} is the *normalized projective camera-to-camera homography matrix.*

Given G_{cN}, G_{csN}, and A_s, we can solve for H_s as

$$H_s = g_{s33}^{-1} A_s^{-1} G_{csN}^{-1} G_{sN} G_{csN} A_s. \tag{4–39}$$

We can solve for G_{csN} given the pixel coordinates of at least four corresponding points p_c in the camera image of points p_s on the screen image using a calibration technique. The calibration technique replaces typical camera calibration, and knowledge of A_c is no longer required. A routine was developed for solving G_{csN} using a calibration technique. An image of 30 squares is displayed on the screen (see Figure. (4–6)). The pixel locations of each corner point on the screen (p_s's) is extracted by hand. A camera image of the screen is captured (see Figure. (4–7)), and the pixel locations of all visible corner points (p_c's) are extracted by hand too. This gives up to 120 point correspondences. Estimation of the coordinates of the points p_c is subject to sensor noise and is a random process as well. To eliminate the effects of noise, G_{csN} is estimated as the solution to linear equations using RANSAC [11] in MATLAB (see Appendix C). RANSAC is an algorithm for robust

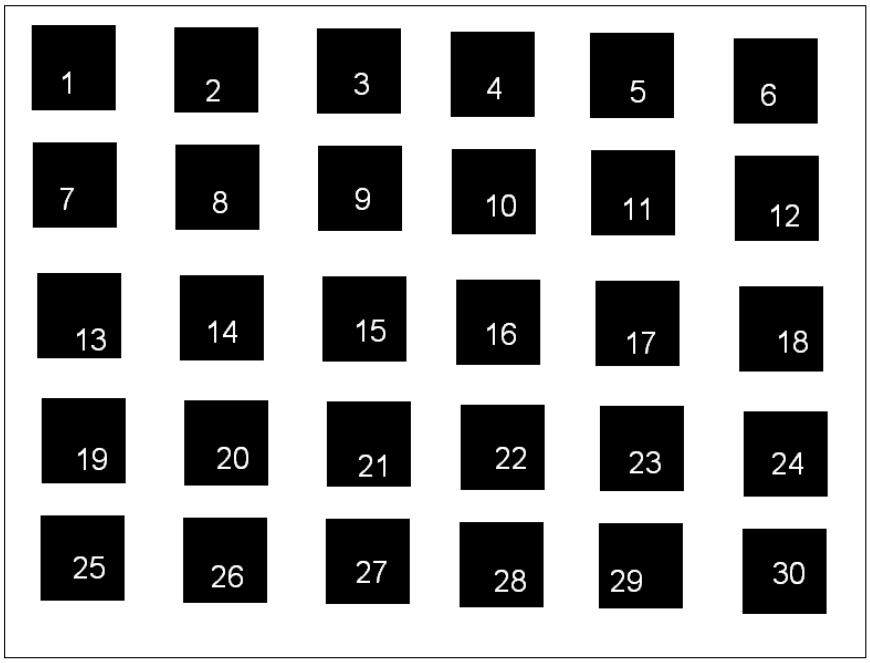

Figure 4–6: (a) Screen image of 1280×1024 resolution.

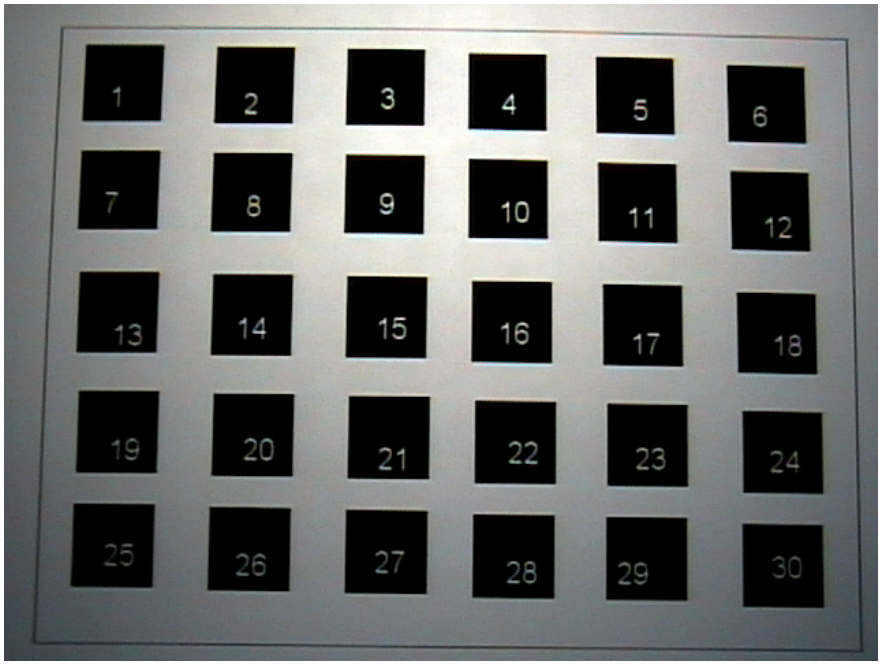

Figure 4–7: (b) Camera image of 640×480 resolution.

fitting of models in the presence of uncertainty and data outliers. It returns a "best guess" of G_{csN} that fits p_c and p_s, but eliminates any points that appear to be corrupted or inconsistent with the solution obtained (i.e., outliers). RANSAC itself is governed by a random process. To counter this, RANSAC is run many times (100 - 1000 times). Any points that are estimated to be outliers more often than a specified threshold (e.g. 20% of the time) are completely removed from consideration, and the process is repeated. This is repeated perhaps three or four times. On the final run, all solutions for G_{csN} are saved and the mean G_{csN} is kept as our estimate. The most recent trial for the setup at the HILS facility gave a mean and standard deviation for G_{csN} of

$$\mu(G_{csN}) = \begin{bmatrix} 0.6500 & -0.0296 & -83.2262 \\ 0.0072 & 0.6366 & -61.7221 \\ -0.0000 & -0.0001 & 1.0000 \end{bmatrix}, \ \sigma(G_{csN}) = \begin{bmatrix} 0.0016 & 0.0007 & 0.6298 \\ 0.0004 & 0.0013 & 0.3714 \\ 0.0000 & 0.0000 & 0.0000 \end{bmatrix}.$$

The constant calibration matrix of the virtual camera A_s can be determined from settings in the Vega Prime software responsible for rendering the visual database on the projectors. Vega Prime has a "field of view" setting, with a default value of 45°. A field of view of 45° means that the focal length of the camera is equal to half the size of the image surface. This is illustrated in Figure 4–8. The number of pixels in the vertical and horizontal directions is a setting of the image rendering program. In the case of the HILS facility, the screen's resolution is 1280×1024. Thus, assuming that the pixels have equal height and width, A_s is computed as

$$A_s = \begin{bmatrix} 640 & 0 & 640 \\ 0 & 640 & 512 \\ 0 & 0 & 1 \end{bmatrix}.$$

Thus, after determining $G_{cN}(t)$, G_{csN}, and A_s, we can solve for $H_s(t)$ using (4–39).

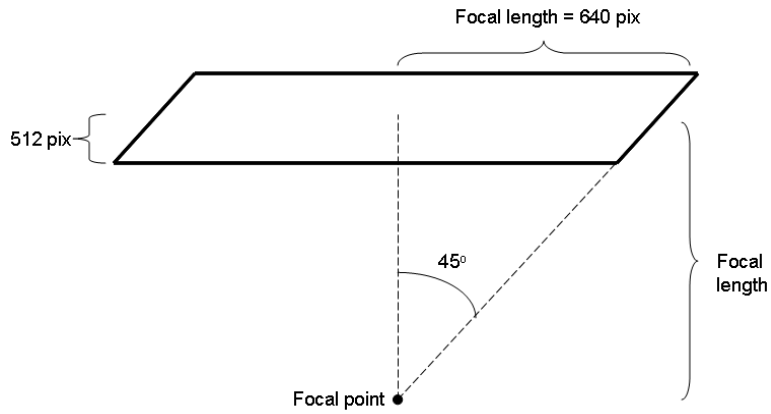

Figure 4–8: Illustration of virtual reality camera's instrinsic parameters.

It is interesting to note that $H_s(t)$ cannot be solved directly from (4–39) because of lack of g_{cs33}. Hence, we use G_{csN} to map p_c to screen points $\widehat{p_s} = G_{csN}p_c$ and $\widehat{p_s^*} = G_{csN}p_c^*$. The point $\widehat{p_s}$ will not generally be in proper homogenous coordinates due to the missing scale factor. We can correct this by renormalizing

$$\widehat{p_{sN}} = \frac{\widehat{p_s}}{\widehat{p_s}(3)}.$$

At this point, we can solve for G_{sN} using $\widehat{p_s}$ and $\widehat{p_s^*}$, and then recover normalized H_s using (4–40) as

$$H_{sN} = A_s^{-1} G_{sN} A_s. \tag{4–40}$$

4.5 Socket Communication

The rotation and translation information obtained from multi-view photogrammetry technique is required to maneuver the virtual scene on the projector screens. Hence, control commands generated at the image processing workstation are communicated to the main server workstation. This communication is set up using sockets. Sockets use TCP/IP protocol and specified port number for sending and receiving data between computers. The sockets program developed communicate data between C/C++ programs since the programs on both workstations are written in C/C++ (see Appendix C). They are based on the client-server architecture

where the image processing workstation acts as a client which is sending data to the main server. The data to be communicated is converted to a string and stored in a buffer of specified size before being sent. The reverse is done at the receiving computer where the control commands are utilized by the virtual database rendering program to control the motion of the virtual scene.

4.6 Conclusion

In this chapter, we discuss development of a testbed for real-time identification and tracking of features required for some visual servo control methods. The development included descriptions of efficient algorithms utilizing image processing and multi-view photogrammetry techniques. Feature point identification and tracking, and homography decomposition can be achieved at 20-25 Hz. In this chapter, we also discuss the solutions to problems faced during the development of the testbed.

CHAPTER 5
CONCLUSION

5.1 Summary of Results

We discuss determination of range and the Euclidean coordinates of an object feature undergoing general affine motion for a central paracatadioptric system via a nonlinear estimator. The nonlinear estimator was proven, via Lyapunov-based analysis, and numerically demonstrated to asymptotically determine the range and Euclidean motion coordinates for an object moving with known affine motion dynamics for a paracatadioptric system.

However, unlike image systems that are based on a planar image surface (or spherical or ellipsoidal surface), the dynamic vision system resulting from a general 3D surface's projected image is not guaranteed to maintain an affine form. In Chapter Three, the nonlinear state estimator developed for the paracatadioptirc system is shown to be applicable for the nonaffine projection resulting from the paraboloid image surface as described in [22]. The nonlinear estimator was proven and numerically demonstrated to asymptotically determine the range information and Euclidean coordinates from a single camera provided some observability conditions are satisfied and that the Euclidean motion parameters are known.

This thesis also describes my efforts in the development of a HILS visualization facility in Chapter Four. Chapter Four describes the development of a testbed for identification of features required for many visual servo control algorithms. The Kanade-Lucas-Tomasi algorithm is incorporated with multi-view photogrammetry techniques. Almost real-time (20-25 Hz) implementation of the "closed loop" was achieved, and this facility can now be utilized as a testbed for implementing developed vision based controllers.

5.2 Recommendations for Future Work

Our results prove that the range and Euclidean coordinates of an object feature undergoing general affine motion with known Euclidean motion parameters can be determined. The result would be strengthened if we can identify range and Euclidean coordinates of an object feature without knowledge of motion parameters a priori.

In Chapters Two and Three radial distortion effects of 3D imaging surfaces and camera lens could be taken into consideration in the model of the vision system. As stated in [3], two factors combine to cause blur in catadioptric systems: (1) the finite size of the lens aperture, and (2) the curvature of the mirror. Inaccuracies in computation caused by defocus blur can be accounted for by implementing methods for simultaneous computation of the defocus blur.

Multi-view photogrammetry techniques works under the assumption that the four feature points are coplanar points. Image segmentation and texture recognition techniques can be used to recognize different planes which would help in defining the region of interest of the feature detection algorithm to ensure feature points are coplanar. This would also make the tracker more consistent and reliable to intensity variations in the scene. Another issue to be addressed is to make sure that the points selected are not collinear. Also, the condition of the four feature points required to be coplanar and collinear can be eliminated by implementing the eight point algorithm proposed in [15], where the feature points don't have to satisfy the mentioned constraints.

Indexing of the feature points being tracked would enable recognizing feature points and assist in storing and retrieving them, even when feature points are no longer in view. This additional information may be required in other 3D motion determination techniques, e.g. structure from motion.

In multi-view photogrammetry techniques, decomposition of the homography matrix does not ensure a unique solution. Additional information is obtained by slightly moving the camera and comparing the current normals to normals of the previous image which helps in determining a unique solution of the normal. Corresponding unique rotation and translation can hence be obtained. However this method still does not guarantee the right solution. Hence, more reliable method can be implemented to obtain the additional information to determine the unique solution during homography decomposition. Also, quaternion representation can be used instead of angle-axis representation to get rid of any singularities.

Implementing socket communication between the image processing program in C/C++ and MATLAB/ Simulink would allow using accurate dynamics of any autonomous system modelled in MATLAB/ Simulink.

Further efficient methods can be used for image capturing from the firewire. A ring buffer mechanism along with genlock techniques can be used instead of using a callback function for each frame.

APPENDIX A
LYAPUNOV-BASED RANGE AND MOTION IDENTIFICATION OF A PARACATADIOPTRIC VISION SYTEM

To prove that $\dot{g}_1(\cdot), \dot{g}_2(\cdot), \dot{g}_3(\cdot) \in \mathcal{L}_\infty$, the time derivative of (3–9) is determined as follows:

$$\dot{g}_1 = \dot{y}_4 b_1 + y_4 \dot{b}_1 + \dot{y}_1 \Omega_0 + y_1 \dot{\Omega}_0 \tag{A–1}$$

$$\dot{g}_2 = \dot{y}_4 b_2 + y_4 \dot{b}_2 + \dot{y}_2 \Omega_0 + y_2 \dot{\Omega}_0 \tag{A–2}$$

$$\dot{g}_3 = \dot{y}_4 b_3 + y_4 \dot{b}_3 + \dot{y}_3 \Omega_0 + y_3 \dot{\Omega}_0 \tag{A–3}$$

where

$$\dot{y}_4 = \frac{a_{31} y_1 + a_{32} y_2 + a_{33} y_3}{L} + y_4 \Omega_0 \tag{A–4}$$

$$\begin{aligned} \dot{\Omega}_0 = \frac{\dot{b}_3}{L} - \frac{b_3 \dot{L}}{L^2} + \frac{(\dot{L} + \dot{x}_3)\,(y_1 \dot{x}_1 + y_2 \dot{x}_2 + y_3 \dot{x}_3)}{2f(L + x_3)^2} \\ - \frac{\dot{y}_1 \dot{x}_1 + y_1 \ddot{x}_1 + \dot{y}_2 \dot{x}_2 + y_2 \ddot{x}_2 + \dot{y}_3 \dot{x}_3 + y_3 \ddot{x}_3}{2f(L + x_3)} \end{aligned} \tag{A–5}$$

$$\dot{L} = \frac{x_1 \dot{x}_1 + x_2 \dot{x}_2 - L \dot{x}_3}{L + x_3} \tag{A–6}$$

$$\ddot{L} = \frac{\dot{x}_1^2 + x_1 \ddot{x}_1 + \dot{x}_2^2 + x_2 \ddot{x}_2 - \dot{L} \dot{x}_3 - L \ddot{x}_3}{L + x_3} - \frac{(x_1 \dot{x}_1 + x_2 \dot{x}_2 - L \dot{x}_3)(\dot{L} + \dot{x}_3)}{(L + x_3)^2} \tag{A–7}$$

$$\ddot{x}_1 = \dot{a}_{11} x_1 + a_{11} \dot{x}_1 + \dot{a}_{12} x_2 + a_{12} \dot{x}_2 + \dot{a}_{13} x_3 + a_{13} \dot{x}_3 + \dot{b}_1 \tag{A–8}$$

$$\ddot{x}_2 = \dot{a}_{21} x_1 + a_{21} \dot{x}_1 + \dot{a}_{22} x_2 + a_{22} \dot{x}_2 + \dot{a}_{23} x_3 + a_{23} \dot{x}_3 + \dot{b}_2 \tag{A–9}$$

$$\ddot{x}_3 = \dot{a}_{31} x_1 + a_{31} \dot{x}_1 + \dot{a}_{32} x_2 + a_{32} \dot{x}_2 + \dot{a}_{33} x_3 + a_{33} \dot{x}_3 + \dot{b}_3. \tag{A–10}$$

The facts that $\dot{y}(t) \in \mathcal{L}_\infty$ can be used along with Assumptions 2–1 through 2-4 to conclude from (A–1) through (A–10) that $\dot{g}_1(\cdot)$, $\dot{g}_2(\cdot)$, $\dot{g}_3(\cdot) \in \mathcal{L}_\infty$.

To prove that $\ddot{g}_1(\cdot)$, $\ddot{g}_2(\cdot)$, $\ddot{g}_3(\cdot) \in \mathcal{L}_\infty$, the time derivative of (A–1) through -(A–3) can be determined as follows:

$$\ddot{g}_1 = \ddot{y}_4 b_1 + 2\dot{y}_4 \dot{b}_1 + y_4 \ddot{b}_1 + \ddot{y}_1 \Omega_0 + 2\dot{y}_1 \dot{\Omega}_0 + y_1 \ddot{\Omega}_0 \tag{A–11}$$

$$\ddot{g}_2 = \ddot{y}_4 b_2 + 2\dot{y}_4 \dot{b}_2 + y_4 \ddot{b}_2 + \ddot{y}_2 \Omega_0 + 2\dot{y}_2 \dot{\Omega}_0 + y_2 \ddot{\Omega}_0 \tag{A–12}$$

$$\ddot{g}_3 = \ddot{y}_4 b_3 + 2\dot{y}_4 \dot{b}_3 + y_4 \ddot{b}_3 + \ddot{y}_3 \Omega_0 + 2\dot{y}_3 \dot{\Omega}_0 + y_3 \ddot{\Omega}_0. \tag{A–13}$$

In (A–11) through (A–13)

$$\begin{aligned} \ddot{y}_1 = {} & \dot{a}_{11} y_1 + a_{11} \dot{y}_1 + \dot{a}_{12} y_2 + a_{12} \dot{y}_2 + \dot{a}_{13} y_3 + a_{13} \dot{y}_3 \\ & + \frac{\dot{a}_{31} y_1^2 + 2a_{31} y_1 \dot{y}_1 + \dot{a}_{32} y_1 y_2 + \dot{a}_{33} y_1 y_3}{2f} \\ & + \frac{a_{32}(\dot{y}_1 y_2 + y_1 \dot{y}_2) + a_{33}(\dot{y}_1 y_3 + y_1 \dot{y}_3)}{2f} + \dot{g}_1 \end{aligned} \tag{A–14}$$

$$\begin{aligned} \ddot{y}_2 = {} & \dot{a}_{21} y_1 + a_{21} \dot{y}_1 + \dot{a}_{22} y_2 + a_{22} \dot{y}_2 + \dot{a}_{23} y_3 + a_{23} \dot{y}_3 \\ & + \frac{\dot{a}_{31} y_1 y_2 + a_{31}(\dot{y}_1 y_2 + y_1 \dot{y}_2) + \dot{a}_{32} y_2^2}{2f} \\ & + \frac{2a_{32} y_2 \dot{y}_2 + \dot{a}_{33} y_2 y_3 + a_{33}(\dot{y}_2 y_3 + y_2 \dot{y}_3)}{2f} + \dot{g}_2 \end{aligned} \tag{A–15}$$

$$\begin{aligned} \ddot{y}_3 = {} & \dot{a}_{31} y_1 + a_{31} \dot{y}_1 + \dot{a}_{32} y_2 + a_{32} \dot{y}_2 + \dot{a}_{33} y_3 \\ & + a_{33} \dot{y}_3 + \frac{\dot{a}_{31} y_1 y_3 + a_{31}(\dot{y}_1 y_3 + y_1 \dot{y}_3) + \dot{a}_{32} y_2 y_3}{2f} \\ & + \frac{a_{32}(\dot{y}_2 y_3 + y_2 \dot{y}_3) + \dot{a}_{33} y_3^2 + 2a_{33} y_3 \dot{y}_3}{2f} + \dot{g}_3 \end{aligned} \tag{A–16}$$

$$\begin{aligned} \ddot{y}_4 = {} & \frac{\dot{a}_{31} y_1 + a_{31} \dot{y}_1 + \dot{a}_{32} y_2 + a_{32} \dot{y}_2 + \dot{a}_{33} y_3 + a_{33} \dot{y}_3}{L} \\ & - \frac{(a_{31} y_1 + a_{32} y_2 + a_{33} y_3)\, \dot{L}}{L^2} + \dot{y}_4 \Omega_0 + y_4 \dot{\Omega}_0 \end{aligned} \tag{A–17}$$

$$\begin{aligned}\ddot{\Omega}_0 = & \frac{\ddot{b}_3}{L} - \frac{\dot{b}_3\dot{L}}{L^2} - \frac{\left(\dot{b}_3\dot{L} + b_3\ddot{L}\right)}{L^2} + \frac{2b_3\dot{L}^2}{L^3} \\ & - \frac{\ddot{y}_1\dot{x}_1 + 2\dot{y}_1\ddot{x}_1 + y_1\dddot{x}_1 + \ddot{y}_2\dot{x}_2 + 2\dot{y}_2\ddot{x}_2}{2f(L+x_3)} \\ & - \frac{y_2\dddot{x}_2 + \ddot{y}_3\dot{x}_3 + 2\dot{y}_3\ddot{x}_3 + y_3\dddot{x}_3}{2f(L+x_3)} + \frac{\dot{y}_1\dot{x}_1\left(\dot{L} + \dot{x}_3\right)}{2f(L+x_3)^2} \\ & + \frac{(y_1\ddot{x}_1 + \dot{y}_2\dot{x}_2 + y_2\ddot{x}_2 + \dot{y}_3\dot{x}_3 + y_3\ddot{x}_3)\left(\dot{L} + \dot{x}_3\right)}{2f(L+x_3)^2} \\ & + \frac{(\dot{L} + \dot{x}_3)\left(\dot{y}_1\dot{x}_1 + y_1\ddot{x}_1 + \dot{y}_2\dot{x}_2 + y_2\ddot{x}_2 + \dot{y}_3\dot{x}_3\right)}{2f(L+x_3)^2} \\ & + \frac{(\dot{L} + \dot{x}_3)y_3\ddot{x}_3}{2f(L+x_3)^2} + \frac{(\ddot{L} + \ddot{x}_3)\left(y_1\dot{x}_1 + y_2\dot{x}_2 + y_3\dot{x}_3\right)}{2f(L+x_3)^2} \\ & - \frac{(\dot{L} + \dot{x}_3)^2\left(y_1\dot{x}_1 + y_2\dot{x}_2 + y_3\dot{x}_3\right)}{f(L+x_3)^3}\end{aligned} \tag{A–18}$$

$$\begin{aligned}\dddot{x}_1 = & \ddot{a}_{11}x_1 + 2\dot{a}_{11}\dot{x}_1 + a_{11}\ddot{x}_1 + \ddot{a}_{12}x_2 + 2\dot{a}_{12}\dot{x}_2 \\ & + a_{12}\ddot{x}_2 + \ddot{a}_{13}x_3 + 2\dot{a}_{13}\dot{x}_3 + a_{13}\ddot{x}_3 + \ddot{b}_1\end{aligned} \tag{A–19}$$

$$\begin{aligned}\dddot{x}_2 = & \ddot{a}_{21}x_1 + 2\dot{a}_{21}\dot{x}_1 + a_{21}\ddot{x}_1 + \ddot{a}_{22}x_2 + 2\dot{a}_{22}\dot{x}_2 \\ & + a_{22}\ddot{x}_2 + \ddot{a}_{23}x_3 + 2\dot{a}_{23}\dot{x}_3 + a_{23}\ddot{x}_3 + \ddot{b}_2\end{aligned} \tag{A–20}$$

$$\begin{aligned}\dddot{x}_3 = & \ddot{a}_{31}x_1 + 2\dot{a}_{31}\dot{x}_1 + a_{31}\ddot{x}_1 + \ddot{a}_{32}x_2 + 2\dot{a}_{32}\dot{x}_2 \\ & + a_{32}\ddot{x}_2 + \ddot{a}_{33}x_3 + 2\dot{a}_{33}\dot{x}_3 + a_{33}\ddot{x}_3 + \ddot{b}_3.\end{aligned} \tag{A–21}$$

The expressions given in (A–1) through (A–21), Assumptions 2-1 through 2-5, and the facts that, $\dot{y}(t)$, $\dot{g}(\cdot) \in \mathcal{L}_\infty$ can now be used to prove that $\ddot{g}_1(\cdot)$, $\ddot{g}_2(\cdot)$, $\ddot{g}_3(\cdot) \in \mathcal{L}_\infty$.

APPENDIX B
LYAPUNOV-BASED RANGE AND MOTION IDENTIFICATION OF A NONAFFINE PERSPECTIVE DYNAMIC SYSTEM

To prove that $\dot{f}_1(\cdot), \dot{f}_2(\cdot),\ \dot{f}_3(\cdot) \in \mathcal{L}_\infty$, the time derivative of equation (3–9) is determined as follows

$$\begin{aligned}\dot{f}_1 = {} & \dot{b}_3 y_5 + b_3 \dot{y}_5 - 2\dot{b}_1 y_1 y_5 - 2b_1 y_1 \dot{y}_5 - 2b_1 \dot{y}_1 y_5 \\ & - 2\dot{b}_2 y_1 y_6 - 2b_2 \dot{y}_1 y_6 - 2b_2 y_1 \dot{y}_6 + \dot{b}_1 y_7 + b_1 \dot{y}_7\end{aligned} \tag{B–1}$$

$$\begin{aligned}\dot{f}_2 = {} & -2\dot{b}_1 y_2 y_5 - 2b_1 \dot{y}_2 y_5 - 2b_1 y_2 \dot{y}_5 \ + \dot{b}_3 y_6 + b_3 \dot{y}_6 \\ & - 2\dot{b}_2 y_2 y_6 - 2b_2 \dot{y}_2 y_6 - 2b_2 y_2 \dot{y}_6 + \dot{b}_2 y_7 + b_2 \dot{y}_7\end{aligned} \tag{B–2}$$

$$\begin{aligned}\dot{f}_3 = {} & -2\dot{b}_1 y_3 y_5 - 2b_1 \dot{y}_3 y_5 - 2b_1 y_3 \dot{y}_5 - 2\dot{b}_2 y_3 y_6 \\ & - 2b_2 \dot{y}_3 y_6 - 2b_2 y_3 \dot{y}_6 + 2\dot{b}_3 y_7 + 2b_3 \dot{y}_7.\end{aligned} \tag{B–3}$$

The facts that $\dot{y}(t) \in \mathcal{L}_\infty$ can be used along with Assumptions 3–1 through 3-4 to conclude from (B–1) through (B–3) that $\dot{f}_1(\cdot)$ to $\dot{f}_3(\cdot) \in \mathcal{L}_\infty$.

To prove that $\ddot{f}_1(\cdot)$, $\ddot{f}_2(\cdot)$ and $\ddot{f}_3(\cdot) \in \mathcal{L}_\infty$, the time derivative of (B–1) through (B–3) can be determined as follows:

$$\begin{aligned}\ddot{f}_1 = {} & \ddot{b}_3 y_5 + 2\dot{b}_3 \dot{y}_5 + b_3 \ddot{y}_5 - 2\ddot{b}_1 y_1 y_5 - 4\dot{b}_1 y_1 \dot{y}_5 - 4\dot{b}_1 \dot{y}_1 y_5 - 4b_1 \dot{y}_1 \dot{y}_5 \\ & - 2b_1 y_1 \ddot{y}_5 - 2b_1 \ddot{y}_1 y_5 - 2\ddot{b}_2 y_1 y_6 - 4\dot{b}_2 \dot{y}_1 y_6 - 4\dot{b}_2 y_1 \dot{y}_6 - 2b_2 \ddot{y}_1 y_6 \\ & - 4b_2 \dot{y}_1 \dot{y}_6 - 2b_2 y_1 \ddot{y}_6 + \ddot{b}_1 y_7 + 2\dot{b}_1 \dot{y}_7 + b_1 \ddot{y}_7\end{aligned} \tag{B–4}$$

$$\ddot{f}_2 = -2\ddot{b}_1 y_2 y_5 - 4\dot{b}_1 \dot{y}_2 y_5 - 4\dot{b}_1 y_2 \dot{y}_5 - 2b_1\ \ddot{y}_2 y_5 - 4b_1 \dot{y}_2 \dot{y}_5 - 2b_1 y_2\ \ddot{y}_5 \tag{B–5}$$
$$+ \ddot{b}_3 y_6 + 2\dot{b}_3 y_6 + b_3 \ddot{y}_6 - 2\ddot{b}_2 y_2 y_6 - 4\dot{b}_2 \dot{y}_2 y_6 - 4\dot{b}_2 y_2 \dot{y}_6 - 2b_2 \ddot{y}_2 y_6$$
$$- 4b_2\ \dot{y}_2 \dot{y}_6 - 2b_2 y_2 \ddot{y}_6 + \ddot{b}_2 y_7 + 2\dot{b}_2 \dot{y}_7 + b_2 \ddot{y}_7$$

$$\ddot{f}_3 = -2\ddot{b}_1 y_3 y_5 - 4\dot{b}_1 \dot{y}_3 y_5 - 4\dot{b}_1 y_3 \dot{y}_5 - 2b_1 \ddot{y}_3 y_5 - 4b_1 \dot{y}_3 \dot{y}_5 - 2b_1 y_3 \ddot{y}_5 \tag{B–6}$$
$$- 2\ddot{b}_2 y_3 y_6 - 4\dot{b}_2 \dot{y}_3 y_6 - 4\dot{b}_2 y_3 \dot{y}_6 - 2b_2 \ddot{y}_3 y_6 - 4b_2 \dot{y}_3 \dot{y}_6 - 2b_2 y_3 \ddot{y}_6$$
$$+ 2\ddot{b}_3 y_7 + 4\dot{b}_3 \dot{y}_7 + 2b_3 \ddot{y}_7.$$

In (B–4) through (B–6)

$$\dot{y}_5 = \dot{y}_1 \sqrt{\frac{y_4}{y_3}} + \frac{y_1 \dot{y}_4}{2y_3} \sqrt{\frac{y_3}{y_4} - \frac{y_1 y_4 \dot{y}_3}{2y_3^2} \sqrt{\frac{y_3}{y_4}}} \tag{B–7}$$

$$\dot{y}_6 = \dot{y}_2 \sqrt{\frac{y_4}{y_3}} + \frac{y_2 \dot{y}_4}{2y_3} \sqrt{\frac{y_3}{y_4}} - \frac{y_2 y_4 \dot{y}_3}{2y_3^2} \sqrt{\frac{y_3}{y_4}} \tag{B–8}$$

$$\dot{y}_7 = \frac{(\dot{y}_3 y_4 + y_3 \dot{y}_4)}{2\sqrt{y_3 y_4}} \tag{B–9}$$

$$\ddot{y}_1 = \dot{a}_{11} y_1 + a_{11} \dot{y}_1 + \dot{a}_{12} y_2 + a_{12} \dot{y}_2 + \dot{a}_{13} y_3 + a_{13} \dot{y}_3 \tag{B–10}$$
$$+ \frac{y_3(\dot{a}_{31} y_1^2 + 2a_{31} y_1 \dot{y}_1) - a_{31} y_1^2 \dot{y}_3}{y_3^2} + \frac{y_3(\dot{a}_{32} y_1 y_2 + a_{32}(\dot{y}_1 y_2 + y_1 \dot{y}_2))}{y_3^2}$$
$$- \frac{a_{32} y_1 y_2 \dot{y}_3}{y_3^2} + \dot{a}_{33} y_1 + a_{33} \dot{y}_1 - \frac{2\left(y_3(\dot{a}_{11} y_1^3 + 3a_{11} y_1^2 \dot{y}_1) - a_{11} y_1^3 \dot{y}_3\right)}{y_3^2}$$
$$- \frac{2y_3(\dot{a}_{12} y_1^2 y_2 + a_{12}(2y_1 \dot{y}_1 y_2 + y_1^2 \dot{y}_2))}{y_3^2} + \frac{2a_{12} y_1^2 y_2 \dot{y}_3)}{y_3} - 2\left[\frac{y_3(\dot{a}_{21} y_1^2 y_2}{y_3^2}\right.$$
$$\left. + \frac{a_{21}(2y_1 \dot{y}_1 y_2 + y_1^2 \dot{y}_2) - a_{21} y_1^2 y_2 \dot{y}_3}{y_3^2}\right] - \frac{2(\dot{a}_{22} y_1 y_2^2 + a_{22}(\dot{y}_1 y_2^2 + 2y_1 y_2 \dot{y}_2))}{y_3}$$
$$+ \frac{2a_{22} y_1 y_2^2 \dot{y}_3}{y_3^2} - 2(\dot{a}_{13} y_1^2 + 2a_{13} y_1 \dot{y}_1) - 2(\dot{a}_{23} y_1 y_2 + a_{23}(\dot{y}_1 y_2 + y_1 \dot{y}_2)) + \dot{f}_1$$

$$\ddot{y}_2 = \dot{a}_{21}y_1 + a_{21}\dot{y}_1 + \dot{a}_{22}y_2 + a_{22}\dot{y}_2 + \dot{a}_{23}y_3 + a_{23}\dot{y}_3 \tag{B–11}$$
$$+ \frac{y_3(\dot{a}_{32}y_2^2 + 2a_{32}y_2\dot{y}_2) - a_{32}y_2^2\dot{y}_3}{y_3^2} + \frac{\dot{a}_{31}y_1y_2 + a_{31}(\dot{y}_1y_2 + y_1\dot{y}_2)}{y_3}$$
$$- \frac{a_{31}y_1y_2\dot{y}_3}{y_3^2} + \dot{a}_{33}y_2 + a_{33}\dot{y}_2 - 2\left[\frac{y_3(\dot{a}_{22}y_2^3 + 3a_{22}y_2^2\dot{y}_2) - a_{22}y_2^3\dot{y}_3}{y_3^2}\right]$$
$$- 2\frac{\dot{a}_{12}y_1y_2^2 + a_{12}(2y_1\dot{y}_2y_2 + y_2^2\dot{y}_1)}{y_3} + \frac{2a_{12}y_2^2y_1\dot{y}_3}{y_3^2} - 2\frac{\dot{a}_{21}y_1y_2^2}{y_3}$$
$$+ \frac{y_3a_{21}(2y_1\dot{y}_2y_2 + y_2^2\dot{y}_1) - a_{21}y_2^2y_1\dot{y}_3}{y_3^2} - 2\left[\frac{(\dot{a}_{11}y_2y_1^2 + a_{11}(\dot{y}_2y_1^2 + 2y_1y_2\dot{y}_1)}{y_3}\right.$$
$$\left.-\frac{a_{11}y_2y_1^2\dot{y}_3}{y_3^2}\right] - 2(\dot{a}_{23}y_2^2 + 2a_{23}y_2\dot{y}_2) - 2(\dot{a}_{13}y_1y_2 + a_{13}(\dot{y}_1y_2 + y_1\dot{y}_2)) + \dot{f}_2$$

$$\ddot{y}_3 = 2(\dot{a}_{31}y_1 + a_{31}\dot{y}_1) + 2(\dot{a}_{32}y_2 + a_{32}\dot{y}_2) + 2(\dot{a}_{33}y_3 + a_{33}\dot{y}_3) \tag{B–12}$$
$$- 2(\dot{a}_{11}y_1^2 + 2a_{11}y_1\dot{y}_1) - 2(\dot{a}_{12}y_1y_2 + a_{12}(\dot{y}_1y_2 + y_1\dot{y}_2))$$
$$- 2(\dot{a}_{21}y_1y_2 + a_{21}(\dot{y}_1y_2 + y_1\dot{y}_2)) - 2(\dot{a}_{22}y_2^2 + 2a_{22}y_2\dot{y}_2)$$
$$- 2(\dot{a}_{13}y_1y_3 + a_{13}(\dot{y}_1y_3 + y_1\dot{y}_3) - 2(\dot{a}_{23}y_2y_3 + a_{23}(\dot{y}_2y_3 + y_2\dot{y}_3)) + \dot{f}_3$$

$$\ddot{y}_4 = -2\left[\frac{\dot{a}_{11}y_1^2y_4 + a_{11}(2y_1\dot{y}_1y_4 + y_1^2\dot{y}_4)}{y_3} - \frac{a_{11}y_1^2y_4\dot{y}_3}{y_3^2}\right] \tag{B–13}$$
$$- 2\left[\frac{\dot{a}_{22}y_2^2y_4}{y_3} + \frac{a_{22}y_3(2y_2\dot{y}_2y_4 + y_2^2\dot{y}_4) - a_{22}y_2^2y_4\dot{y}_3}{y_3^2}\right]$$
$$- 2\left[\frac{\dot{a}_{12}y_1y_2y_4 + a_{12}\dot{y}_1y_2y_4}{y_3} + \frac{a_{12}y_1y_3(\dot{y}_2y_4 + y_2\dot{y}_4) - a_{12}y_1y_2y_4\dot{y}_3}{y_3^2}\right]$$
$$- 2\left[\frac{\dot{a}_{21}y_1y_2y_4 + a_{21}\dot{y}_1y_2y_4}{y_3} + \frac{a_{21}y_1y_3(\dot{y}_2y_4 + y_2\dot{y}_4) - a_{21}y_1y_2y_4\dot{y}_3}{y_3^2}\right]$$
$$- 2(\dot{a}_{13}y_1y_4 + a_{13}(\dot{y}_1y_4 + y_1\dot{y}_4)) - 2(\dot{a}_{23}y_2y_4 + a_{23}(\dot{y}_2y_4 + y_2\dot{y}_4))$$
$$- 2(\dot{b}_1y_4y_5 + b_1(\dot{y}_4y_5 + y_4\dot{y}_5)) - 2(\dot{b}_2y_4y_6 + b_2(\dot{y}_4y_6 + y_4\dot{y}_6))$$

$$\ddot{y}_5 = \ddot{y}_1\sqrt{\frac{y_4}{y_3}} + \frac{\dot{y}_1}{2y_3^2}\sqrt{\frac{y_3}{y_4}}(y_3\dot{y}_4 - y_4\dot{y}_3) \tag{B–14}$$

$$+\frac{1}{2}\left\{\sqrt{\frac{y_3}{y_4}}\frac{\dot{y}_1\dot{y}_4}{y_3} + y_1 - \sqrt[3/2]{\frac{y_3}{y_4}}(\dot{y}_4y_3 - \dot{y}_3y_4)\frac{\dot{y}_4}{2y_3^3} + \sqrt{\frac{y_3}{y_4}}\frac{(y_3\ddot{y}_4 - \dot{y}_4\dot{y}_3)}{y_3^2}\right\}$$

$$+\frac{1}{2} - \left\{\sqrt{\frac{y_3}{y_4}}\frac{\dot{y}_1\dot{y}_3y_4}{y_3^2} + y_1\left[\sqrt[3/2]{\frac{y_3}{y_4}}(y_3\dot{y}_4 - y_4\dot{y}_3)\frac{y_4\dot{y}_3}{2y_3^4}\right.\right.$$

$$\left.\left.+\sqrt{\frac{y_3}{y_4}}\left(\frac{-\dot{y}_4\dot{y}_3 - y_4\ddot{y}_3}{y_3^2} + \frac{2y_4y_3\dot{y}_3^2}{y_3^4}\right)\right]\right\} \tag{B–15}$$

$$\ddot{y}_6 = \ddot{y}_2\sqrt{\frac{y_4}{y_3}} + \frac{\dot{y}_2}{2y_3^2}\sqrt{\frac{y_3}{y_4}}(y_3\dot{y}_4 - y_4\dot{y}_3) \tag{B–16}$$

$$+\frac{1}{2}\left\{\sqrt{\frac{y_3}{y_4}}\frac{\dot{y}_2\dot{y}_4}{y_3} + y_2\left[-\sqrt[3/2]{\frac{y_3}{y_4}}(\dot{y}_4y_3 - \dot{y}_3y_4)\frac{\dot{y}_4}{2y_3^3}\right.\right.$$

$$\left.\left.+\sqrt{\frac{y_3}{y_4}}\frac{(y_3\ddot{y}_4 - \dot{y}_4\dot{y}_3)}{y_3^2}\right]\right\}$$

$$+\frac{1}{2}\left\{-\sqrt{\frac{y_3}{y_4}}\frac{\dot{y}_2\dot{y}_3y_4}{y_3^2} + y_2\left[\sqrt[3/2]{\frac{y_3}{y_4}}(y_3\dot{y}_4 - y_4\dot{y}_3)\frac{y_4\dot{y}_3}{2y_3^4}\right.\right.$$

$$\left.\left.+\sqrt{\frac{y_3}{y_4}}\left(\frac{-\dot{y}_4\dot{y}_3 - y_4\ddot{y}_3}{y_3^2} + \frac{2y_4y_3\dot{y}_3^2}{y_3^4}\right)\right]\right\}$$

$$\ddot{y}_7 = \frac{1}{2}\left\{\left[-(y_3\dot{y}_4 + y_4\dot{y}_3)\frac{\dot{y}_3y_4}{2\sqrt[3/2]{y_3y_4}} + \frac{(\dot{y}_3\dot{y}_4 + y_4\ddot{y}_3)}{\sqrt{y_3y_4}}\right]\right. \tag{B–17}$$

$$\left.+\left[\frac{-(y_3\dot{y}_4)^2}{2\sqrt[3/2]{y_3y_4}} - \frac{y_4\dot{y}_3y_3\dot{y}_4}{2\sqrt[3/2]{y_3y_4}} + \frac{(\dot{y}_3\dot{y}_4 + \ddot{y}_4y_3)}{\sqrt{y_3y_4}}\right]\right\}.$$

The expressions given in (B–4) through (B–17), Assumptions 3-1 through 3-4, and the facts that, $\dot{y}(t)$, $f(y_5, y_6, y_7) \in \mathcal{L}_\infty$ can now be used to prove that $\ddot{f}_1(\cdot)$, $\ddot{f}_2(\cdot)$, $\ddot{f}_3(\cdot) \in \mathcal{L}_\infty$.

APPENDIX C
Hardware In The Loop

C.1 Program on the Server Machine

```
// This program render the virtual database on the projector screen
// using Vega Prime 2.0. The program receives control information
// through sockets to manuever the virtual scene.
#include <vsgu.h>
#include <vp.h>
#include <vpSearchPath.h>
#include <vpApp.h>
#include "vuAllocTracer.h"
#include "vuDistributed.h"
#include <vpMotion.h>
#include <vpMotionUFO.h>
#include <vpMotionGame.h>
#include <vuMatrix.h>
#include <vpObject.h>
#include "vuSocketTCP.h"
#include "vuImageLoader.h"
#include "vuImageFactory.h"
#include "vsgi_bmp.h"
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <windows.h>
```

```
#include <winsock.h>
#include <stdio.h>
#define N 18
#define M 4
#define PI180 .01745329251
#define VSB_PORT 9898
#define NETWORK_ERROR -1
#define NETWORK_OK 0
vuAllocTracer tracer(true, true);
void ReportError(int, const char *);
// ################################
// # To run this distributed rendering sample, use the #
// # Distributed Rendering Utilites to setup a master and #
// # a slave system #
// ################################
class myApp : public vpApp, public vpKernel::Subscriber
{
    public:
    myApp() : //m_pMotionUFO(NULL),
    m_pMotionGame(NULL),
    m_pObserver(NULL)
    {};
    ~myApp()
    {
    //if (m_pMotionUFO) { m_pMotionUFO->unref(); }
    if (m_pMotionGame) { m_pMotionGame->unref();}
    if (m_pObserver) { m_pObserver->unref(); }
```

```
if (m_pPlane) { m_pPlane->unref(); }
};
virtual void run();
virtual void setSockets();
double C[N][1];
double D[N][1];
double x[N][1];
double x0[N][1];
double U,V,W,vn,ve,vh,No,Ea,He,k;
int configure()
{
// pre-configuration
// configure vega prime system first
vpApp::configure();
m_pMotionGame = new vpMotionGame ;
assert(m_pMotionGame != NULL);
m_pMotionGame->ref();
// Make sure the observer exists
m_pObserver = vpObserver::find("myObserver");
assert(m_pObserver != NULL);
m_pObserver->ref();
m_pPlane = vpObject::find("plane");
assert(m_pPlane != NULL);
m_pPlane->ref();
start_recv=0;
vx=0;vy=0;vz=0;wx=0;wy=0;wz=0;
Tx=0;Ty=0;Tz=0;Tx=0;Ty=0;Tz=0;
```

```
m_pObserver->setPosition(2200,2809,150);
m_pObserver->setOrientation(0,0,0);
// Turn off the slaves motion strategy so it can be positioned
if( vuDistributed::getMode() == vuDistributed::MODE_SLAVE )
{
m_pObserver->setStrategy(NULL);
k=1;
}
return vsgu::SUCCESS;
}
virtual void onKeyInput(vrWindow::Key key, int mod)
{
k=0;
switch (key)
{
case vrWindow::KEY_z:
    wy = .1;
    break;
case vrWindow::KEY_c:
    wy= -.1;
    break;
case vrWindow::KEY_x:
    wy= 0;
    break;
case vrWindow::KEY_s:
case vrWindow::KEY_S:
    setSockets();
```

```
            start_recv=1;
            break;
        case vrWindow::KEY_w:
        case vrWindow::KEY_W:
            m_pObserver->getPosition(&Tx,&Ty,&Tz);
            m_pObserver->getOrientation(&Rz, &Rx, &Ry);
            break;
        case vrWindow::KEY_v:
        case vrWindow::KEY_V:
            m_pObserver->setPosition(6897.260 ,1319.363 ,646.864);
            m_pObserver->setOrientation(239.727 ,272.676 ,0.000);
            break;
        default:
            vpApp::onKeyInput(key, mod);
            break;
            k=1;
        }
    }
        protected:
*
* Handler of the non-latency critical event. Print out selected
* input sources. This is only an example that based on Saitek
* Cyborg 3D Rumble Force joystick. Please refer to your joystick's
* manual for the available input sources.
*/
        void notify(vpKernel::Event event, const vpKernel *service)
        {
```

```
 #ifdef WIN32
system("cls");
 #endif
}
private:
//vpMotionUFO* m_pMotionUFO;
vpMotionGame* m_pMotionGame;
 vpObserver* m_pObserver;
vpObject* m_pPlane;
vuMatrix< double> Twc1;
vuMatrix< double> Rwc1;
vuMatrix< double> Tc1c2;
vuMatrix< double> Rc1c2;
int amount, nret, start_recv;
double Tx,Ty,Tz,Rx,Ry,Rz, vx,vy,vz,wx,wy,wz;
SOCKET theClient;
SOCKET listeningSocket;
};
void ReportError(int errorCode, const char *whichFunc)
{
char errorMsg[92]; // Declare a buffer to hold
// the generated error message
ZeroMemory(errorMsg, 92); // Automatically NULL-terminate the string
 // The following line copies the phrase, whichFunc string, and integer
errorCode into the buffer
sprintf(errorMsg, "Call to %s returned error %d!", (char *)whichFunc,
errorCode);
```

```
        MessageBox(NULL, errorMsg, "socketIndication", MB_OK);
    }
    void myApp::setSockets()
    {
        if( vuDistributed::getMode() == vuDistributed::MODE_MASTER )
        {
        WORD sockVersion;
        WSADATA wsaData;
        sockVersion = MAKEWORD(1, 1); // We'd like Winsock version 1.1
        // We begin by initializing Winsock
        WSAStartup(sockVersion, &wsaData);
        listeningSocket = socket(AF_INET, // Go over TCP/IP
            SOCK_STREAM, // This is a stream-oriented socket
            IPPROTO_TCP); // Use TCP rather than UDP
        if (listeningSocket == INVALID_SOCKET)
        {
            nret = WSAGetLastError(); // Get a more detailed error
            ReportError(nret, "socket()"); // Report the error with our custom
function
            WSACleanup(); // Shutdown Winsock
            printf("error creating listen socket\n");
            return ;
        } // Return an error value
        // Use a SOCKADDR_IN struct to fill in address information
        SOCKADDR_IN serverInfo;
        serverInfo.sin_family = AF_INET;
        serverInfo.sin_addr.s_addr = INADDR_ANY; // Since this socket
```

```
 // is listening for connections, any local address will do
serverInfo.sin_port = htons(6262); // Convert integer 8888 to
 //network-byte order and insert into the port field
// Bind the socket to our local server address
nret = bind(listeningSocket, (LPSOCKADDR)&serverInfo, sizeof(struct
sockaddr));
if (nret == SOCKET_ERROR)
{
    nret = WSAGetLastError();
    ReportError(nret, "bind()");
    WSACleanup();
    printf(" NETWORK_ERROR 1");
    return ;
}
// Make the socket listen
nret = listen(listeningSocket, 10); // Up to 10 connections may
 //wait at any one time to be accept()'ed
if (nret == SOCKET_ERROR)
{
    nret = WSAGetLastError();
    printf("%d \n",nret);
    Sleep(2000);
    WSACleanup();
    return ;
}
theClient = accept(listeningSocket,
    NULL, // Address of a sockaddr structure (see explanation below)
```

```
            NULL); // Address of a variable containing size of sockaddr struct
        if (theClient == INVALID_SOCKET)
        {
            nret = WSAGetLastError();
            ReportError(nret, "accept()");
            printf("error accepting socket\n");
            WSACleanup();
            return ;
        }
        printf("3 - Client Connected\n");//output to see how far it gets, delete
when happy
        }
    }
    void myApp::run()
    {
        uint frameNum;
        m_pMotionGame->setSpeed(0.1);
        m_pMotionGame->setRateLook(5.0);
        amount = 1;
        struct DRTest
            {
                double x1, y1, z1, h1, p1, r1;
            };
        struct DRTest drBuffer;
        m_pObserver->setStrategyEnable(true);
        m_pObserver->setStrategy(m_pMotionGame);
        char buffer[24];// On the stack
```

```
((int*)buffer)[0] =0; ((int*)buffer)[1]=0; ((int*)buffer)[2]=0;
((int*)buffer)[3] =0;((int*)buffer)[4]=0;((int*)buffer)[5]=0;
int got_dat=1, quit = 0;
int frame = 0, tmp=0;
// rendering loop
while ( (frameNum = vpKernel::instance()->beginFrame()) > 0 && quit
== 0)
{
    //printf("-");
    got_dat=0;
    int myframenum = frameNum;
    tmp=0;
    if( vuDistributed::getMode() == vuDistributed::MODE_MASTER )
    {
    if(start_recv==1 )
    {
        nret = recv(theClient,
            buffer,
            24, // Complete size of buffer
            MSG_PEEK);
        if(nret != 0)
            start_recv=2;
    }
    if(start_recv==2)
    {
//              printf("recv\n");
        nret = recv(theClient,
```

```
            buffer,
            24, // Complete size of buffer
            0);
        got_dat=1;
    }
    if (nret == SOCKET_ERROR)
    {
        // Get a specific code
        // Handle accordingly
        printf(" NETWORK_ERROR 2: ");
        nret = WSAGetLastError();
        printf("%d \n",nret);
        Sleep(2000);
        quit=1;
    }
    else if (nret != 0) // if(nret==24)//if(got_dat)
    {
    m_pObserver->getPosition(&Tx,&Ty,&Tz);
    m_pObserver->getOrientation(&Rz, &Rx, &Ry);
    //note that observer frame is not the same as vision system camera
frame!
    //need to change order
    //vision system:
    //+Tx is camera right, +Ty is camera down, +Tz is camera forward
    //+wx is pitch //+wy is yaw +wz is roll
    //observer:
    //+x is camera right, +y is camera forward, +z is camera up
```

```
            Twc1.makeTranslate(Tx,Ty,Tz);
            //in order of yaw, pitch, roll (z, x, y in observer frame)
            Rwc1.makeRotate(Rz,Rx,Ry);
            Rwc1.postMultiply(Twc1);
            vx = (double)((int*)buffer)[0]/1000;
            vy = (double)((int*)buffer)[2]/1000;
            vz = -(double)((int*)buffer)[1]/1000;
            wz = -(double)((int*)buffer)[5]/1000;
            wy = (double)((int*)buffer)[3]/1000;
            wx = (double)((int*)buffer)[4]/1000;
            Tc1c2.makeTranslate(vx,vy,vz);
            //in order of yaw, pitch, roll (z, x, y in observer frame)
            Rc1c2.makeRotate(wz,wx,wy);
            Rc1c2.postMultiply(Tc1c2); //makes a proper transform
            Rc1c2.postMultiply(Rwc1) ;
            Rc1c2.getTranslate(&Tx,&Ty,&Tz);
            Rc1c2.getRotate(&Rz, &Rx, &Ry);
            m_pObserver->setPosition(Tx,Ty,Tz);
            m_pObserver->setOrientation(Rz, Rx, Ry);
            }//end got data
        }
        if (tmp == 100)
            quit=1;
        m_pObserver->getPosition(&drBuffer.x1,&drBuffer.y1,&drBuffer.z1);
        m_pObserver->getOrientation(&drBuffer.h1,&drBuffer.p1,&drBuffer.r1);
        vuDistributed::sync( (char*) &drBuffer, sizeof(drBuffer), vuDistrib-
uted::SYNC_LABEL_USER, vuDistributed::WAIT_SLAVES );
```

```
        // If a machine is a slave, take data from the struct and position the
observer.
        if( vuDistributed::getMode() == vuDistributed::MODE_SLAVE )
        {
            m_pObserver->setPosition(drBuffer.x1,drBuffer.y1,drBuffer.z1);
            m_pObserver->setOrientation(drBuffer.h1,drBuffer.p1,drBuffer.r1);
        }
        vpKernel::instance()->endFrame();
    }
    // Shutdown Winsock
    closesocket(theClient);
    closesocket(listeningSocket);
    WSACleanup();
    unconfigure();
    }
    int main(int argc, char *argv[])
    {
    // initialize vega prime
    vp::initialize(argc, argv);
    vuDistributed::setSyncEnable( vuDistributed::SYNC_LABEL_PRE
    _SWAPBUFFERS, true );
    vuDistributed::setSyncEnable( vuDistributed::SYNC_LABEL_POST
    _SWAPBUFFERS, false );
    // ACF synchronization across DR slaves. Setting to "true" will send the
    // master's version of an ACF to the slaves (slaves will ignore their local
    // copy). Setting to "false" will force slaves to use their local copy of
    // the ACF passed into ::define().
```

```
// Default is "true."
vuDistributed::setSyncEnable( vuDistributed::SYNC_LABEL_ACF, true );
// Time synchronization across DR slaves. Setting to "true" will give every
// system the same frame time. Setting to "false" will let each system
// use its system's clock for time.
// Default is "true."
vuDistributed::setSyncEnable( vuDistributed::SYNC_LABEL_TIME, true );
// Window messages synchronization. Setting to "true" will send the
// master's messages to all the slaves. Setting to "false" will prevent
// sending the master's window messages to the slaves.
// Default is "true."
vuDistributed::setSyncEnable( vuDistributed::SYNC_LABEL_MESSAGES,
true );
// Virtual Texture synchronization. Setting to "true" will force all
// systems to page virtual texture at the same time. Setting to "false"
// will cause systems to page virtual texture independently.
// Default is "true."
vuDistributed::setSyncEnable( vuDistributed::SYNC_LABEL_VT, true );
// Synchronize paging. Setting to "true" will force all systems to page
// texture and geometry at the same time. Setting to "false" will cause
// systems to page texture and geometry independently.
// Default is "true."
vuDistributed::setSyncEnable( vuDistributed::SYNC_LABEL_PAGING, true
);
// Synchronize input devices. Setting to "true" will make each system
// react as if the master's input device is connected to the slave.
// Setting to "false" will not pass input device updates from the master
```

```
// to the slaves.
// Default is "true."
vuDistributed::setSyncEnable( vuDistributed::SYNC_LABEL_INPUT, true );
// Synchronize Level of Detail Stress. Setting to "true" will ensure all
// LOD changes are synchronized. Setting to "false" let each system
// determine independently when LOD's should change.
// Default is "true."
vuDistributed::setSyncEnable( vuDistributed::SYNC_LABEL_
                LOD_STRESS, true );
// User defined synchronization. Setting to "true" will enable user
// defined synchronization. Setting to "false" disables all user
// defined synchronization. In this example, disabling user defined sync
// result in different lowest random numbers for each system.
// Default is "true."
vuDistributed::setSyncEnable( vuDistributed::SYNC_LABEL_USER, true );
// create a vpApp instance
myApp *app = new myApp;
// load acf file
if (argc <= 1)
app->define("vpinput_user.acf");
else
app->define(argv[1]);
// configure my app
app->configure();
// runtime loop
app->run();
// unref my app instance
```

```
app->unref();
    Sleep(2000);
// shutdown vega prime
vp::shutdown();
return 0;
}
```

C.2 Program on the Image Processing Machine

```
// Image processing project of the HILS
// Feature Point Tracking, Homography Decomposition and Socket Communi-
cation
#include <atlbase.h>
#include <dshow.h>
#include <stdio.h>
#include <qedit.h>
#include <streams.h>
#include "cv.h"
#include "highgui.h"
#include "video_stream.h"
#include "feature_tracking.h"
#include <Homography/Homography.hpp>
#include <Homography/Homography.cpp>
#include <Homography/Tensor3333.cpp>
#include <Homography/Math-Utility.cpp>
#include <iostream>
#include <fstream>
#include <string>
#include <iomanip>
```

```
#include <cmath>
#include <windows.h>
#include <winsock.h>
#include <stdlib.h>
#include <time.h>
using namespace std;
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
#define USE_SOCKETS
SOCKET theSocket;
char sockbuffer[24];
// this definition allows the filter graph to be remotely viewed using
// the graph edit utility (comment to remove this functionality
// #define REGISTER_FILTERGRAPH
// this definition enables the rendering of the frame grabber output
#define RENDER_OUTPUT
// global variables – video streaming
HWND ghApp=0;// display window handle
IGraphBuilder *pGraph = NULL; // graph builder
IMediaControl *pControl = NULL; // graph control
IMediaEventEx *pEvent = NULL; // event handler
ICaptureGraphBuilder2 *pBuild = NULL;      // capture graph
IVideoWindow *pVidwin = NULL; // video window
IBaseFilter *pSmarttee = NULL; // smart tee filter
```

```
IBaseFilter *pGrabberF = NULL; // sample grabber filter
ISampleGrabber *pGrabber = NULL; // callback interface
PLAYSTATE g_psCurrent = Stopped; // state of playback
DWORD g_dwGraphRegister=0; // name of filter graph
AM_MEDIA_TYPE m_mediaType; // streamed media format
int mediatype = 0; // used in callback function
int first_frame = 0; // used in callback function
CSampleGrabberCB CB; // frame grabber
IplImage *img_prev; // stored previous image
IplImage *frame_scaled;
IplImage *cur_frame;
int frame_count = 0,nret;
BYTE *pData;// Pointer to the actual image buffer
CvSize size, scaled_size;
VIDEOINFOHEADER * vih;
int add_remove_pt=0;
CvVideoWriter* savetestvideo;
// THE HOMOGRAPHY DECOMPOSITION ALGORITHM IS USING 4 POINTS FOR HOMOGRAPHY DECOMPOSITION
Matrix<3,4> pi,ps,psn,mi_star;// current frame feature point coordinates
Matrix<3,4> pi_star,ps_star,psn_star;// previous frame feature point coordinates
Matrix<3,3> Gn, Hn, G_csn;// homography and projective homography matrices
Matrix<3,3> R_bar;// rotation matrix
Vector<4> alpha_g33;// (zi_star/zi)/g33
Vector<3> x_h_bar;// translation vector
```

```
Vector<3> n_starActual,n_star1,n_star2;// normal unit vector (taken as
z-axis)
Vector<3> rpy;
Matrix<3,3> As;
Matrix<3,3> invAs;
double dum, dest = 1;
CvPoint new_pt;
void sleep( clock_t wait );
void on_mouse( int event, int x, int y, int flags, void* param )
{
if( !frame_scaled )
return;
if( frame_scaled->origin )
y = frame_scaled->height - y;
if( event == CV_EVENT_LBUTTONDOWN )
{
new_pt = cvPoint(x,y);
add_remove_pt = 1;
}
}
// returns a vector of RPY angles
// corresponding to the rotational part of the homogeneous transform TR.
void tr2rpy(Matrix<3,3> &R , Vector<3> &rpy)
{
    float sp, cp;
    if ( fabsf( R(1,1) ) < 1e-5 && fabsf( R(2,1) < 1e-5))
    {
```

```
            rpy(1) =0;
            rpy(2) = atan2( R(3,1), R(1,1) );
            rpy(3) = atan2( -R(2,3), R(2,2));
        }else
        {
            rpy(1) = atan2( R(2,1), R(1,1) );
            sp = sin(rpy(1));
            cp = cos(rpy(1));
            rpy(3) = atan2( -R(3,1), cp*R(1,1)+sp*R(2,1) );
            rpy(2) = atan2( sp*R(1,3) - cp*R(2,3), cp*R(2,2) - sp*R(1,2));
        }
        //converting radians to degrees
        rpy=rpy*(180*7/22);
    }
    void ReportError(int errorCode, const char *whichFunc)
    {
        char errorMsg[92];// Declare a buffer to hold
                                // the generated error message
    ZeroMemory(errorMsg, 92);// Automatically NULL-terminate the string
    // The following line copies the phrase, whichFunc string, and integer error-
Code into the buffer
    sprintf(errorMsg, "Call to %s returned error %d!", (char *)whichFunc, error-
Code);
    MessageBox(NULL, errorMsg, "socketIndication", MB_OK);
    }
    // main program
    void main()
```

```
{
    cvInitSystem( 0,NULL );
    remove("final homography output.txt");
    remove("feature_points.txt");
    remove("homography function output.txt");
    createHGWorkspace(4);
    //initilize variables
    // Camera Caliberation Matrix
    G_csn = 0.6507, -0.0282, -82.4109, 0.0085, 0.6380, -61.7579, 0, -0.0001, 1;
    As= 640, 0, 640, 0, 640, 512, 0, 0, 1;
    invAs = inverse(As);
    // Establishing socket communication
    #ifdef USE_SOCKETS
    WORD sockVersion;
    WSADATA wsaData;
    int nret;
    sockVersion = MAKEWORD(1, 1);
    // Initialize Winsock as before
    WSAStartup(sockVersion, &wsaData);
    // Store information about the server
    LPHOSTENT hostEntry;
    in_addr iaHost;
    ////////////////////////////////////
    iaHost.s_addr = inet_addr("192.168.211.12"); //change IP address
accordingly
    ////////////////////////////////////
```

```
hostEntry = gethostbyaddr((const char *)&iaHost, sizeof(struct in_addr),
AF_INET);
if (!hostEntry)
{
    nret = WSAGetLastError();
    ReportError(nret, "gethostbyaddr()");// Report the error as before
    WSACleanup();
    return;
}
theSocket = socket(AF_INET,// Go over TCP/IP
SOCK_STREAM,// This is a stream-oriented socket
IPPROTO_TCP);// Use TCP rather than UDP
if (theSocket == INVALID_SOCKET)
{
    nret = WSAGetLastError();
    ReportError(nret, "socket()");
    WSACleanup();
    return ;
}
// Fill a SOCKADDR_IN struct with address information
SOCKADDR_IN serverInfo;
serverInfo.sin_family = AF_INET;
// At this point, we've successfully retrieved vital information about the
server,
// including its hostname, aliases, and IP addresses. Wait; how could a
single
```

```
// computer have multiple addresses, and exactly what is the following
line doing?
// See the explanation below.
serverInfo.sin_addr = *((LPIN_ADDR)*hostEntry->h_addr_list);
//////////////////////////
serverInfo.sin_port = htons(6262);// Change to network-byte order and
                                  // insert into port field
//////////////////////////
// Connect to the server
nret = connect(theSocket,
(LPSOCKADDR)&serverInfo,
sizeof(struct sockaddr));
if (nret == SOCKET_ERROR)
{
    nret = WSAGetLastError();
    ReportError(nret, "connect()");
    WSACleanup();
    return ;
}
#endif
MSG msg={0};
HINSTANCE hInstance = NULL;
int nCmdShow = 1;
WNDCLASS wc;
cvNamedWindow( "OpenCV", 1 );
cvSetMouseCallback( "OpenCV", on_mouse, 0 );
// initialize the COM library.
```

```
HRESULT hr = CoInitialize(NULL);
if (FAILED(hr))
{
printf("ERROR - Could not initialize COM library");
return;
}
// register the window class (dshow render window)
ZeroMemory(&wc, sizeof wc);
wc.lpfnWndProc = WndMainProc;
wc.hInstance = hInstance;
wc.lpszClassName = CLASSNAME;
wc.lpszMenuName = NULL;
wc.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
wc.hIcon = LoadIcon(hInstance,
      MAKEINTRESOURCE(IDI_VIDPREVIEW));
if(!RegisterClass(&wc))
{
     printf("failed to register window class\n");
     CoUninitialize();
     exit(1);
}
// create the main window. The WS_CLIPCHILDREN style is required.
ghApp = CreateWindow(CLASSNAME, APPLICATIONNAME,
WS_OVERLAPPEDWINDOW | WS_CAPTION |
WS_CLIPCHILDREN, CW_USEDEFAULT,
CW_USEDEFAULT, DEFAULT_VIDEO_WIDTH,
```

```
DEFAULT_VIDEO_HEIGHT, 0, 0, hInstance, 0);
if(ghApp)
{
    // create DirectShow graph and start capturing video
    hr = CaptureVideo();
    if (FAILED (hr))
    {
        CloseInterfaces();
        DestroyWindow(ghApp);
    }
    else
    {
// don't display the main window until the DirectShow preview graph has
been created.
    ShowWindow(ghApp, nCmdShow);
    // this loop handles changes to the render window (close, change size,
etc.)
    while(GetMessage(&msg,NULL,0,0))
        {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
        }
    }
}
// destroy OpenCV preview window
cvDestroyWindow("OpenCV");
// release COM
```

```
    CoUninitialize();
    cvReleaseVideoWriter(&savetestvideo);
    releaseHGWorkspace();
    cvReleaseImage( &frame_scaled );
    //cvReleaseImage( &cur_frame );
}/

//*****Image Capturing Fucntions****
// function for capturing the video from the first available capture device
HRESULT CaptureVideo()
{
    HRESULT hr; // error checking
    IBaseFilter *pCap = NULL; // video capture filter.
    // intialize graph structure
    hr = GetInterfaces();
    if (FAILED(hr))
    {
        printf("failed to get interfaces\n");
        return hr;
    }
    // attach the filter graph to the capture graph
    hr = pBuild->SetFiltergraph(pGraph);
    if (FAILED(hr))
    {
        printf("failed to set capture filter graph\n");
        return hr;
    }
    ICreateDevEnum *pDevEnum = NULL; // device enumerator
```

```
IEnumMoniker *pEnum = NULL; // moniker enumerator
// create the system device enumerator.
hr = CoCreateInstance(CLSID_SystemDeviceEnum, NULL,
    CLSCTX_INPROC_SERVER, IID_ICreateDevEnum,
    reinterpret_cast<void**>(&pDevEnum));
if (SUCCEEDED(hr))
{
    // create an enumerator for the video capture category.
    hr = pDevEnum->CreateClassEnumerator(CLSID_
            VideoInputDeviceCategory,&pEnum, 0);
}
// search for a capture device and choose the first available
IMoniker *pMoniker = NULL;
while (pEnum->Next(1, &pMoniker, NULL) == S_OK)
{
    IPropertyBag *pPropBag;
    hr = pMoniker->BindToStorage(0, 0, IID_I
                    PropertyBag, (void**)(&pPropBag));
    if (FAILED(hr))
    {
        pMoniker->Release();
        continue; // skip this one, maybe the next one will work.
    }
    // find the description or friendly name.
    VARIANT varName;
    VariantInit(&varName);
    hr = pPropBag->Read(L"Description", &varName, 0);
```

```
    if (FAILED(hr))
    {
        hr = pPropBag->Read(L"FriendlyName", &varName, 0);
    }
    if (SUCCEEDED(hr))
    {
        break; // capture device found, quit searching
    }
    pPropBag->Release();
    pMoniker->Release();
}

// add pCap to the filter graph
hr = pMoniker->BindToObject(0, 0, IID_IBaseFilter, (void**)&pCap);
if (SUCCEEDED(hr))
{
    hr = pGraph->AddFilter(pCap, L"Capture Filter");
}
// find available output pin
IPin *pOut = 0;
hr = pBuild->FindPin( pCap, PINDIR_OUTPUT,&PIN_
            CATEGORY_CAPTURE, NULL, TRUE, 0, &pOut );
if (FAILED(hr))
{
    printf("failed to find output pin\n");
    return hr;
}
```

```
// connect frame grabber to available output pin
hr = ConnectFilters(pGraph, pOut , pGrabberF);
if (FAILED(hr))
{
    printf("failed to connect sample grabber\n");
    return hr;
}
#ifdef RENDER_OUTPUT
// render the output of the frame grabber
hr = pBuild->RenderStream(NULL, &MEDIATYPE_Video, pGrabberF,
NULL, NULL);
if (FAILED(hr))
{
    printf("failed rendering the stream\n");
    return hr;
}
// set video window style and position
hr = SetupVideoWindow();
if (FAILED(hr))
{
    printf("couldn't setup video window\n");
    return hr;
}
#endif
pCap->Release();
#ifdef REGISTER_FILTERGRAPH
```

```
        // Add our graph to the running object table, which will allow
        // the GraphEdit application to "spy" on our graph
        hr = AddGraphToRot(pGraph, &g_dwGraphRegister);
        if (FAILED(hr))
        {
            printf("failed to register filtergraph\n");
            g_dwGraphRegister = 0;
        }
        #endif
        // run the filter graph
        hr = pControl->Run();
        if (FAILED(hr))
        {
            printf("couldn't run the graph\n");
            return hr;
        }
        // remember current state
        g_psCurrent = Running;
        return S_OK;
    }
    // function for getting the basic filter interfaces for constructing the filter
graph
    HRESULT GetInterfaces(void)
    {
        HRESULT hr;
        // create the filter graph
        hr = CoCreateInstance (CLSID_FilterGraph, NULL, CLSCTX_
```

```
                    INPROC, IID_IGraphBuilder, (void **) &pGraph);
        if (FAILED(hr))
        return hr;
        // create the capture graph builder
        hr = CoCreateInstance (CLSID_CaptureGraphBuilder2 , NULL,
CLSCTX_INPROC, IID_ICaptureGraphBuilder2, (void **) &pBuild);
        if (FAILED(hr))
        return hr;
        // create the sample grabber
        hr = CoCreateInstance(CLSID_SampleGrabber, NULL, CLSCTX
            _INPROC_SERVER, IID_IBaseFilter, (void**)&pGrabberF);
        if (FAILED(hr))
        {
            printf("could not create the sample grabber\n");
        }
        // obtain interfaces for media control, video window, and sample grabber
        hr = pGraph->QueryInterface(IID_IMediaControl,(LPVOID *) &pCon-
trol);
        if (FAILED(hr))
        return hr;
        hr = pGraph->QueryInterface(IID_IVideoWindow, (LPVOID *) &pVid-
win);
        if (FAILED(hr))
        return hr;
        hr = pGraph->QueryInterface(IID_IMediaEvent, (LPVOID *) &pEvent);
        if (FAILED(hr))
        return hr;
```

```
hr = pGrabberF->QueryInterface(IID_
                ISampleGrabber, (void**)&pGrabber);
if (FAILED(hr))
    return hr;
// set media type for sample grabber
AM_MEDIA_TYPE mt;
ZeroMemory(&mt, sizeof(AM_MEDIA_TYPE));
mt.majortype = MEDIATYPE_Video;
mt.subtype = MEDIASUBTYPE_RGB24;
mt.formattype = FORMAT_VideoInfo;
hr = pGrabber->SetMediaType(&mt);
if( FAILED( hr ) )
{
    printf("failed setting media type\n");
    return hr;
}
// don't do one-shot mode (one-shot mode grabs a single frame and quits)
hr = pGrabber->SetOneShot( FALSE );
if( FAILED( hr ) )
{
    printf("failed setting one-shot mode\n");
    return hr;
}
// don't buffer samples
hr = pGrabber->SetBufferSamples( FALSE );
if( FAILED( hr ) )
{
```

```
            printf("failed setting buffer mode\n");
            return hr;
        }
        // add the sample grabber to the graph
        hr = pGraph->AddFilter(pGrabberF, L"Sample Grabber");
        if (FAILED(hr))
        {
            printf("could not add sample grabber to the graph\n");
        }
        // setup callback function
        hr = pGrabber->SetCallback(&CB, 0 );
        if(FAILED(hr))
        {
            printf("failed to set callback\n");
            return hr;
        }
    // set the window handle used to process graph events
    hr = pEvent->SetNotifyWindow((OAHWND)ghApp, WM_GRAPHNOTIFY,
0);
    return hr;
    }
    // function for configuring the render window
    #ifdef RENDER_OUTPUT
    HRESULT SetupVideoWindow(void)
    {
        HRESULT hr;
        // set the video window to be a child of the main window
```

```
    hr = pVidwin->put_Owner((OAHWND)ghApp);
    if (FAILED(hr))
    return hr;
    // set video window style
    hr = pVidwin->put_WindowStyle(WS_CHILD | WS_CLIPCHILDREN);
    if (FAILED(hr))
    return hr;
    // use helper function to position video window in client rect
    // of main application window
    ResizeVideoWindow();
    // make the video window visible, now that it is properly positioned
    hr = pVidwin->put_Visible(OATRUE);
    if (FAILED(hr))
    return hr;
    return hr;
}
#endif
// function for allowing the render window to be resized
void ResizeVideoWindow(void)
{
    // resize the video preview window to match owner window size
    if (pVidwin)
    {
        RECT rc;
        // make the preview video fill our window
    GetClientRect(ghApp, &rc);
    pVidwin->SetWindowPosition(0, 0, rc.right, rc.bottom);
```

```
        }
    }
    // callback for handeling window events (close, resize, etc.)
    LRESULT CALLBACK WndMainProc (HWND hwnd, UINT message,
WPARAM wParam, LPARAM lParam)
    {
        switch (message)
        {
        case WM_GRAPHNOTIFY:
            HandleGraphEvent();
            break;
        case WM_SIZE:
            ResizeVideoWindow();
            break;
        case WM_WINDOWPOSCHANGED:
            ChangePreviewState(! (IsIconic(hwnd)));
            break;
        case WM_CLOSE:
            // hide the main window while the graph is destroyed
            ShowWindow(ghApp, SW_HIDE);
            CloseInterfaces(); // stop capturing and release interfaces
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
        return 0;
        }
```

```
        // pass this message to the video window for notification of system
changes
        if (pVidwin)
        pVidwin->NotifyOwnerMessage((LONG_PTR) hwnd, message, wParam,
lParam);
        return DefWindowProc (hwnd , message, wParam, lParam);
    }
    // function for handling graph events
    HRESULT HandleGraphEvent(void)
    {
        LONG evCode;
        LONG_PTR evParam1, evParam2;
        HRESULT hr=S_OK;
        if (!pEvent)
        return E_POINTER;
        while(SUCCEEDED(pEvent->GetEvent(&evCode, &evParam1,
&evParam2, 0)))
        {
        // Free event parameters to prevent memory leaks associated with
        // event parameter data. While this application is not interested
        // in the received events, applications should always process them.
        hr = pEvent->FreeEventParams(evCode, evParam1, evParam2);
        // Insert event processing code here, if desired
        }
        return hr;
    }
    // function that stops and starts the rendering
```

```
HRESULT ChangePreviewState(int nShow)
{
    HRESULT hr=S_OK;
    // If the media control interface isn't ready, don't call it
    if (!pControl)
        return S_OK;
        if (nShow)
        {
        if (g_psCurrent != Running)
        {
            // Start previewing video data
            hr = pControl->Run();
            g_psCurrent = Running;
        }
    }
    else
    {
        // Stop previewing video data
        hr = pControl->StopWhenReady();
        g_psCurrent = Stopped;
    }
    return hr;
}
// function that shuts down the filter graph interfaces
void CloseInterfaces(void)
{
    // stop previewing data
```

```
    if (pControl)
    pControl->StopWhenReady();
    g_psCurrent = Stopped;
    // stop receiving events
    if (pEvent)
    pEvent->SetNotifyWindow(NULL, WM_GRAPHNOTIFY, 0);
    // Relinquish ownership (IMPORTANT!) of the video window.
    // Failing to call put_Owner can lead to assert failures within
    // the video renderer, as it still assumes that it has a valid
    // parent window.
    if(pVidwin)
    {
        pVidwin->put_Visible(OAFALSE);
        pVidwin->put_Owner(NULL);
    }
    #ifdef REGISTER_FILTERGRAPH
    // Remove filter graph from the running object table
    if (g_dwGraphRegister)
    RemoveGraphFromRot(g_dwGraphRegister);
    #endif
    // Release DirectShow interfaces
    SAFE_RELEASE(pControl);
    SAFE_RELEASE(pEvent);
    SAFE_RELEASE(pVidwin);
    SAFE_RELEASE(pGraph);
    SAFE_RELEASE(pBuild);
}
```

```
// function for connecting filters in the filter graph
HRESULT ConnectFilters(IGraphBuilder *pGraph,IPin *pOut,IBaseFilter
*pDest)
{
    if ((pGraph == NULL) || (pOut == NULL) || (pDest == NULL))
    {
        return E_POINTER;
    }
    // find an input pin on the downstream filter.
    IPin *pIn = 0;
    HRESULT hr = GetUnconnectedPin(pDest, PINDIR_INPUT, &pIn);
    if (FAILED(hr))
    {
        return hr;
    }
    // try to connect them.
    hr = pGraph->Connect(pOut, pIn);
    pIn->Release();
    return hr;
}
// function for finding an unconnected pin on a filter
HRESULT GetUnconnectedPin(IBaseFilter *pFilter, PIN_DIRECTION
PinDir, IPin **ppPin)
{
    *ppPin = 0;
    IEnumPins *pEnum = 0;
    IPin *pPin = 0;
```

```
// enumerate the pins
HRESULT hr = pFilter->EnumPins(&pEnum);
if (FAILED(hr))
{
    return hr;
}
while (pEnum->Next(1, &pPin, NULL) == S_OK)
{
    PIN_DIRECTION ThisPinDir;
    pPin->QueryDirection(&ThisPinDir);
    if (ThisPinDir == PinDir)
    {
        IPin *pTmp = 0;
        hr = pPin->ConnectedTo(&pTmp);
        if (SUCCEEDED(hr)) // already connected, not the pin we
want.
        {
            pTmp->Release();
        }
        else // unconnected, this is the pin we want.
        {
            pEnum->Release();
            *ppPin = pPin;
            return S_OK;
        }
    }
    pPin->Release();
```

```
    }
    pEnum->Release();
    // did not find a matching pin.
    return E_FAIL;
}
#ifdef REGISTER_FILTERGRAPH
// adds the current filter graph to the Running Object Table
HRESULT AddGraphToRot(IUnknown *pUnkGraph, DWORD *pdwRegister)
{
    IMoniker * pMoniker;
    IRunningObjectTable *pROT;
    WCHAR wsz[128];
    HRESULT hr;
    if (!pUnkGraph || !pdwRegister)
    return E_POINTER;
    if (FAILED(GetRunningObjectTable(0, &pROT)))
    return E_FAIL;
    hr = StringCchPrintfW(wsz, NUMELMS(wsz),
        L"FilterGraph %08x pid %08x\0", (DWORD_PTR)pUnkGraph,
        GetCurrentProcessId());
    hr = CreateItemMoniker(L"!", wsz, &pMoniker);
    if (SUCCEEDED(hr))
    {
        hr = pROT->Register(ROTFLAGS_
            REGISTRATIONKEEPSALIVE, pUnkGraph,
            pMoniker, pdwRegister); pMoniker->Release();
    }
```

```
        pROT->Release();
        return hr;
    }
    // removes a filter graph from the Running Object Table
    void RemoveGraphFromRot(DWORD pdwRegister)
    {
        IRunningObjectTable *pROT;
        if (SUCCEEDED(GetRunningObjectTable(0, &pROT)))
        {
            pROT->Revoke(pdwRegister);
            pROT->Release();
        }
    }
    #endif
    /******* image processing *****/
    // sample grabber (Callback function)
    STDMETHODIMP CSampleGrabberCB::SampleCB( double SampleTime,
IMediaSample * pSample )
    {
        long i = 6000000L;
        clock_t start, finish;
        double duration;
        start = clock();
        if (mediatype == 0)
        {
            HRESULT hr = pGrabber->GetConnectedMediaType
                                        ( &m_mediaType );
```

```
    if(FAILED(hr))
        printf("failed to get connected media type\n");
    else
        mediatype = 1;
}
vih = (VIDEOINFOHEADER*) m_mediaType.pbFormat;
// Get data from rgb24 image
pSample->GetPointer(&pData);
//if variables not allocate them, then do it
if (frame_count==0)
{
    size = cvSize(vih->bmiHeader.biWidth,vih->bmiHeader.biHeight);
    int stride = (size.width*3+3)&-4; // byte offset between rows
    cur_frame = cvCreateImageHeader(size,IPL_DEPTH_8U,3);
    cvSetData(cur_frame,pData,stride);

    scaled_size = cvSize(int(cur_frame->width/img_scale),
                         int(cur_frame->height/img_scale));
    frame_scaled = cvCreateImage( scaled_size,8, 3 );
    savetestvideo= cvCreateVideoWriter("test.avi",
            -1/*CV_FOURCC('P','I','M','1')*/, 30, scaled_size);
}
//reduce size of image to be processed by scale factor
cvResize( cur_frame, frame_scaled, CV_INTER_LINEAR );
// see if previous frame exists
if (frame1_1C == NULL)
{
```

```
        // if previous image does not exist get feature points, but do not
track
        allocateOnDemand( &frame1_1C, scaled_size, IPL_DEPTH_8U, 1);
        cvConvertImage(frame_scaled, frame1_1C, CV_CVTIMG_FLIP);

        cvSmooth(frame1_1C,frame1_1C,CV_GAUSSIAN,3,3);
        // allocate necessary structures for GoodFeaturesToTrack function
        allocateOnDemand( &eig_image, scaled_size, IPL_DEPTH_32F, 1);
        allocateOnDemand( &temp_image, scaled_size, IPL_DEPTH_32F,
1);
        //CvPoint2D32f frame1_features[number_of_features];
        // find good features in the previous image
        int num_features = number_of_features;
    }
    else
    {
        // if previous image exists calculate optical flow
        // get current frame
        allocateOnDemand( &frame2_1C, scaled_size, IPL_DEPTH_8U, 1);
        cvConvertImage(frame_scaled, frame2_1C, CV_CVTIMG_FLIP);
        cvSmooth(frame2_1C,frame2_1C,CV_GAUSSIAN,3,3);
        // create copy of current frame for display
        allocateOnDemand( &frame2, scaled_size, IPL_DEPTH_8U, 3);
        cvConvertImage(frame_scaled, frame2, CV_CVTIMG_FLIP);
        // find additional features if number tracked falls below a threshold
        //CvPoint2D32f frame1_features[number_of_features];
```

```
//printf("tracked_pts: %d, min_features: %d\n", tracked_pts,
min_features);
if (tracked_pts < min_features)
{
    if( add_remove_pt )
    {
        frame1_features[tracked_pts] = cvPointTo32f(new_pt);
        cvFindCornerSubPix( frame1_1C, frame1_features, 1,
            optical_flow_window, cvSize(-1,-1),
        cvTermCriteria(CV_TERMCRIT_ITER|CV
            TERMCRIT_EPS,20,0.03));
        add_remove_pt = 0;
        frame2_features[tracked_pts].x = frame1_
                features[tracked_pts].x;
        frame2_features[tracked_pts].y = frame1_
                features[tracked_pts].y;
        tracked_pts++;
        frame_count=0;
    }
}
else
{
    for (int j = 0; j < number_of_features; j++)
    {
        frame1_features[j].x = frame2_features[j].x;
        frame1_features[j].y = frame2_features[j].y;
    }
```

```
// define necessary structures for CalcOpticalFlowPyrLK
char found_feature[number_of_features];
float feature_error[number_of_features];
// define termination critera for CalcOpticalFlowPyrLK (number of iterations, accuracy to achieve)
CvTermCriteria optical_flow_temination_criteria = cvTermCriteria(CV_TERMCRIT_ITER | CV_TERMCRIT_EPS, 20, .01);
// allocate necessary structures for CalcOpticalFlowPyrLK
allocateOnDemand( &pyramid1, scaled_size, IPL_DEPTH_8U, 1);
allocateOnDemand( &pyramid2, scaled_size, IPL_DEPTH_8U, 1);
// find the correspondence between the good features in both images
cvCalcOpticalFlowPyrLK(frame1_1C, frame2_1C, pyramid1, pyramid2, frame1_features, frame2_features, number_of_features, optical_flow_window, 3, found_feature, feature_error, optical_flow_temination_criteria, 0);

// plot tracked feature points
tracked_pts = 0;
//cvSaveImage("test_img.bmp", frame1_1C);
// REFERENCE PIXEL COORDINATES
// THESE PIXEL CORDINATES CORRESPOND TO FEATURE POINTS WHICH ARE BEING TRACKED
pi =frame2_features[0].x, frame2_features[1].x,
frame2_features[2].x, frame2_features[3].x,
    frame2_features[0].y, frame2_features[1].y,
frame2_features[2].y, frame2_features[3].y,
```

```
        1, 1, 1, 1;
    ps = inverse(G_csn)*pi;
    for (int j = 1; j < 5; j++)
        {
            for (int i=1;i<4;i++)
            psn(i,j)= ps(i,j)/ps(3,j);
        }
    // TO INITIALIZE REFERENCE COORDINATES
    if ( frame_count <= 9)///dues to delay in initial camera adjustments
    {
        pi_star =frame2_features[0].x, frame2_features[1].x,
        frame2_features[2].x, frame2_features[3].x,
                frame2_features[0].y, frame2_features[1].y,
        frame2_features[2].y, frame2_features[3].y,
                1, 1, 1, 1;
        ps_star = inverse(G_csn)*pi_star;
        for (int j = 1; j < 5; j++)
        {
            for (int i=1;i<4;i++)
            psn_star(i,j)= ps_star(i,j)/ps_star(3,j);
        }
        n_old1=0,0,0;//these are global variables
        n_old2=0,0,0;
    }
    //counteract the noise in feature point tracker - enter homography
only when diff. in feature points is greater than a threshold
```

```
if (frame_count>9)//( (abs(pi(1,1)-pi_star(1,1))+ abs(pi(2,2)-
pi_star(2,2))+abs(pi(1,3)-pi_star(1,3))+abs(pi(2,4)-pi_star(2,4)))>3)
{
    // compute homography
if(getHomographySVD(psn, psn_star, Gn, alpha_g33) == -1)
    {
        cout << "Homography determination failed." << endl;
        //break;
    }
//Hn = invAs*inverse(G_csn)* Gn * G_csn*As;
Hn= invAs*Gn*As;
    // Hn = Hn/Hn(3,3);///normalised projective homography
matrix
mi_star = invAs*psn;
if(decomposeHomography2Sol(Hn, mi_star, R_bar, x_h_bar,
n_starActual, dum) == -1)
    {
        cout << "Homography Decomposition failed." << endl;
        //break;
    }
tr2rpy( R_bar , rpy) ;
x_h_bar = dest*x_h_bar/dum;//*alpha_g33(1);
#ifdef USE_SOCKETS
// WRITE HOMOGRAPHY OUTPUT TO A FILE
((int*)sockbuffer)[0] = 5.*x_h_bar(1)*1000;
((int*)sockbuffer)[1] = 5.*x_h_bar(2)*1000;
((int*)sockbuffer)[2] = 5.*x_h_bar(3)*1000;
```

```
((int*)sockbuffer)[3] = 0.01*rpy(1)*1000;
((int*)sockbuffer)[4] = 0.01*rpy(2)*1000;
((int*)sockbuffer)[5] = 0.01*rpy(3)*1000;
nret =      send(theSocket,
            (char*) sockbuffer,
            24,////// Note that this specifies the length of the string;
not
            0);// Most often is zero, but see MSDN for other options
//recv(theSocket,chck,2,0);
if (nret == SOCKET_ERROR)
{
    // Get a specific code
    // Handle accordingly
    return 0;
}
else
{
    // nret contains the number of bytes sent
}
#endif
}
// define line style
int thickness=1;
CvScalar color = CV_RGB(0,255,0);
CvPoint pt1, pt2,pt3,pt4,pt5,pt6;
for(int i = 0; i < number_of_features; i++)
{
```

```
            // remove features outside the image
            // this is necessary because the tracking function creates a repeated
border around the image
            if ((frame2_features[i].x < win)
                || (frame2_features[i].y < win)
                || (frame2_features[i].x > scaled_size.width - win)
                || (frame2_features[i].y > scaled_size.height - win))
                found_feature[i] = 0;
            // plot only if a feature was tracked
            if (found_feature[i] == 0) continue;
            else tracked_pts++;
            // create rectangle coordinates
            pt1.x = (int) frame2_features[i].x - win;
            pt1.y = (int) frame2_features[i].y - win;
            pt2.x = (int) frame2_features[i].x + win;
            pt2.y = (int) frame2_features[i].y + win;
            // plot feature point rectangle
            cvRectangle(frame2, pt1, pt2, color, thickness, CV_AA, 0);
        }
        for (int i = 1; i<5; i++)
            {
                pt3 = cvPoint(((int)pi_star(1,i)- win), ((int)pi_star(2,i)- win));
                pt4 = cvPoint(((int)pi_star(1,i)+ win), ((int)pi_star(2,i)+
win));
                cvRectangle(frame2, pt3, pt4,CV_RGB(255,0,0), thickness);
            }
        }
```

```
        // copy current frame to previous frame
        allocateOnDemand( &frame1_1C, scaled_size, IPL_DEPTH_8U, 1);
        frame1_1C = cvCloneImage(frame2_1C);

        // display image overlayed with optical flow vectors
        cvWriteFrame( savetestvideo, frame2 );
        cvShowImage("OpenCV",frame2);
        }
    frame_count++;
    finish = clock();
    duration = (double)(finish - start) / CLOCKS_PER_SEC;
    //printf( "%f seconds\n", duration );
    return 0;
    }
    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);
```

```
    }
}
```

C.3 Program to Compute Screen-Camera Homography

```
%This program in MATLAB computes the screen-camera homography
clear
clc
close all
% projector_points;
screen_pts
% image_points;
square_pts
histvals=[];
for i=1:1000
[Gr, inliers] = ransacfithomography(scrn_pts',img_pts', .001);
Grs(:,:,i)=Gr/Gr(3,3);
histvals=[histvals inliers];
end
N=hist(histvals, size(scrn_pts',2));
hist(histvals, size(scrn_pts',2));
keepers=[];
for j=1:size(scrn_pts,1)
if N(j)>800
keepers=[keepers j];
end
end
img_pts = img_pts(keepers,:);
scrn_pts = scrn_pts(keepers,:);
```

```
histvals=[];
for i=1:1000
[Gr, inliers] = ransacfithomography(scrn_pts',img_pts', .001);
Grs(:,:,i)=Gr/Gr(3,3);
histvals=[histvals inliers];
end
figure
N=hist(histvals, size(scrn_pts',2));
hist(histvals, size(scrn_pts',2));
keepers=[];
for j=1:size(scrn_pts,1)
if N(j)>800
keepers=[keepers j];
end
end
img_pts = img_pts(keepers,:);
scrn_pts = scrn_pts(keepers,:);

histvals=[];
for i=1:1000
[Gr, inliers] = ransacfithomography(scrn_pts',img_pts', .001);
Grs(:,:,i)=Gr/Gr(3,3);
histvals=[histvals inliers];
end
figure
N=hist(histvals, size(scrn_pts',2));
hist(histvals, size(scrn_pts',2));
keepers=[];
```

```
for j=1:size(scrn_pts,1)
if N(j)>800
keepers=[keepers j];
end
end
img_pts = img_pts(keepers,:);
scrn_pts = scrn_pts(keepers,:);
histvals=[];
for i=1:100
[Gr, inliers] = ransacfithomography(scrn_pts',img_pts', .001);
Grs(:,:,i)=Gr/Gr(3,3);
histvals=[histvals inliers];
end
% Gmean
% Gstd
% Ghmean
% Ghstd
Grmean = mean(Grs,3)
Grstd = std(Grs,1,3)
figure
hist(histvals, size(scrn_pts',2))
```

REFERENCES

[1] M. Aggarwal and N. Ahuja, "High Dynamic Range Panoramic Imaging," *IEEE International Conference on Computer Vision*, Vol. 1, pp. 2-9, (2001).

[2] D. Aiken, S. Gupta, G. Hu, W .E. Dixon, "Lyapunov-Based Range and Motion Identification For A Paracatadioptric Camera", *IEEE Conference on Decision and Control*, San Diego, California, (2006).

[3] S. Baker and S. Nayar, "A Theory of Single-Viewpoint Catadioptric Image Formation," *International Journal of Computer Vision,* Vol. 35, pp. 175-196, (1999).

[4] J. Y. Bouguet, "Pyramidal Implementation of the Lucas Kanade Feature Tracker Description of the Algorithm", Intel Corporation, (2000).

[5] T. J. Broida, S. Chandrashekhar, and R. Chellappa, "Recursive 3-D Motion Estimation From a Monocular Image Sequence," *IEEE Transactions on Aerospace and Electronic Systems*, Vol. 26, No. 4, (1990).

[6] X. Chen and H. Kano, "A New State Observer for Perspective Systems," *IEEE Transactions on Automatic Control*, Vol. 47, pp. 658-663, (2002).

[7] D. M. Dawson, J. Hu, and T. C. Burg, *Nonlinear Control of Electric Machinery*, Marcel Dekker, New York, NY, (1998).

[8] W. E. Dixon, Y. Fang, D. M. Dawson, and T. J. Flynn, "Range Identification for Perspective Vision Systems," *IEEE Transactions on Automatic Control*, Vol. 48, No. 12, pp. 2232-2238, (2003).

[9] O. Faugeras, *Three-Dimensional Computer Vision*, The MIT Press, Cambridge Massachusetts, (2001).

[10] O. D. Faugeras, F. Lustaman, and G. Toscani, "Motion and Structure From Point and Line Matches," *International Conference on Computer Vision*, pp. 25-33, (1987).

[11] M. A. Fischler, R. C. Bolles, "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography," *Comm. of the ACM*, Vol 24, pp 381-395, (1981).

[12] C. Geyer and K. Danilidis, "Catadioptric Projective Geometry," *International Journal of Computer Vision*, Vol. 45, No. 3, pp. 223-243, (2001).

[13] C. Geyer and K. Danilidis, "A Unifying Theory for Central Panoramic Systems and Practical Implications," *European Conference on Computer Vision*, pp. 445-461, (2000).

[14] S. Gupta, D. Aiken, G. Hu, W. E. Dixon. "Lyapunov-Based Range and Motion Identification For A Nonaffine Perspective Dynamic System," *IEEE American Control Conference,* Minneapolis, Minnesota, June, (2006).

[15] R. I. Hartley, "In Defense of the Eight-Point Algorithm," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 19, No. 6, (1997).

[16] E. Hecht and A. Zadac, *Optics*. 3rd ed., Addison-Wesley, (1997).

[17] M. Jankovic and B. J. Ghosh, "Visually Guided Ranging From Observations of Points, Lines and Curves Via an Identifier Based Nonlinear Observer," *Systems and Control Letters*, Vol. 25, pp. 63-73, (1995).

[18] H. C. Longuet-Higgins, "A Computer Algorithm for Reconstructing a Scene From Two Projections," *Nature*, Vol. 293, pp. 133-135, (1981).

[19] B. D. Lucas and T. Kanade, "An Iterative Image Registration Technique with an Application to Stereo Vision, " *7th International Joint Conference on Artificial Intelligence*, 1981.

[20] L. Ma, Y. Chen and K. L. Moore, "Range Identification for Perspective Dynamic System with Single Homogeneous Observation," *IEEE International Conference on Robotics and Automation*, New Orleans, LA, pp. 5207-521, (2004).

[21] L. Ma, Y. Chen and K. L. Moore, "Range Identification for Perspective Dynamic System using Linear Approximation," *IEEE International Conference on Robotics and Automation*, New Orleans, LA, pp. 1658-1663, (2004).

[22] L. Ma, Y. Chen and K. L. Moore, "Range Identification for Perspective Dynamic Systems with 3D Imaging Surfaces," *American Control Conference,* Portland, Oregon, pp. 3671-3675, (2005).

[23] E. Malis and F. Chaumette, "2 1/2 D Visual Servoing with Respect to Unknown Objects Through a New Estimation Scheme of Camera Displacement," *International Journal of Computer Vision*, Vol. 37, No. 1, pp. 79-97, June (2000).

[24] S. Nayar, "Catadioptric Omnidirectional Camera," *IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (1997).

[25] S. Peleg, M. Ben-Ezra and Y. Pritch, "Omnistereo: Panoramic Stereo Imaging," *IEEE Transactions Pattern Analysis and Machine Intelligence*, Vol. 23, No. 3, pp. 279-290, (2001).

[26] J. Shi and C. Tomasi, "Good Features to Track," *IEEE Conference on Computer Vision and Pattern Recognition*, (1994).

[27] J. J. Slotine and W. Li, *Applied Nonlinear Control*, Prentice Hall, Englewood Cliffs, NJ, (1991).

[28] R. Swaminathan and S. K. Nayar, "Nonmetric Calibration of Wide-Angle Lenses and Polycameras," *IEEE Transactions in Pattern Analysis and Machine Intelligence*, Vol. 22, No. 10, (2000).

[29] C. Tomasi and T. Kanade, "Detection and Tracking of Point Features," *Technical Report*, (1991).

[30] R. Y. Tsai and T. S. Huang, "Estimating Three-Dimensional Motion Parameters of a Rigid Planar Patch," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. 29, No. 6, pp. 1147-1152, (1981).

[31] R. Y. Tsai, and T. S. Huang, "Uniqueness and Three-Dimensional Motion Parameters of Rigid Objects with Curved Surfaces," *IEEE Transactions in Pattern Analysis and Machine Intelligence*, Vol. 6, pp. 13-27, (1984).

[32] Y. Xiong and K. Turkowski, "Creating Image-Based Virtual Reality Using a Self-Calibrating Fisheye Lens," *IEEE Computer Soc. Conference on Computer Vision and Pattern Recognition*, (1997).

[33] Z. Zhang and A. R. Hanson, "Scaled Euclidean 3D Reconstruction Based on Externally Uncalibrated Cameras," *IEEE Symposium on Computer Vision*, pp. 37-42, (1995).

BIOGRAPHICAL SKETCH

Sumit Gupta was born in Kolkata, India on December 4, 1981. He received his Bachelor of Engineering degree in mechanical engineering at MS Ramaiah Institute of Technology, India, in May 2004.

Sumit joined the University of Florida in August, 2004 for the Masters of Science degree program in Mechanical Engineering. During his master's program, he worked as a graduate research assistant with Dr. Warren Dixon.

The focus of his research was designing Lyapunov-based nonlinear controllers for range and motion identification using 3D vision systems and implementating visual servo control using image processing and computer vision-based techniques. Sumit plans to join the Test Research and Development group of Intel Corporation.